An Integrative Human-Centered Architecture for Interactive Programming Assistants

Andrew Blinn
University of Michigan
Ann Arbor, MI, USA
blinnand@umich.edu

David Moon
University of Michigan
Ann Arbor, MI, USA
dmoo@umich.edu

Eric Griffis
University of Michigan
Ann Arbor, MI, USA
egriffis@umich.edu

Cyrus Omar
University of Michigan
Ann Arbor, MI, USA
comar@umich.edu

***Abstract*—Programming has become a collaboration between human programmers, who drive intent, and interactive assistants that suggest contextually relevant editor actions. There has been considerable work on suggestion synthesis strategies—from semantic autocomplete to modern program synthesis, repair, and machine learning research. This diversity of contextually viable strategies creates a need for an integrative, human-centered perspective on the problem of programming assistant design that (1) confronts the problem of integrating a variety of synthesis strategies, fed by shared semantic analyses capable of operating on program sketches, and (2) centers the needs of the human programmer: comprehending, comparing, ranking, and filtering suggestions generated by various synthesizers, and in some cases participating in a synthesizer's search by supplying additional expressions of intent. This paper contributes a conceptual architecture and API to guide programming assistant designers as they confront these integration and human-centered design challenges. We then instantiate this architecture with two prototype end-to-end assistant designs, both developed for the Hazel programming environment, that emphasize understudied design aspects, namely continuity, explainability, human-in-the-loop synthesis, and the integration of multiple analyses with multiple synthesis strategies.**

I. Introduction

A *programming assistant* is an editor service that analyzes the editor state (consisting primarily of a program sketch, perhaps with additional data such as history) to present edit action suggestions to a human user and help the user select an action consistent with their broader intent [1–3]. Programming assistants promise to improve programmer productivity by automating tedious programming tasks. They may also improve software quality by helping programmers avoid mistakes and, in some cases, guarantee that the suggestions satisfy programmer-specified correctness constraints. Moreover, they reduce knowledge gaps by surfacing structures and idioms that a programmer might not have otherwise discovered [4–6].

Given these benefits, it is unsurprising that simple assistants like code completion and "hotfix" systems are ubiquitous in modern programming environments, competing in frequency with manual editor actions like code insertion and deletion [7]. There has in turn been substantial research interest [8] in techniques that can synthesize better suggestions, including program synthesis using types [9], examples [10], program sketches [11], edit history [12], demonstrations [13], and logical constraints [14] to generate "hole completions" [15, 16]. Github Copilot [17] is one of several recent efforts focused on synthesizing long-form completions by using machine learning techniques that learn idioms from a large corpus of example programs [18–20].

While much of this research has focused on the underlying synthesis algorithms, there has recently been a renaissance of human-centered approaches that remind us that ultimately, the human programmer remains the driver of intent and the arbiter of correctness. Consequently, the interface through which the human communicates intent to the assistant and considers its suggestions must be designed with cognitive costs in mind. For example, many of these synthesis techniques are capable of substantial associative leaps, which carry with them concerns about explainability and the costs of validating correctness. In addition, code search spaces can become large, which can lead either to lengthy synthesis delays or too many suggestions. Concerns like these have led to work on interpretable synthesis [21] and interactive search space exploration [22].

We seek to organize this often-overwhelming diversity of efforts by developing an integrative architecture for programming assistant designers that confronts the problem of integrating a wide variety of synthesis techniques (and requisite program analyses) while centering the needs of the human user. Each component of this architecture is the subject of ongoing research, as is the overall design problem. We demonstrate that our architecture can serve to characterize and situate some existing assistant designs. We then describe our ongoing work on two end-to-end prototypes intended to emphasize understudied design criteria, namely continuity of service, integration of multiple shared analyses with multiple synthesis strategies, explainability, independent semantic ranking techniques, and interfaces for integrating the human into an incremental search.

Our intention is *not* to make empirical claims about the specific design choices made in these prototypes. Indeed, there is much work to be done before successors to these designs can claim to improve overall programmer productivity. Rather we present these prototypes as illustrations of the proposed integrative architecture, which we hope will help organize and provide vocabulary for the assistant design community, and to draw attention to understudied but important design criteria that we hope will draw more interest from the community.

