Statically Contextualizing Large Language Models with Typed Holes

ANONYMOUS AUTHOR(S)

Large language models have changed the landscape for code completion, but the techniques by which prompts are constructed and model responses are handled fail to fully leverage the wealth of structured semantic information available to modern language servers and from there, IDEs. We contend that AIs need IDEs, too! This paper integrates LLMs into the Hazel Assistant, which uses static retrieval and error correction techniques driven by the Hazel Language Server. The Hazel Language Server is able to identify types and typing contexts for holes in program sketches, with total error correction ensuring that a semantically meaningful program sketch is always available. This allows for semantic contextualization with information that is not lexically local to the cursor, but is semantically local to the programmer's intent as expressed through the type system. Contextualized model completions are then iteratively refined via dialog with a language server, providing static error correction. To validate these methods, we present MVUBench, a dataset of MVU applications written in Hazel and TypeScript, particular suited to contrast methods of static retrieval and contextualization. Further, we duplicate our methods and test suite in TypeScript in order to validate the applicability of these methods to higher-resource mainstream languages.

ACM Reference Format:
Anonymous Author(s). 2024. Statically Contextualizing Large Language Models with Typed Holes. 1, 1 (April 2024), 26 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Fig. 1. Hazel Assistant Conversational Architecture (Photo: Corbis/Getty Images, Illustrations: Generated with Dall-e 3)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM XXXX-XXXX/2024/4-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Recent advances in generative AI have triggered an avalanche of new AI programming assistants—the most prominent is Copilot [20], but it has many competitors—that generate code completions by prompting a large language model (LLM) pre-trained on a corpus of diverse natural language documents as well as code written in various programming languages [4, 13, 14, 43, 62, 66]. Once trained, an LLM iteratively transforms an input token sequence, called the *prompt*, into *next-token probability distributions* from which *completions* are sampled. This approach to code completion relies exclusively on learned statistical patterns in the training data [25], so completions are not guaranteed to be syntactically or semantically correct (the **correctness problem**) [50, 53]. AI-generated completions are nevertheless often useful after review and correction [7, 52, 63, 74], particularly (but not exclusively) when working with *high resource* libraries and languages, i.e. those well-represented in the training data [66].

Contemporary AI assistants construct the prompt primarily using the program text appearing in a textual window around the developer's cursor (the *cursor window*) [58]. This approach leads to poor performance in situations where critical task-relevant context comes from definitions that appear neither in the cursor window nor in the training data (the **semantic contextualization problem**) [2, 17, 34, 51, 59, 65, 70, 71]. For example, consider the following cursor window, which would arise when a developer is implementing a GUI component using the model-view-update (MVU) architecture (central to popular GUI application frameworks like React [5] and Elm [15]):

```
// update the room booking data after a user action
function update(model: Model, action: Action): Model {
```

Correctly completing this function requires knowing the definitions of this specific component's `Model` and `Action` types, which commonly appear in different files in the repository and therefore outside the cursor window. Various other files might also contain relevant definitions, e.g. other types that can be reached from the definitions of `Model` and `Action` and useful helper functions for working with values of these types. Without access to these definitions, an LLM will either be unable to generate sufficiently probable completions (which may result in no completion being generated) or it will hallucinate plausible but nearly-always incorrect definitions based only on the provided comment [68, 72].

To address this problem, assistant designers use various retrieval augmented generation (RAG) techniques [22] to retrieve additional code from other files in the repository and external libraries for inclusion in the prompt. Real-world code bases often span hundreds of thousands of lines of code, so exhaustive retrieval quickly runs into scaling issues. While prompt (i.e. context) size limits continue to increase [18], generation costs (measured in both time and energy) scale with token count [26]. These costs are substantial (because LLMs typically have billions of parameters) so *token efficiency* remains a critical metric [58]. Moreover, contemporary LLMs struggle to attend to relevant information and ignore irrelevant information (such as the `Model` and `Action` types for *other* GUI components) in large prompts [31, 33, 36, 67].

Given these issues, assistant designers need retrieval techniques that prioritize task-relevant code. For example, Copilot retrieves code from locations in other files that the developer has recently visited, based on the heuristic that these are likely to be task-relevant [61]. Another prominent retrieval strategy, which we will refer to as *vector retrieval*, involves performing a vector search across the repository (and perhaps beyond) to retrieve code similar to the code in the cursor window [39, 70, 71], as measured by a learned vector similarity metric [32, 43]. This relies on the heuristic that similar code is likely to be task-relevant code. In the example above, since the type names `Model` and `Action` appear explicitly in the cursor window, this approach *may* find their definitions. However, it may also find irrelevant definitions of other types named `Model` and

`Action` and other implementations of `update`, e.g. those from other GUI components in this or other applications. It may also be less effective when the task-relevant definitions are not named explicitly, e.g. if the developer is later writing a call to `update`, the fact that the relevant types are `Model` and `Action` requires reasoning about the type signature of the `update` function.

These retrieval approaches are language-agnostic, treating source code as a sequence of tokens like any other, so they must necessarily deploy imprecise heuristics. In this paper, we instead consider language-aware approaches that take advantage of the fact that in many languages, code is governed by a rich type and binding discipline, i.e. a static semantics [24, 54]. To retrieve relevant semantic information and analyze candidate code completions, we rely on a modern language server [8, 10] to provide various *semantic services*, namely type reporting, typing context search, and error reporting. Integrated development environments (IDEs) interact with these services to drive various human-facing affordances, e.g. type hints and hover messages. The hypothesis that we investigate is that LLM code completion can also benefit from interactions with semantic services. Put pithily, we hypothesize that *AIs need IDEs, too.*

We investigate two language-aware approaches independently and in combination.

Static Retrieval. The first approach we consider, in Sec. 2, is static retrieval, wherein we ask the language server to (1) determine the type and typing context of the "hole" (here implicit) at the cursor, and (2) transitively retrieve semantically relevant type definitions and function headers, from wherever they might appear, for inclusion in the prompt.

In the example above, the hole that the language server inserts (either implicitly or explicitly) at the cursor is of type `Model` (because it is in the body of the `update` function, which has return type `Model`), and the local typing context includes the argument `action : Action`, so the language server can look up the definitions of the `Model` and `Action` types. These types might themselves refer to other types, so we can transitively continue type retrieval. We can also retrieve information about relevant helper functions in the typing context, e.g. those that operate on the types that have been looked up, continuing transitively up to a token limit.

Static Error Correction. To further improve correctness, we combine static retrieval in Sec. 2.7 with a straightforward syntactic and static error correction pass: we ask the language server to localize and generate error messages for syntactic and static errors in the generated completion, then feed this information back into an instruction-tuned model, prompting it to correct these errors, potentially over multiple rounds (which requires trading off latency for correctness).

1.1 Evaluation Overview

1.1.1 Programming Languages and Language Servers. The approaches that we investigate in this paper are in principle applicable to any programming language with a well-structured type and binding discipline. The more challenging requirement, which has limited the prior experiments in these directions (which we review in Sec. 5), is that we also need a rather capable language server. In particular, the language server must be capable of robust syntax error and type error recovery, producing a semantically meaningful program sketch (i.e. a program with *holes*) in any situation where the developer might request code completion [46]. For instance, the example situations above are syntactically erroneous, because the developer has not yet closed a curly brace or parenthesis, so a standard compiler would simply report a syntax error and halt. In other cases, there may be localized type errors elsewhere in the program that should ideally not cause gaps in the availability of code completion.

For this reason, we evaluate these ideas primarily by developing an AI programming assistant for Hazel, extending the Hazel Assistant with LLM support [9]. Hazel is a typed functional programming environment designed specifically around typed holes, inserting them fully automatically to

ensure total syntax error recovery [41] and total type error recovery [73]. Hazel is also capable of evaluating programs with holes [44, 69] (including the holes that are inserted as membranes around marked type errors [73]), which makes it possible to use unit testing to granularly evaluate the correctness of even locally ill-typed model outputs (rather than the more *ad hoc* methods that are common in the literature, like edit distance from a single canonical solution).

The Hazel language is similar to Elm, OCaml, and other languages in the ML family, i.e. it is a pure typed functional language with support for algebraic datatypes and pattern matching. Unlike Elm and OCaml, contemporary LLMs have not been trained on a substantial body of Hazel code, i.e. Hazel is a *low resource language*. This presents both a challenge and an opportunity for research. We have found that when asked to write Hazel code, contemporary LLMs fail to follow Hazel's syntax and semantics, often borrowing syntactic forms and library functions from OCaml and Elm. However, LLMs are capable of in-context learning [21, 56], suggesting that it is possible to include few-shot examples and instructions in the prompt to quickly teach contemporary LLMs about how Hazel differs from related higher resource languages. The error correction approach we investigate may also be of particular interest in preventing errors in this sort of low-resource setting, which is of considerable interest to the PL research community [12].

To demonstrate that static retrieval is useful even for mainstream high resource languages, we also perform additional more limited experiments in Sec. 3 with TypeScript via the TypeScript Language Server, an instance of the Language Sever Protocol (LSP) [8]. We find that the LSP does not provide simple, direct access to the sort of information that is necessary to implement static retrieval, so we propose a more direct interface as we introduce the various approaches throughout the paper.

1.1.2 Tasks. The most commonly reported LLM code completion benchmarks are HumanEval [14], EvalPlus [35], MBPP [4], and LiveCodeBench [28]. These are unsuitable for evaluating the proposed approaches because they consist of single-file tasks (constructed either manually, or derived from public repositories or programming contests) that are low-context, i.e. they require only builtin or standard datatypes, and therefore do not highlight the semantic contextualization problem.

Repository-level benchmarks like RepoEval [71], RepoBench [37], CrossCodeEval [16], the CoCoMIC dataset [17], and defects4j [30] are more suitable because they are high-context, i.e. they require completing code that depends on definitions in different files. However, these benchmarks also present difficulties:

(1) **Data Contamination**: LLMs are known to be able to memorize code that they have seen during training, and evidence suggests that this data contamination issue has indeed caused public models to be overfitted to publicly available code [28, 57]. All of these benchmarks source examples from GitHub or PyPI. Based on reported cut-off dates, the projects in these benchmarks will have by now likely been incorporated into the training of contemporary models (noting that with few exceptions discussed below, training data is not disclosed).
(2) **Language Exclusivity**: None of these include Hazel code, nor is it easy to manually port arbitrary projects taken from GitHub or PyPI that depend on various complex libraries to Hazel or any other low resource language of interest to the community. Existing porting techniques have only generated low-context datasets [12]. In addition, RepoEval and CoCoMIC exclusively feature Python code, which is difficult to statically analyze.
(3) **Missing Tests**: None of the tasks in these benchmarks include unit tests, which means that we can only evaluate correctness based on brittle textual similarity metrics. We observe that often produce correct output with substantial syntactic variation, similar to human programmers [23].

For these reasons, we construct a new repository-level benchmark suite, MVUBench, in Sec. 2.8.1, that consists of various MVU web applications. Web application development is an important and under-studied application domain (many visions for the future of AI imagine LLMs generating complete application logic, not just solving code competition problems [64]). MVU applications are high-context in that they must necessarily define a number of different datatypes which by convention are often located in separate files. Some of these datatypes have generic names, like `Model` and `Action`, and a single application might have multiple such types, one for each GUI component, presenting a significant challenge to language-agnostic techniques that do not understand binding structure. Indeed, it is easy to construct particularly challenging yet realistic examples simply by combining multiple components of MVUBench, as we demonstrate in Sec. 2.8.5.

We address the data contamination problem following the approach taken by HumanEval, by conceptualizing and implementing these applications from scratch, without directly adapting any code from GitHub. We will control the release of these benchmarks to limit the likelihood of future contamination [27]. We address the language exclusivity problem by ensuring that these MVU applications do not have any external library dependencies, so it is easy to port them to various languages other than Hazel and TypeScript, notably including pure functional languages. New MVU examples are also easy to develop and add to the benchmark, because they can implement the logic of essentially any front-end web application or GUI component. Finally, the lack of side effects also makes it easy to unit test the core application logic.

1.1.3 LLMs. We selected two pre-trained language models with which to perform experiments. First, we selected OpenAI's GPT-4(-0613) [1], which is consistently at or near the top of code completion benchmarks, to evaluate whether even the most capable contemporary foundation models (i.e. models so large that only large organizations like OpenAI have the resources to train and deploy them) benefit from the approaches we consider.

GPT-4 is a closed model and many of its specific details, including its size and training, have not been publicly disclosed. This presents a significant challenge to reproducibility. Consequently, we also conduct experiments using StarCoder2-15B, the most capable fully open-source model responsibly trained on a fully open code corpus, The Stack v2, as of this writing [38]. The Stack v2 has notably extensive coverage of low resource languages. StarCoder2-15B is small enough to run locally on sufficiently powerful workstations, so our results should be reproducible indefinitely.

GPT-4 is an instruction-tuned model, so we are able to use it to evaluate static error correction. StarCoder2-15B is completion-tuned, meaning it is not designed to receive and respond to instructions. We are not aware of a comparably powerful fully open source model that is instruction-tuned as of this writing, so we do not evaluate error correction with StarCoder2-15B.

2 STATIC RETRIEVAL AND ERROR CORRECTION IN THE HAZEL ASSISTANT

In this section, we will first introduce Hazel and the Hazel Assistant by example from the developer's perspective. We continue by describing how the Hazel Assistant prompts GPT-4 and interfaces with the Hazel Language Server to generate code completions augmented with static retrieval and error correction. Finally, we will introduce the MVUEval benchmark and report on the results of an ablation study of each of these features to investigate their relative contributions to the overall performance of the assistant on these high-context MVU tasks, relative to various baselines that establish lower-bounds and upper-bounds on performance.

2.1 Hazel

Hazel is a web-based live functional programming environment that features total syntax and type error recovery via automatic hole insertion [41, 73]. This ensures that every editor state in Hazel is

Fig. 2. The Hazel IDE

a semantically meaningful program sketch and that Hazel's various editor services, include code completion, never experience gaps in service [46].

As a running example, we consider a scenario where the developer is implementing EmojiPaint, a simplified MVU app where a user picks an emoji from a palette and creates designs by 'stamping' it on a grid. Fig. 2 shows the user editing the file where the `update` function is defined. In Fig. 2 the developer's cursor is shown as a red convex triangle to the right of a hole, shown as a small convex hexagon. The Hazel IDE interfaces with the Hazel Language Server to report static information which both user and model will use to inform completions. The bottom bar, called the Cursor Inspector, reports information on the syntax as well the expected type of the expression, here a typed hole of the function type `(Model, Action) -> Model`, at the cursor. The lower left popup, called the Context Inspector, reports the typing context at the cursor.

The `update` function is intended to respond to the EmojiPaint user's GUI actions, represented by values of type `Action`, by transforming the current GUI state, represented by values of type `Model`, to a new GUI state, also of type `Model`. These types and associated helper functions appear in different files, excerpted in Fig. 3(a-b). Algebraic datatypes, i.e. recursive sum types, in Hazel separate constructors with + (rather than the more common | in related languages like OCaml).

Fig. 3. (a) The types relevant to the EmojiPainter MVU app (b) An excerpt of already-implemented functions in another file (with definitions collapsed by Hazel's outliner). (c) The developer has requested LLM completion by inserting ?? in the hole.

2.2 Hazel Assistant

The Hazel Assistant is a programming assistant that generates code completions by two mechanisms. To provide fast, local completions, the Hazel Assistant generates type-directed completions [9], using localized syntactic and static information to inform small completions with type-directed lookahead as shown in Fig. 4(a-b). This feature can be invoked even when there are syntax errors because Hazel tracks syntactic obligations in a backpack, e.g. as shown in Fig. 4(b) where both => and **end** are necessary to complete the case expression [41].

Fig. 4. The Hazel Assistant defaults to providing only type-directed token completion.

To request LLM completion from the Hazel Assistant, the developer can fill any expression hole with ?? which starts to animate as suggested in Fig. 3(c). GPT-4, our underlying model in this section, is not particularly performant as of this writing, so the developer can continue to edit elsewhere while waiting for GPT-4 to return a completion. For this example, Fig. 5 shows an example of a GPT-4-generated completion. The developer can inspect this completion (which would display any type errors if found) and accept it with the Tab key.

Fig. 5. The Hazel LLM assistant, combining static information with generative creation via language models, is capable of offering more substantial completions

2.3 The Hazel Assistant Trialogue

Our generative hole filling process consists of the following steps, construed here as a "trialogue" between programmer, the Hazel language server, and a language model. This is depicted in figure 1 as a series of messages. In our setup, the primary interaction is between the language model and the Hazel Language Server, which acts on behalf of the user in response to a request for LLM hole filling, kicking off the following exchange. In the following, we use the messaging terminology from the OpenAI GPT-4 API, which distinguishes System, User, and Model messages:

(1) *System Message*: Hazel Crash Course and few-shot examples
(2) *User Message*: Program sketch augmented by static retrieval, i.e. relevant semantic context information from the language server

(3) *Model Message*: Suggested hole filling
(4) *User Message*: Syntax and type errors in the completion
(5) *Model Message*: A corrected completion if necessary

We repeat steps 4-5, i.e. we perform syntactic and static error correction, if necessary (stopping after at most 2 iterations to limit latency).

2.4 System Message: The Hazel Crash Course

The system message is generic, common to each prompt. For an instruction-tuned model (GPT-4), the system message consists of three portions.

First, we provide a list of instructions which delineate the task. In particular, this instructs the model to provide a code fragment which replaces a sentinel value representing the target hole in the program sketch. For example, the model is given the instructions:

- `"Reply only with code"`
- `"DO NOT include the program sketch in your reply"`

Second, an informal specification of the Hazel syntax, with emphasis on 'negative characterization' of hazel syntax, listing differences from higher resource syntactically similar languages. This section was constructed though an ad-hoc process of discovering repeated syntactic errors in model output. For example:

- `"No 'rec' keyword is necessary for 'let' to define a recursive function"`
- `"There is no dot accessor notation for tuples; use pattern matching"`

Finally, we positively characterise hazel syntax via a list of several input/output pairs of sketches and program completions (few-shot prompting). We show one example below.

- Sketch:
```
let List.length: [(String, Bool)] -> Int =
  fun xs -> ?? end in
```
- Completion:
```
case xs
| [] => 0
| _::xs => 1 + List.length(xs)
end
```

For the completion model (StarCoder2-15B) discussed below, which has a longer context window (16k versus 8k for GPT-4) but is not instruction-tuned we omit the first two sections, in lieu of providing a longer list of syntax examples, which are provided simply as a list of definitions rather than input-output pairs.

2.5 Static Retrieval

The base EmojiPaint update function program sketch is as follows:

```
"Update the EmojiPaint app model based on an action"
let update: (Model, Action) -> Model = ?? in
```

We augment the sketch provided by the developer with additional static information obtained via the Hazel Language Server, serialized into text, essentially constituting a simple static program slice. Specifically, we retrieve the following static information, shown diagrammed in Fig. 6.

- *Type Retrieval*: The expected type at the cursor, along with the definitions of any types aliases occurring in that type, and the definitions of aliases recurring in that definition, and so on recursively until we hit base types.

Fig. 6. The programmer requests a hole filling (A) by entering '??', either intentionally or in a fit of frustration. The Hazel Assistant queries the Hazel Language Server (B) and collates relevant semantic information (D & E) into a text prompt, which is dispatched (F) to the language model

- *Header Retrieval*: A selection of values, annotated with their types, filtered from the typing context based on a type-directed metric of relatedness to the expected type described below.

The language server augments the initial program sketch with semantic information localized to the program hole at the cursor. While we describe our particular approach in Hazel by way of illustration, our goal is to outline an approximate API which could be implemented by any language server for a typed language which could drive a similar system in another language.

2.5.1 Relevant Type Definitions. Given the above program sketch, the expected type of the hole ?? is `(Model, Action) -> Model`. While in this case adding this type information to the prompt is strictly redundant, as it already appears as the function's type annotation, in generate Hazel's bidirectional type system [45, 73] allows a similar expected type to be extracted in any position for which there exist type constraints, e.g. in function argument position or in a module signature governing the module in which `update` exists as is common in languages like OCaml.

Absent this sort of context, this type is very elucidatory on its own. Based on the provided comment, a language model might and likely will guess that these refer to the state and state changes of an Model-View-Update application. But as we shall see, it is vanishingly unlikely to guess the precise structures of the types the programmer has actually used. An example demonstrating the common case is in Figure 7.

Hence we do automatically what a programmer in an unfamiliar codebase might do manually in a chat-based system like ChatGPT: recursively pursue type definitions to unwind the local semantic context hinted at by the type expectation. Providing this list to a language model is analogous on an IDE user interface level to hovering over types and jumping recursively to their definitions.

First, we extract relevant type aliases from the expected type:

- `Model`
- `Action`

Then, we retrieve their definitions:

```
let update: (Model, Action) -> Model =
  fun (model, action) ->
    case action
      | Paint(emoji) => {model with tool = Some(emoji)}
      | Clear => {model with canvas = blank_canvas; tool = None }
      | Undo => {model with canvas = do_undo(model.undo_stack) }
    end
 in
```

Fig. 7. A typical completion with no static retrieval. Here, the language model hallucinates plausible but incorrect constructors for the Action type, and hallucinates Model as a record type using syntax that is not not supported in Hazel.

- **type** `Model = (Grid, Emoji, [Emoji])`
- **type** `Action = SelectEmoji(Emoji) + StampEmoji(Row, Col) + ...`

And finally, we recurse on the process, retrieving any additional aliases which occur in those definitions; in this case, the aliases *Grid*, *Emoji* from the *Model* definition and the aliases *Row*, and *Col* from the *Action* definition:

- **type** `Emoji = String`
- **type** `Row = Int`
- **type** `Col = Int`
- **type** `Grid = [[Emoji]]`

Note that even though these type definitions are, from the point of view of the Language Server, abstract entries in the typing context, we are specifically electing to reproduce them in format that resembles their original lexical concrete syntax. We have noticed that attempting this kind of naturalistic reproduction increases that chance that language model generations stay on-task, generating code in the relevant concrete syntax without recoursing to prose or other ill-formed syntax.

To support the above method, a language server needs to implement the following API, which may be implementable as thin glosses on top of existing methods such as 'Go to type definition':

- `getExpectedType: (Program, LexicalLocation) -> Type`
- `extractAliases: Type -> [Type]`
- `getTypeDefinition: TypeAlias -> Type`

To briefly contrast this process to standard Retrieval-Augmented Generation:

- Although RAG might flag these definitions as possibly relevant, note that they are necessarily relevant. Our knowledge of the language semantics means we know that any completion must necessarily respect these types, an assurance which allows us to offload burden from the more expensive and imprecise associative lookup.
- This necessary relevance also significantly increases the chance that subsequent recursive retrievals will be relevant
- The lookups necessarily respect scope. RAG might return reasonably-related definitions, or definitions of similar names, but without special knowledge of the language semantics, we are not guaranteed that these will be *the* definitions relevant to this lexical context.

2.6 Relevant Headers from the Typing Context

In additionally to relevant type definitions, we augment the prompt with the names and types (collectively, the headers) of relevant values, typically functions, from the typing context. From a

user interface perspective, this is analogous to a type-directed autocomplete menu. Note however that it must suffice for the entire generation, not simply the completion of the next token.

Our method divides into three stages:

- Use the expected type to identify a list of target types
- Filter the typing context for values whose types stand in a certain relation to these target types
- Assign scores to each element of the resulting list, and return the prefix of that list truncated at some scoring and length thresholds (here, >0.0 and 10 items respectively)

The resulting context entries are formatted as code sketches to simplify language model ingestion, for example the pair:

```
("string_of_int", Int -> String )
```

is formatted as:

```
let string_of_int : Int -> String =  in
```

Here, the body of the definition is simply omitted. Interestingly, we originally used ellipsis (...) in place of the body, but this resulted in an increased chance the model would itself emit the token ... in lieu of completing the implementation.

2.6.1 Identification of Target Types. The first stage, identification of target types, proceeds by deconstructing the expected type to identify relevant sub-components which could be used, in conjunction with their relevant elimination forms, to construct the target. Our first target type is simply the type of the hole itself.

Target Types = `(Model, Action) -> Model`

Then, if that type is a compound type, we consider its components. In particular, if the type is a product type, we consider its components to be targets, and if the type is an arrow type (as it is here), we consider its return type to be a target.

Target Types = `(Model, Action) -> Model, Model, ...`

In principle, we could continue this deconstruction recursively indefinitely, but for our immediate purpose of identifying likely-relevant types, we've found it suffices to extend one more iteration; that is, product/arrow types containing product/arrow types. It simplifies our calculations to internally normalize all type definitions (Hazel is structurally typed). Here, we will only do so opportunistically for clarity of presentation:

Type of Hole: `((String, (Grid, Emoji, [Emoji])), Action) -> (Grid, Emoji, [Emoji])`

Target Types : `(Model, Action) -> Model, Model, Grid, Emoji, [Emoji]`

One can easily see other ways of extending target type identifications. Possibilities included more sophisticated compound types such as records, or for function types, also considering the input types as a kind of negative target, in that we may want to prioritize types that consume a relevant type from the local context. For now we proceed with the simple approach outlined.

We do not, however, return unaliased base types such as `Bool` or `String` as target types. Early experimentation indicated that, given that there are large amounts of functions on base types, often with no a priori way to distinguish their relevance based on types, such functions would often act as conflaters, since which happened to be included was essentially incidental, sometimes decreasing the change an appropriate base type function would be guessed by the language model. In practice, a standard library would already be well-understood by an LLM from the pre-training or fine-tuning step. For Hazel, we replicate much of the OCaml standard library to sidestep this need.

2.6.2 Filtering the Context. For each target type, we filter the typing context to retrieve types which can be used, again in conjunction with appropriate elimination forms, to produce the target.

This is essentially similar to target type extraction. In particular, we return types which are consistent with the target, arrow types whose return type is consistent with the target, product types whose have a component consistent with the target. For example:

Target Type: `Grid`

Yields:

- `updateGrid: (Grid, Row, Col, Emoji) -> Grid`
- `clearGrid: Grid -> Grid`
- `fillRowInGrid: (Grid, Row, Emoji) -> Grid`

As the target type is consistent with these functions' return types.

2.6.3 Sorting and scoring the filtered context. Prior work [9] has surveyed various ways in which semantic information can be used to sort typing-context-originating variable suggestions for relevance. For our purposes here we use a simple scheme intended as a proof of concept to establish a baseline for more sophisticated methods

By default, Hazel context entries are sorted by locality of definition, which provides a reasonable default for relevance. Thus we sort in a way that respects this original ordering for context entries having the same score.

Our scoring scheme is simple. By default, all entries are assigned a score of 1.0. However, if a type contains the (gradual) unknown type, a multiplier is applied based on the ratio of unknown to known type constructors in the type (for example, the type [?] (i.e. a list of unknown type) would receive a multiplier of 0.5, since the list constructor is known). This acts simply to de-prioritize incomplete implementations, about which not enough information is available to make it a good suggestion.

These are the relevant values from the EmojiPainter example:

```
"Consider using these variables relevant to the expected type:"
let model_init: Model =  in
let fillRowInGrid: ((Grid, Row, Emoji) -> Grid) =  in
let clearGrid: (Grid -> Grid) =  in
let updateGrid: ((Grid, Row, Col, Emoji) -> Grid) =  in
```

In order to implement relevant context extraction in an arbitrarily language server, one might wish to model the above process through the following API:

- `getTargetTypes: Typ -> [Typ]`
- `filterCtx: Ctx, Typ -> [(Name, Typ)]`
- `scoreEntry: (Name, Typ) -> Float`

2.7 Syntactic and Semantic Error Correction

The use of instruction-tuned language models makes available a lightweight form of program repair based on an iterative loop of generated completions and the error messages emitted by compilers or other means of static analysis. The general technique of looping LLM code generation on compiler errors appears to have emerged essentially in tandem with LLM code generation itself (cite some blogs) and has been examined in greater detail by Joshi et al. [29].

After receiving a response from the model, we substitute the received completion into the original program sketch. We then query the Hazel language server to parse the resulting program. Hazel is very liberal about what is considered syntactically meaningful, in the sense that a program can be partially parsed even in the presence of malformed or missing tokens, which is often

sufficient to typecheck and run the program, even though it will be incomplete. If there are no fatal parse errors, we query the language server for a list of static errors, which include non-fatal parse errors and type errors. If there are any such errors, we serialize them to a string, and send them to the model.

In order to maintain model context, we append the errors to a growing alternation of messages begun with the original prompt. The number of correction rounds which can be performed in this way is thus limited by the length of the context window; in our case, using the 8k token window of GPT4-0613, we are effectively capped at 5 rounds. However, we have noticed that 2 rounds are often sufficient to eliminate static errors, and that rounds in excess of 2 tend to show diminishing returns. Given these observations, coupled with the additional time and cost involved, we have capped our maximum number of rounds at 2.

To support this, we need a language server to be able to localize static errors, reporting locations and error messages in the usual way. Ideally, we receive a total set of error messages, rather than just the first error message, which is supported by Hazel's total type error localization and recovery system [73].

- `getStaticErrors: Expression -> StaticErrors`

2.8 Experimental Evaluation

We would next like to evaluate the basic effectiveness of this method of using static information to proactively contextualize and retrospectively correct language model code completion.

2.8.1 MVUBench. Hazel is a low-resource language, so we are unable to conduct an at-scale evaluation in this context. Instead as previously motivated, we construct a benchmark suite of five MVU applications similar to the EmojiPainter example just described:

- Todo (*TO*): Maintains a list of tasks
- Room Booking (*BO*): Manages a room booking schedule
- Emoji Painter (*EM*): Paint emoji stamps on a small canvas
- Playlist Manager (*PL*): Manage a music playlist
- Password Strength Checker (*PA*): Rate a password via a dynamic set of properties

The baseline program sketch provided to the language model for each of these programs is simply the type-annotated function header for the `update` function, along with a single-line comment describing the function's general purpose, including the name of the application, in line with the running example above.

Each application also comes with a simulated repository containing other relevant type and accompanying helper function definitions.

We also write a small test suite for each example, consisting of 10-15 tests that ensure that each MVU action behaves as a human user might reasonably expect without additional specification.

In such a situation, the language model completion is informed only by the `update` function type names (which are non-specific) and the single-line comment (which only hints at the intended functionality). While it is still possible that in very prosaic situations, the model will guess at the appropriate types and names, more likely (as we shall see) it will hallucinate extensively. By varying the methods through which additional context is provided, and the corrective methods applied to resulting completions, we provide a baseline analysis for the relative effects and interactions of these methods on LLM code completion.

2.8.2 Feature Ablation Experiment. Our main experiment consisted of 320 completions trials, each of which makes 1-3 calls to the language model. These 320 trials divide as follows:

- 8 feature ablation configurations:

  - *Type Retrieval*: include expected type and type definitions as described above
  - *Header Retrieval*: include relevant headers from the typing context as described above
  - *Error Rounds*: whether to perform up to 2 static error correction rounds as described above
- 5 program sketches (TO, BO, EM, PL, PA)
- 20 completion trials per combination (to account for model non-determinism [48])

2.8.3 Comparison Baseline 1: No Context. The ablation configurations lacking all static retrieval feature serves as a lower bound baseline – without any context except the brief comment on the `update` function, we would expect even high performing models to perform poorly due to lack of semantic context. This is the current situation in AI programming assistants that do not attempt repository-level retrieval.

2.8.4 Comparison Baseline 2: Exhaustive Retrieval. An additional baseline configuration, beyond those outlined above, is to perform exhaustive retrieval of all application code, excluding tests. This serves as a token-inefficient upper bound on performance.

2.8.5 Comparison Baseline 3: Vector Retrieval with Confounds. Finally, we compare our approach to vector retrieval. Given that our test corpus consists of 5 relatively small programs, we have emulated a larger more realistic codebase by combining those five programs, minus tests and update functions, along with Hazel's standard library, to create a 1000-line simulated codebase from which context can be drawn. Combining these programs together has the effect of creating some possible associative conflaters, e.g. two types having the same name; we contend that this, albeit synthetic, nonetheless emulates a legitimate source of confusion for a scope-unaware method such as vector retrieval.

We have used the simplest standard RAG strategy, uniformly dividing the codebase into 150-character chunks, which were then submitted to OpenAI's Ada (text-embedding-ada-002), a commercial embeddings model [43]. The retrieved 1536-element vectors, along with their associated text chunks, are then stored locally in a JSON file acting as a basic vector database.

In order to retrieve the chunks most similar to a provided sketch (our function headers and comments), we submit that header to the same API endpoint, and then search the vector database for the top 6 entries with the highest cosine similarity [43].

The above parameters (150 character chunks, 6 entries) are chosen so that the total (900 characters) lines up with the average length of the total typed semantic RAG context (types + relevant context) of our 5 examples, with the chunk size being set as small as possible while still being able to fully contain approximately 2 lines of code, sufficient for most type definitions in our corpus.

It should be noted that there exist a plethora of more advanced chunking strategies which may yield better results, including overlapping chunks, and semantic chunking which takes into account source syntax. However, all these strategies have complex trade-offs which take us beyond the immediate comparisons at hand; for example, chunking by top-level definitions (a language-aware approach) might prevent issues with only a partial definition being included in a prompt, but seeing as definitions can range widely in size, being forced to include an entire definition may prevent multiple chunks which are together more relevant from being included.

As such, we have elected to leave the RAG setup structurally agnostic, so as to more cleanly contrast it with semantic methods, while noting it is likely that ultimately these two methods are not exclusive and can be used synergistically in a production setting (for example, balancing the ratio of typed semantic versus associative RAG depending on the amount of static information available at a given lexical location).

2.9 Hazel GPT-4 Results

Retrieval Types	Retrieval Headers	Error Rounds	Average % Tests Pass BO	EM	PA	PL	TO	ALL
no	no	no	0	0	0	0	0	0
		yes	0	0	0	0	0	0
	yes	no	1	0	0	0	0	0
		yes	0	0	0	0	0	0
yes	no	no	26	0	7	10	15	12
		yes	31	1	40	53	47	34
	yes	no	28	79	29	68	35	48
		yes	41	95	93	81	71	76

Vector Retrieval	Error Rounds	Average % Tests Pass BO	EM	PA	PL	TO	ALL
yes	no	0	0	0	15	33	10
	yes	0	3	0	20	67	18

Exhaustive Retrieval	Error Rounds	Average % Tests Pass BO	EM	PA	PL	TO	ALL
yes	no	9	90	61	43	69	54
	yes	43	100	91	81	80	79

Fig. 8. Hazel GPT-4: Results for guided completion (20 trials per, temperature 0.6)

Fig. 8 shows the results of our evaluation with GPT-4. We see a clear trend of more semantic information yielding better results on the held-out tests. The no-context baseline (no/no/no) alone does not suffice to yield meaningful generations, often returning syntactically incorrect code as the model hallucinates data types and syntax (such as OCaml-style records) which do not exist in Hazel, even despite the Hazel Crash Course. Including only function headers and error rounds also does not improve matters, with failure modes discussed below.

Including type definitions seem absolutely necessary to allow the model to scaffold the update function (bottom half). Without this scaffolding, relevant function headers alone did not prove sufficient informative to improve results. However, in combination with types, function headers have a large multiplicative effect, increasing test performance threefold. Figure 9 provides concrete examples of some of this interaction.

Similarly error rounds on their own are ineffective on code consisting purely of hallucinated types and functions. But given the scaffolding effect of relevant static information, they act multiplicatively, increasing performance by a factor of 4 for types without headers, and a factor of 1.5 for both types and headers. Error rounds were particularly effective at bringing almost-correct completions to fully correct ones, as in figure 10.

One phenomenon of note was that sometimes even bad error messages proved effective; the error in Figure 11 is a misattribution of the source of the parse error, but knowing there was a parse error proved sufficient for the model to correct it, perhaps due to the additional context provided in the Hazel crash course.

The combination of types and headers performed well against the Vector Retrieval baseline, though it should be noted that this was disproportionately due to a single confounding chunk which was retrieved for each example, even though it is only relevant to the Todo application (see Fig. 12). We believe that, due to the fact that this chunk coincidentally includes the word symbol

(a) No relevant type definitions or values

```
let update: (Model, Action) -> Model =
  fun (model, action) ->
    case action
    | Paint(emoji) => {model with tool = Some(emoji)}
    | Clear => {model with canvas = blank_canvas; tool = None }
    | Undo => {model with canvas = do_undo(model.undo_stack) }
    end
 in
```

(c) Relevant values but no types

```
let update: (Model, Action) -> Model =
  fun model, action ->
    case action
    | Paint(row, col, emoji) => updateGrid(model.grid, row, col, emoji)
    | FillRow(row, emoji) => fillRowInGrid(model.grid, row, emoji)
    | Clear => clearGrid(model.grid)
    end in
```

(b) Type definitions but no values

```
let update: (Model, Action) -> Model =
  fun model, action ->
    let (grid, selected_emoji, emoji_palette), (action) = model, action in
    case action
    | SelectEmoji(emoji) => (grid, emoji, emoji_palette)
    | StampEmoji((row, col)) =>
    let grid = Grid.stamp(row, col, selected_emoji, grid) in
    (grid, selected_emoji, emoji_palette)
    | ClearCell((row, col)) =>
    let grid = Grid.clear(row, col, grid) in
    (grid, selected_emoji, emoji_palette)
    | ClearGrid => (Grid.empty_grid(), selected_emoji, emoji_palette)
```

(d) Both values and types (correct implementation)

```
let update: (Model, Action) -> Model =
  fun model, action ->
    let (grid, current_emoji, emoji_list) = model in
    case action
    | SelectEmoji(emoji) => (grid, emoji, emoji_list)
    | StampEmoji((row, col)) =>
    (updateGrid(grid, row, col, current_emoji), current_emoji, emoji_list)
    | ClearCell((row, col)) => (updateGrid(grid,row, col, ""), current_emoji, emoji_list)
    | ClearGrid => (clearGrid(grid), current_emoji, emoji_list)
    | FillRow(row) => (fillRowInGrid(grid, row, current_emoji), current_emoji, emoji_list)
    end in
```

Fig. 9. Some sample completions for various configurations. In (a), without any supporting context, we see reasonable but incorrect guesses at both the Action constructors and the model type; furthermore, the model uses record syntax which does not exist in Hazel. (b) uses the provided types correctly but hallucinates helper names. In (c) we see uses of appropriate helpers, but (mostly) incorrect guesses for Action constructors. (d) exploits the provided context to produce a fully correct solution

(a) Initial response

```
let update: (Model, Action) -> Model =
  fun model, action ->
    case action
      | SelectEmoji(emoji) =>
      let (_, _, emoji_palette) = model in
      (updateGrid(model, -1, -1, emoji), emoji, emoji_palette)
      | StampEmoji((row, col)) =>
      let (grid, selected_emoji, emoji_palette) = model in
      (updateGrid(grid, row, col, selected_emoji), selected_emoji, emoji_palette)
      | ClearCell((row, col)) =>
      let (grid, selected_emoji, emoji_palette) = model in
      (updateGrid(grid, row, col, ""), selected_emoji, emoji_palette)
      | ClearGrid =>
      let (_, selected_emoji, emoji_palette) = model in
      (clearGrid(grid), selected_emoji, emoji_palette)
      | FillRow(row) =>
      let (grid, selected_emoji, emoji_palette) = model in
      (fillRowInGrid(grid, row, selected_emoji), selected_emoji, emoji_palette)
    end in
```

(b) Reply after error round

```
let update: (Model, Action) -> Model =
  fun model, action ->
    case action
      | SelectEmoji(emoji) =>
      let grid, _, emoji_palette = model in
      (updateGrid(grid, -1, -1, emoji), emoji, emoji_palette)
      | StampEmoji((row, col)) =>
      let grid, selected_emoji, emoji_palette = model in
      (updateGrid(grid, row, col, selected_emoji), selected_emoji, emoji_palette)
      | ClearCell((row, col)) =>
      let grid, selected_emoji, emoji_palette = model in
      (updateGrid(grid, row, col, ""), selected_emoji, emoji_palette)
      | ClearGrid =>
      let grid, selected_emoji, emoji_palette = model in
      (clearGrid(grid), selected_emoji, emoji_palette)
      | FillRow(row) =>
      let grid, selected_emoji, emoji_palette = model in
      (fillRowInGrid(grid, row, selected_emoji), selected_emoji, emoji_palette)
    end in
```

Fig. 10. Error rounds were generally very effective at correcting an almost-correct program. Here, the error round reply included a type inconsistency error on 'model' and an unbound variable error in 'grid', both of which were corrected in the model's reply

(a) Initial response

```
let update: (Model, Action) -> Model =
  fun model, action ->
    match (model, action) with
    (grid, current_emoji, available_emojis), SelectEmoji(emoji) ->
    (grid, emoji, available_emojis)
    | (grid, current_emoji, available_emojis), StampEmoji((row, col)) ->
      (updateGrid(grid, row, col, current_emoji), current_emoji, available_emojis)
      | (grid, current_emoji, available_emojis), ClearCell((row, col)) ->
        (updateGrid(grid, row, col, ""), current_emoji, available_emojis)
        | (grid, current_emoji, available_emojis), ClearGrid ->
          (clearGrid(grid), current_emoji, available_emojis)
          | (grid, current_emoji, available_emojis), FillRow(row) ->
            (fillRowInGrid(grid, row, current_emoji), current_emoji, available_emojis)) in
```

(b) Reply after error round

```
let update: (Model, Action) -> Model =
  fun model, action ->
    case (model, action)
      | (grid, current_emoji, available_emojis), SelectEmoji(emoji) =>
      (grid, emoji, available_emojis)
      | (grid, current_emoji, available_emojis), StampEmoji((row, col)) =>
      (updateGrid(grid, row, col, current_emoji), current_emoji, available_emojis)
      | (grid, current_emoji, available_emojis), ClearCell((row, col)) =>
      (updateGrid(grid, row, col, ""), current_emoji, available_emojis)
      | (grid, current_emoji, available_emojis), ClearGrid =>
      (clearGrid(grid), current_emoji, available_emojis)
      | (grid, current_emoji, available_emojis), FillRow(row) =>
      (fillRowInGrid(grid, row, current_emoji), current_emoji, available_emojis)
    end in
```

Fig. 11. Here, a parse error (match/with used instead of case) was corrected, even though the Hazel error message in this case is not evocative: "The parser has detected the following unmatched delimiters:. The presence of a '=>' in the list likely indicates that a '->' was mistakenly used in a case expression: =>, =>, =>, =>"

`Model` twice alongside `Action`, it is deemed relevant to each function header. The inclusion of this snippet results in the language model attempting to implement Todo, or some hybrid application.

We debated refining the chunking strategy to remove this confounder, but found it easy to inadvertently create similar scenarios; ultimately, we feel it a representative artefact of process which is fundamentally non-scope-aware. See Sec. 4 for further discussion.

```
# SNIPPET 1 #
, ts2) in

let Model.init: Model = ("", []) in

type Action =
  + AddTodo
  + RemoveTodo(Int)
  + ToggleTodo(Int)
  + UpdateBuffer(String) in
```

Fig. 12. A confounding snippet commonly retrieved by vector retrieval

See Figure. 13 for additional examples of more exotic completions.

(a) Types but no values

```
let update: (Model, Action) -> Model =
  fun model, action ->
    case action
     | SelectEmoji(emoji) =>
     let (grid, _, palette) = model in
     (grid, emoji, palette)
     | StampEmoji((row, col)) =>
     let (grid, emoji, palette) = model in
     let newGrid = if row < List.length(grid) then
       let rowContents = List.nth(grid, row) in
       if col < List.length(rowContents)
       then List.update(rowContents, col, emoji)
       else rowContents
     else grid
     in
     (newGrid, emoji, palette)
```

(b) Types but no values

```
let update: (Model, Action) -> Model =
  fun (model, action) ->
    case action
     | SelectEmoji(emoji) => (model.1, emoji, model.3)
     | StampEmoji((row, col)) =>
     let stampedGrid = List.map(
       fun idx, row ->
         if idx == row then model.2 else row, model.1) in
     (stampedGrid, model.2, model.3)
     | ClearCell((row, col)) => ??
     | ClearGrid => ((([]), "", [])
     | FillRow(row) => ??
     end in
```

(c) Types and values (incorrect)

```
let update: (Model, Action) -> Model =
  fun (model, action) ->
    case action
     | ClearCell((row, col)) =>
      let (grid, emoji, palette) = model in
      let updated_grid = updateGrid(grid, row, col, "😀") in
      (updated_grid, emoji, palette)
     | ClearGrid =>
      let (_, emoji, palette) = model in
      let cleared_grid = clearGrid(grid) in
      (cleared_grid, emoji, palette)
     | FillRow(row) =>
      let (grid, emoji, palette) = model in
      let filled_grid = fillRowInGrid(grid, row, emoji) in
      (filled_grid, emoji, palette)
```

Fig. 13. Some slightly more exotic completions. (a) is an excerpt of a very long solution which almost successfully managed to re-implement helper code. In (b) at the bottom we see an example of the model occasionally leaving holes in completions, likely due to their presence in our few-shot example sketches. In (c), the model somewhat mysteriously suggests substituting a different emoji in lieu of the empty string.

2.10 Hazel StarCoder2-15B Results

Completion Results (Hazel, StarCoder, Trials=20, Temp=0.5)

Retrieval		Average % Tests Pass					
Types	Headers	BO	EM	PA	PL	TO	ALL
no	no	0	0	0	0	3	1
yes	no	30	3	5	20	15	15
	yes	18	44	23	24	5	23

Fig. 14. Hazel StarCoder2: Results for guided completion (20 trials per, temperature 0.5)

To assess the effectiveness of static retrieval with smaller completion models, we conducted tests using StarCoder2-15B. To accommodate the constraints of a completion model, the retrieved types and headers were appended before the target header in their surface syntax. The average percentage of tests passed, shown in the rightmost column of Fig 14, exhibits a consistent trend with

the GPT-4 results. In the absence of any type or header information, StarCoder2 performed poorly. The addition of type information drastically improves performance, increasing the percentage of correct solutions from 1% to 15%. Furthermore, incorporating headers leads to an additional 8% increase in average performance.

However, two examples, BO and TO, experienced degraded performance after the inclusion of headers. Upon close examination of the headers and the output programs, we discovered that the LLM had picked up several irrelevant headers from the retrieved results and used them during the generation process. These irrelevant headers diminish the quality of the generated programs. We hypothesize that smaller completion models, such as StarCoder2, are more sensitive to code that appears near the end of the context window, making them more susceptible to the influence of irrelevant information.

3 STATIC RETRIEVAL IN TYPESCRIPT

To ensure that the above results are not an artifact of using a low-resource language, we also experimented with static retrieval in TypeScript.

3.1 TypeScript Methodology

Our methodology closely mirrors the Hazel methodology. As TypeScript is a high resource language, we do not require a syntax crash course as we do for Hazel. TypeScript lacks the explicit support for typed holes and thus a convenient way to extract semantic information. We emulated typed holes using a previously established approach [11] using generic functions (Fig. 15).

```
declare function _<T>(): T;
import { Model, Action } from "./prelude";

// Update the EmojiPaint app model based on an action
const update: (model: Model, action: Action) => Model =

  function _<(model: Model, action: Action) => Model>(): (model: Model, action: Action) => Model
  _()
```

Fig. 15. Hovering over simulated program hole in TypeScript

Static information retrieval is done using the TypeScript Language Server. This language server is designed to support human-facing affordances, so in particular we use *Go to type definition* and *Hover* commands in tandem to automatically recursively extract expected types. Our TypeScript implementation should be considered a proof-of-concept; owing to the amount of engineering required to complete relevant value retrieval (there is no suitable direct method to retrieve the typing context at a given location), we have elected to assemble these manually for examples, emulating the same methodology as the Hazel Language Server employs. We used the TypeScript compiler to collate static errors for correction rounds.

Adapting MVUBench to Hazel was done with the substantial aid of Claude [3], an LLM chat agent. Transliterated code was manually adjusted to establish basic conformance to TypeScript idioms, for example adding elements to array at the back, versus at the front is standard for lists in functional languages like Hazel. This suggests that MVUBench can be ported with relative ease to other similar languages.

3.2 TypeScript GPT-4 Results

In broad strokes TypeScript results are similar to the Hazel results. Even without the inclusion of syntax examples, we see, somewhat unsurprisingly, that a higher-resource language, well represented in the training set, receives better overall completions from the model. Some trials pass some tests even with no type information provided.

Completion Results (TypeScript, GPT-4, N=20, Temp=0.6)

Retrieval		Error	Average % Tests Pass					
Types	Headers	Rounds	BO	EM	PA	PL	TO	ALL
no	no	no	0	0	0	0	0	0
		yes	0	0	0	0	0	0
	yes	no	0	0	0	0	0	0
		yes	0	4	0	3	0	1
yes	no	no	46	26	43	85	66	53
		yes	69	39	46	93	66	63
	yes	no	47	95	83	64	75	73
		yes	58	99	85	86	75	80

Fig. 16. Typescript GPT-4: Results for guided completion (20 trials per, temperature 0.6)

With type definitions included, the TypeScript results are flatter than the Hazel results.

The ratio of tests passed with and without headers is 3 for Hazel, and 1.5 for TypeScript. From examining the generated completions, we see that the model, when not provided with relevant headers, is significantly more able to produce working facsimiles inline than it was in Hazel.

TypeScript performs well even without error rounds, and sees less benefit from the relevant function headers. In Hazel, the ratio of tests passing with or without error rounds was about 2, whereas for Typescript it's about 1.2. Again, this is likely due to the fact that the model is far more familiar with TypeScript syntax, and unlikely to make the kind of syntax errors which the error rounds were vital for correcting in the Hazel runs.

3.3 TypeScript StarCoder2-15B Results

Completion Results (TypeScript, StarCoder, Trials=20, Temp=0.5)

Retrieval		Average % Tests Pass					
Types	Headers	BO	EM	PA	PL	TO	ALL
no	no	3	0	0	0	2	1
	yes	0	0	0	0	0	0
yes	no	3	5	8	21	46	17
	yes	9	69	43	18	69	42

Fig. 17. Typescript StarCoder2: Results for guided completion (20 trials per, temperature 0.6)

TypeScript performs similarly to Hazel without headers, but 1.5 times better with them. The ratio of tests passed with and without headers for Hazel is 1.7 (decrease from its GPT-4 results), but 2.5 for TypeScript (increase from its GPT-4 results). This is mainly due to the difference between the language models—StarCoder2 works to write a whole completion to a program sketch without knowing the language it is currently working with. By providing more in-context examples in the form of function headers, the LLM is primed to output more TypeScript-like code.

4 THREATS TO VALIDITY

4.1 Internal

The improvement seen from the inclusion of relevant function headers is highly contingent on the fact that many relevant functions have already been implemented. While we do suggest that this approximates a common case in real programming processes (as previously described) in which naive contextualization strategies fail, validation of this claim would require larger-scale study, using more scale programs which were more neutrally selected.

4.2 External

MVUBench is not meant to be representative of all coding tasks, but rather to present a challenge to contemporary techniques and help evaluate approaches to semantic contextualization (e.g. vector retrieval, which we evaluate in Sec. 2.8.5).

Our TypeScript MVUBench is a very close translation of the Hazel code; although the MVU paradigm is in use in the TypeScript world [6], there is a question as to the applicability of these methods to the broader range of idiomatic TypeScript code in wide use.

The appropriateness of our baselines is debatable. This is an idealized setup in which the baseline case is to provide no information beyond the function header; it is unsurprising that adding related context drastically improves the result. In many real cases the cursor window would contain a large amount of relevant code. We have chosen here to portray the situation, also true in many real cases, where the window does not, but it remains to validate the relative rate of occurrence of these scenarios in the wild.

Our RAG baseline is relatively simplistic, and practical implementations are increasingly integrating more sophisticated methods of chunking which may more closely approximate static retrieval. The fact that we compensated for the small size of our examples by creating a conjoined codebase to create our embedding vector database may not be adequately representative of a large-scale codebase.

5 RELATED WORK

The introduction covered the broader literature on LLMs for code, so we focus here specifically on other methods for semantic contextualization of LLM-based code generation systems.

Error correction using instruction-tuned models is a widespread technique and not itself a novel contribution of this paper, e.g. much work on program repair with LLMs is fundamentally rooted in this idea [29, 55]. The contribution of this paper is the observation that error looping alone is not sufficient in a context-poor setting, and that error looping together with contextualization is the most effective technique, particularly for a low-resource language like Hazel.

The observation that LLMs perform poorly when they lack repository-level context has been made in a number of recent papers, which have approached it in a variety of ways. We discussed the benchmarks used in these papers in Sec 1.1.2 so we do not repeat the discussion here.

RepoCoder [71] uses vector retrieval to contextualize Python code. Our experiments demonstrate that vector retrieval is highly sensitive to semantic confounds that are easily handled by static retrieval.

The Repo-Level Prompt Generator [59] uses machine learning to decide how to construct a useful prompt, drawing information from coarse-grained static information like imports and parent-child relationships between classes. Even this level of contextualization showed substantial promise relative to baselines.

Pei et al. [51] tackle the difficult problem of contextualizing Python function calls using a static analyzer from Python, which can provide function implementations and function usage examples. Again, even this level of contextualization is quite helpful. Our focus here was on gradually typed languages, and we did not include function implementations or usage examples, suggesting potential future work.

CodeTrek [49] also uses program analyses, generated from task- and error-relevant queries and expressed in CodeQL, to generate semantic contextualization for program repair tasks in Python. This too was quite effective and suggests that richer static analyses might be of interest in particular settings. For hole filling, however, it may be that lightweight static methods, like type checking, are more efficient. However, we look forward to future direct comparisons of these methods.

Li et al. [34] also identify the semantic contextualization problem and propose IDECoder, a system that uses the static information tracked by an IDE or language server to contextualize LLM code completion. This is an outline of early experiments in this direction which have not yet been fully evaluated, but we agree with the thrusts of the argument made here and look forward to additional experimentation in this direction by the community.

CoCoMIC [17] is a framework that learns in-file and cross-file context jointly atop an LLM. This differs from our approach in that it deploys a learning step to decide which cross-file context to attend to, which may be subject to similar issues as vector retrieval approaches when given confounding contexts. However, this represents a fascinating future direction when combined with static retrieval, which can often lead to too much information to include in a token window.

Dehallucinator [19] is an approach that performs semantic lookup after an initial generation phase to lookup potentially relevant definitions that were invalid, e.g. not in scope. This is a more sophisticated form of error correction and could be combined with the proactive contextualization that we propose. Other approaches, e.g. prompting the model to implement

Agrawal et al. [2] and Wei et al. [65] propose an approach that modifies token sampling by leveraging the semantic code completion systems already available in modern IDEs, which implicitly provide some context. One issue with this approach is that they can only sample from tokens that the model has assigned some baseline level of probability, but without semantic context this may not be the case. There is likely substantial room for future work in combining static retrieval with this sort of structure-guided sampling, and perhaps with providing more fine-grained retrieval at each token rather than once at the onset of code completion.

Zan et al. [70] retrieves potentially relevant code from API documentation, then further proposes a continuous training approach to incorporate this information into the model weights. In contrast, our approach is focused on black-box pre-trained models. In the future, incorporating commonly used private APIs into a continuous training loop would improve token efficiency, leaving more room in the context for truly novel definitions.

We note that there is also a vast literature on non-LLM-based code generation systems, some of which also use types to restrict the search space [47]. Our approach helps bring these two worlds together, e.g. by using a form of typed term enumeration to generate the relevant headers. We hope that our results will lead to more interactions between the programming languages community and the AI community.

6 DISCUSSION AND CONCLUSION

An AI model, no matter how powerful, cannot divine a human's intent without access to necessary context. Fortunately, type theory is rooted fundamentally in a formal notion of *typing contexts*. In particular, contextual modal type theory (CMTT) [42] and gradual type theory [60] provide a formal foundation for program sketching with holes, where expression holes corresponding to metavariables with a corresponding type and typing context and type holes correspond to unknown types. The Hazel programming environment, with its roots in gradual CMTT [44] as a foundational theory of holes and its support for total syntax and type error recovery with holes [41, 73], therefore presents an ideal environment for *statically contextualizing large language models with typed holes*. Our results demonstrate that this form of contextualization, together with some in-context prompting about the specific choices made in Hazel, a low-resource language, can take a model incapable of even basic MVU tasks up to, or nearly up to, the performance observed in a fully contextualized setting for a high resource language like TypeScript. These ideas have been realized in a functional programming assistant, the Hazel Assistant, that supports both fast-pass and slow-pass code completion based on types.

These ideas can also be ported directly to other languages, like TypeScript, albeit with some difficulty due to limitations of standard language servers.

Our comparisons to vector retrieval, a common language-agnostic approach, suggests that language-aware programming assistants may significantly outperform language-agnostic retrieval systems in the short- and medium-term, and perhaps far into the future.

Additional forms of semantic contextualization, e.g. using dynamic test results passed backwards to holes [40], the results of various static and dynamic analyses, and the result of library searches to find helpers that may not yet be imported are interesting avenues for future work.

REFERENCES

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[2] Lakshya A Agrawal, Aditya Kanade, Navin Goyal, Shuvendu K. Lahiri, and Sriram K. Rajamani. 2023. Guiding Language Models of Code with Global Context using Monitors. arXiv:2306.10763 [cs.CL]

[3] Anthropic. 2024. Introducing the next generation of Claude. https://www.anthropic.com/news/claude-3-family

[4] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *CoRR* abs/2108.07732 (2021). arXiv:2108.07732 https://arxiv.org/abs/2108.07732

[5] Thomas Bandt. 2020. Pragmatic MVU With React And TypeScript. (2020). https://thomasbandt.com/model-view-update-with-react-and-typescript.

[6] Thomas Bandt. 2020. Pragmatic MVU With React And TypeScript. https://thomasbandt.com/model-view-update-with-react-and-typescript

[7] Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 85–111. https://doi.org/10.1145/3586030

[8] Djonathan Barros, Sven Peldszus, Wesley K. G. Assunção, and Thorsten Berger. 2022. Editing support for software languages: implementation practices in language server protocols. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems* (Montreal, Quebec, Canada) *(MODELS '22)*. Association for Computing Machinery, New York, NY, USA, 232–243. https://doi.org/10.1145/3550355.3552452

[9] Andrew Blinn, David Moon, Eric Griffis, and Cyrus Omar. 2022. An Integrative Human-Centered Architecture for Interactive Programming Assistants. In *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 1–5. https://doi.org/10.1109/VL/HCC53370.2022.9833110

[10] Frédéric Bour, Thomas Refis, and Gabriel Scherer. 2018. Merlin: a language server for OCaml (experience report). *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 1–15.

[11] Giulio Canti. 2019. Type holes in TypeScript. https://dev.to/gcanti/type-holes-in-typescript-2lck

[12] Federico Cassano, John Gouwar, Francesca Lucchetti, Claire Schlesinger, Anders Freeman, Carolyn Jane Anderson, Molly Q Feldman, Michael Greenberg, Abhinav Jangda, and Arjun Guha. 2024. Knowledge Transfer from High-Resource to Low-Resource Programming Languages for Code LLMs. arXiv:2308.09895 [cs.PL]

[13] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[14] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[15] Czaplicki, Evan. [n. d.]. Elm Architecture. ([n. d.]). https://guide.elm-lang.org/architecture/.

[16] Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2023. CrossCodeEval: A Diverse and Multilingual Benchmark for Cross-File Code Completion. arXiv:2310.11248 [cs.LG]

[17] Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2023. CoCoMIC: Code Completion By Jointly Modeling In-file and Cross-file Context. arXiv:2212.10007 [cs.CL]

[18] Yiran Ding, Li Lyna Zhang, Chengruidong Zhang, Yuanyuan Xu, Ning Shang, Jiahang Xu, Fan Yang, and Mao Yang. 2024. LongRoPE: Extending LLM Context Window Beyond 2 Million Tokens. arXiv:2402.13753 [cs.CL]

[19] Aryaz Eghbali and Michael Pradel. 2024. De-Hallucinator: Iterative Grounding for LLM-Based Code Completion. arXiv:2401.01701 [cs.SE]

[20] Nat Friedman. 2021. Introducing GitHub Copilot: your AI pair programmer. https://github.blog/2021-06-29-introducing-github-copilot-ai-pair-programmer/

[21] Shuzheng Gao, Xin-Cheng Wen, Cuiyun Gao, Wenxuan Wang, Hongyu Zhang, and Michael R. Lyu. 2023. What Makes Good In-Context Demonstrations for Code Intelligence Tasks with LLMs?. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. https://doi.org/10.1109/ase56229.2023.00109

[22] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Qianyu Guo, Meng Wang, and Haofen Wang. 2023. Retrieval-Augmented Generation for Large Language Models: A Survey. *CoRR* abs/2312.10997 (2023). https://doi.org/10.48550/ARXIV.2312.10997 arXiv:2312.10997

[23] Elena L. Glassman, Jeremy Scott, Rishabh Singh, Philip J. Guo, and Robert C. Miller. 2015. OverCode: Visualizing Variation in Student Solutions to Programming Problems at Scale. *ACM Trans. Comput.-Hum. Interact.* 22, 2, Article 7 (mar 2015), 35 pages. https://doi.org/10.1145/2699751

[24] Robert Harper. 2016. *Practical Foundations for Programming Languages (2nd. Ed.)*. Cambridge University Press. https://www.cs.cmu.edu/%7Erwh/pfpl/index.html

[25] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Commun. ACM* 59, 5 (2016), 122–131.

[26] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. 2022. Training Compute-Optimal Large Language Models. arXiv:2203.15556 [cs.CL]

[27] Alon Jacovi, Avi Caciularu, Omer Goldman, and Yoav Goldberg. 2023. Stop Uploading Test Data in Plain Text: Practical Strategies for Mitigating Data Contamination by Evaluation Benchmarks. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 5075–5084. https://doi.org/10.18653/v1/2023.emnlp-main.308

[28] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. *CoRR* abs/2403.07974 (2024). https://doi.org/10.48550/ARXIV.2403.07974 arXiv:2403.07974

[29] Harshit Joshi, José Cambronero Sanchez, Sumit Gulwani, Vu Le, Ivan Radiček, and Gust Verbruggen. 2023. Repair is nearly generation: multilingual program repair with LLMs. In *Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence and Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence and Thirteenth Symposium on Educational Advances in Artificial Intelligence (AAAI'23/IAAI'23/EAAI'23)*. AAAI Press, Article 573,

10 pages. https://doi.org/10.1609/aaai.v37i4.25642

[30] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, Corina S. Pasareanu and Darko Marinov (Eds.). ACM, 437–440. https://doi.org/10.1145/2610384.2628055

[31] Mosh Levy, Alon Jacoby, and Yoav Goldberg. 2024. Same Task, More Tokens: the Impact of Input Length on the Reasoning Performance of Large Language Models. *arXiv preprint arXiv:2402.14848* (2024).

[32] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2021. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. arXiv:2005.11401 [cs.CL]

[33] Jiaqi Li, Mengmeng Wang, Zilong Zheng, and Muhan Zhang. 2023. LooGLE: Can Long-Context Language Models Understand Long Contexts? *arXiv preprint arXiv:2311.04939* (2023).

[34] Yichen Li, Yun Peng, Yintong Huo, and Michael R. Lyu. 2024. Enhancing LLM-Based Coding Tools through Native Integration of IDE-Derived Static Context. arXiv:2402.03630 [cs.SE]

[35] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. arXiv:2305.01210 [cs.SE]

[36] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics* 12 (2024), 157–173.

[37] Tianyang Liu, Canwen Xu, and Julian J. McAuley. 2023. RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems. *CoRR* abs/2306.03091 (2023). https://doi.org/10.48550/ARXIV.2306.03091 arXiv:2306.03091

[38] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2024. StarCoder 2 and The Stack v2: The Next Generation. arXiv:2402.19173 [cs.SE]

[39] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung won Hwang, and Alexey Svyatkovskiy. 2022. ReACC: A Retrieval-Augmented Code Completion Framework. arXiv:2203.07722 [cs.SE]

[40] Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. 2020. Program Sketching with Live Bidirectional Evaluation. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020), 1–29.

[41] D. Moon, A. Blinn, and C. Omar. 2023. Gradual Structure Editing with Obligations. In *2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE Computer Society, Los Alamitos, CA, USA, 71–81. https://doi.org/10.1109/VL-HCC57772.2023.00016

[42] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual modal type theory. *ACM Transactions on Computational Logic (TOCL)* 9, 3 (2008), 1–49.

[43] Arvind Neelakantan, Tao Xu, Raul Puri, Alec Radford, Jesse Michael Han, Jerry Tworek, Qiming Yuan, Nikolas Tezak, Jong Wook Kim, Chris Hallacy, Johannes Heidecke, Pranav Shyam, Boris Power, Tyna Eloundou Nekoul, Girish Sastry, Gretchen Krueger, David Schnurr, Felipe Petroski Such, Kenny Hsu, Madeleine Thompson, Tabarak Khan, Toki Sherbakov, Joanne Jang, Peter Welinder, and Lilian Weng. 2022. Text and Code Embeddings by Contrastive Pre-Training. arXiv:2201.10005 [cs.CL]

[44] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live Functional Programming with Typed Holes. *Proceedings of the ACM on Programming Languages (PACMPL), Issue POPL* (2019).

[45] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. 2017. Hazelnut: A Bidirectionally Typed Structure Editor Calculus. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*.

[46] Cyrus Omar, Ian Voysey, Michael Hilton, Joshua Sunshine, Claire Le Goues, Jonathan Aldrich, and Matthew A. Hammer. 2017. Toward Semantic Foundations for Program Editors. In *Summit on Advances in Programming Languages (SNAPL)*.

[47] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-Example-Directed Program Synthesis. In *Conference on Programming Language Design and Implementation (PLDI)*.

[48] Shuyin Ouyang, Jie M. Zhang, Mark Harman, and Meng Wang. 2023. LLM is Like a Box of Chocolates: the Nondeterminism of ChatGPT in Code Generation. arXiv:2308.02828 [cs.SE]

[49] Pardis Pashakhanloo, Aaditya Naik, Yuepeng Wang, Hanjun Dai, Petros Maniatis, and Mayur Naik. 2021. Codetrek: Flexible modeling of code using an extensible relational representation. In *International Conference on Learning Representations*.

[50] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 754–768.

[51] Hengzhi Pei, Jinman Zhao, Leonard Lausen, Sheng Zha, and George Karypis. 2023. Better context makes better code language models: a case study on function call argument completion. In *Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence and Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence and Thirteenth Symposium on Educational Advances in Artificial Intelligence (AAAI'23/IAAI'23/EAAI'23)*. AAAI Press, Article 584, 9 pages. https://doi.org/10.1609/aaai.v37i4.25653

[52] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. 2023. The Impact of AI on Developer Productivity: Evidence from GitHub Copilot. *CoRR* abs/2302.06590 (2023). https://doi.org/10.48550/ARXIV.2302.06590 arXiv:2302.06590

[53] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 2023. Do users write more insecure code with AI assistants?. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2785–2799.

[54] Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press.

[55] Julian Aron Prenner, Hlib Babii, and Romain Robbes. 2022. Can OpenAI's codex fix bugs? an evaluation on QuixBugs. In *Proceedings of the Third International Workshop on Automated Program Repair* (Pittsburgh, Pennsylvania) *(APR '22)*. Association for Computing Machinery, New York, NY, USA, 69–75. https://doi.org/10.1145/3524459.3527351

[56] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[57] Oscar Sainz, Jon Ander Campos, Iker García-Ferrero, Julen Etxaniz, Oier Lopez de Lacalle, and Eneko Agirre. 2023. Nlp evaluation in trouble: On the need to measure llm data contamination for each benchmark. *arXiv preprint arXiv:2310.18018* (2023).

[58] Anton Semenkin, Yaroslav Sokolov, and Evgeniia Vu. 2024. Context Composing for Full Line Code Completion. *arXiv preprint arXiv:2402.09230* (2024).

[59] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023. Repository-level prompt generation for large language models of code. In *Proceedings of the 40th International Conference on Machine Learning* (<conf-loc>, <city>Honolulu</city>, <state>Hawaii</state>, <country>USA</country>, </conf-loc>) *(ICML'23)*. JMLR.org, Article 1314, 23 pages.

[60] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA (LIPIcs, Vol. 32)*, Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 274–293. https://doi.org/10.4230/LIPICS.SNAPL.2015.274

[61] Parth Thakkar. 2022. Copilot Explorer. https://thakkarparth007.github.io/copilot-explorer/posts/copilot-internals.html

[62] Priyan Vaithilingam, Elena L. Glassman, Peter Groenwegen, Sumit Gulwani, Austin Z. Henley, Rohan Malpani, David Pugh, Arjun Radhakrishna, Gustavo Soares, Joey Wang, and Aaron Yim. 2023. Towards More Effective AI-Assisted Programming: A Systematic Design Exploration to Improve Visual Studio IntelliCode's User Experience. In *45th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, SEIP@ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 185–195. https://doi.org/10.1109/ICSE-SEIP58684.2023.00022

[63] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *CHI '22: CHI Conference on Human Factors in Computing Systems, New Orleans, LA, USA, 29 April 2022 - 5 May 2022, Extended Abstracts*, Simone D. J. Barbosa, Cliff Lampe, Caroline Appert, and David A. Shamma (Eds.). ACM, 332:1–332:7. https://doi.org/10.1145/3491101.3519665

[64] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI EA '22)*. Association for Computing Machinery, New York, NY, USA, Article 332, 7 pages. https://doi.org/10.1145/3491101.3519665

[65] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated Program Repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (<conf-loc>, <city>San Francisco</city>, <state>CA</state>, <country>USA</country>, </conf-loc>) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 172–184. https://doi.org/10.1145/3611643.3616271

[66] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *MAPS@PLDI 2022: 6th ACM SIGPLAN International Symposium on Machine Programming,*

San Diego, CA, USA, 13 June 2022, Swarat Chaudhuri and Charles Sutton (Eds.). ACM, 1–10. https://doi.org/10.1145/3520312.3534862

[67] Rongwu Xu, Zehan Qi, Cunxiang Wang, Hongru Wang, Yue Zhang, and Wei Xu. 2024. Knowledge Conflicts for LLMs: A Survey. *arXiv preprint arXiv:2403.08319* (2024).

[68] Ziwei Xu, Sanjay Jain, and Mohan Kankanhalli. 2024. Hallucination is inevitable: An innate limitation of large language models. *arXiv preprint arXiv:2401.11817* (2024).

[69] Yongwei Yuan, Scott Guest, Eric Griffis, Hannah Potter, David Moon, and Cyrus Omar. 2023. Live Pattern Matching with Typed Holes. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 609–635. https://doi.org/10.1145/3586048

[70] Daoguang Zan, Bei Chen, Yongshun Gong, Junzhi Cao, Fengji Zhang, Bingchao Wu, Bei Guan, Yilong Yin, and Yongji Wang. 2023. Private-library-oriented code generation with large language models. *arXiv preprint arXiv:2307.15370* (2023).

[71] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 2471–2484. https://doi.org/10.18653/v1/2023.emnlp-main.151

[72] Yue Zhang, Yafu Li, Leyang Cui, Deng Cai, Lemao Liu, Tingchen Fu, Xinting Huang, Enbo Zhao, Yu Zhang, Yulong Chen, et al. 2023. Siren's song in the AI ocean: a survey on hallucination in large language models. *arXiv preprint arXiv:2309.01219* (2023).

[73] Eric Zhao, Raef Maroof, Anand Dukkipati, Andrew Blinn, Zhiyi Pan, and Cyrus Omar. 2024. Total Type Error Localization and Recovery with Holes. *Proc. ACM Program. Lang.* 8, POPL (2024), 2041–2068. https://doi.org/10.1145/3632910

[74] Albert Ziegler, Eirini Kalliamvakou, X. Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2024. Measuring GitHub Copilot's Impact on Productivity. *Commun. ACM* 67, 3 (2024), 54–63. https://doi.org/10.1145/3633453