978-1-6654-4214-5/22/$31.00 ©2022 IEEE

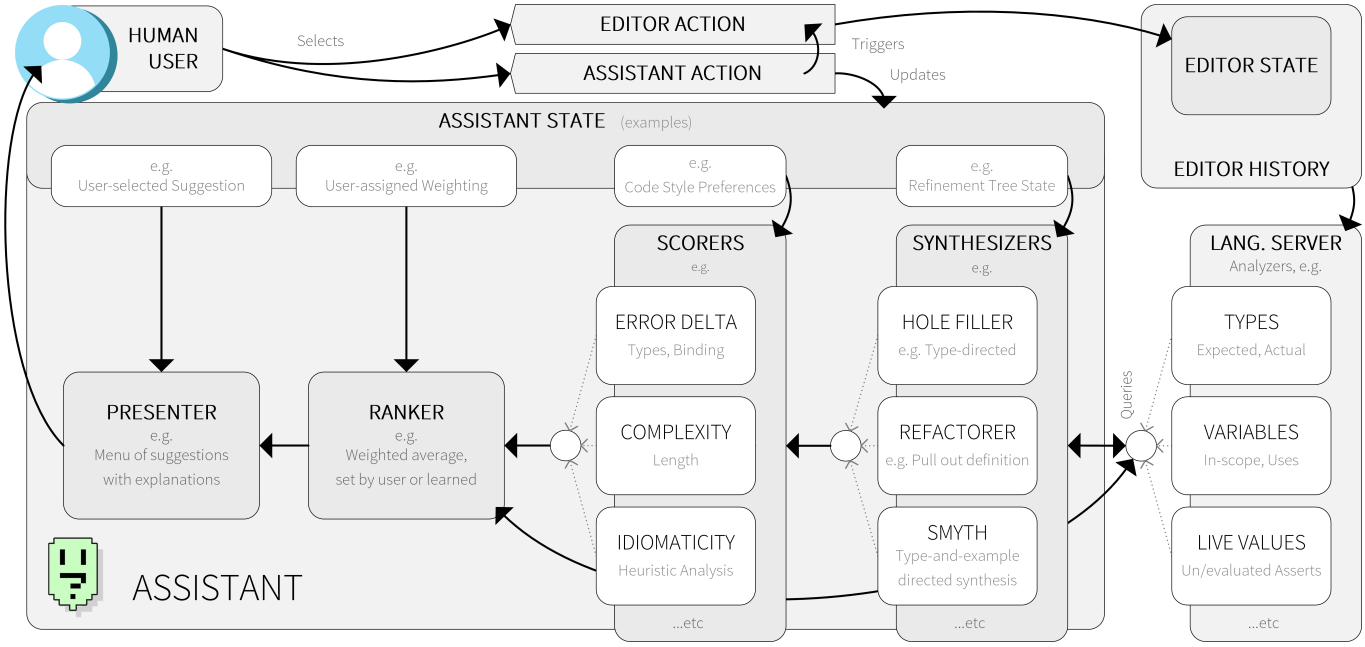

Fig. 1. Integrative human-centered assistant architecture diagram detailing suggestion dataflow and the user interaction loop

II. Architectural Overview

Essentially all program editors support a basic interaction loop whereby a human user triggers editor actions, resulting in an updated editor state [23]. The editor state is presented to the user alongside various editor services that provide additional feedback, e.g. syntax highlighting and type feedback [24]. This feedback requires performing language-aware analyses. Because the same analyses might be relevant to multiple services and across multiple editors, analyzers are typically collected behind a shared *language server* interface [25].

Analyses made available by a language server also feed the programming assistant (Fig. 1.Assistant). An important consideration, particularly relevant to programming assistants, is that these servers must be able to cope with program sketches, i.e. editor states that are not yet syntactically valid or complete, or where there are static errors [23, 26]. When unable to do so, this can lead to gaps in service. Consequently, there has been much effort put into heuristics such as error-recovery and incremental parsing [25, 27, 28] or in automatic hole insertion [23, 26]. In the context of a programming assistant, it is exactly these incomplete states where assistance is most necessary, so gaps in the availability of the analyzers offered by the language server must be avoided whenever possible.

Turning now to the assistant itself, we see a dataflow beginning with a collection of *Synthesizers* which each generate sets of edit action suggestions with accompanying explanatory metadata, as expressed in these notional type definitions:

```
type Suggestion = (EditAction, Explanation)
type Synthesizer = LanguageServer -> Set(Suggestion)
```

For example, the standard Java code completion Synthesizer generates field name suggestions by requesting the type of the target expression from the language server, and variations of that synthesizer also incorporate edit history [29], examples [6], or abbreviations [30].

Generated suggestions are collated and assessed by our third layer, the *Scorers*, resulting in reports used by the *Ranker* and *Presenter* to convey suggestions to the user:

```
type Scorer = Suggestion -> Score
type ScoreReport = Map(Scorer, Score)
type RankedSuggestion = (Suggestion, RankExplanation)
type Ranker = Map(Suggestion, ScoreReport)
              -> List(RankedSuggestion)
```

A variety of ranking and sorting methods have been previously considered [4] including alphabetically, by-type, by-relevance, by prevalence in a corpus, and via logical grouping. Explanation of suggestions is under-researched in programming assistants, but has been recognized as increasingly important [31] and *explainable AI* is a burgeoning topic [32].

Finally, the ranked suggestions are presented to the user together with various affordances, i.e. assistant actions, for updating the assistant state, e.g. to sort, filter, or interact with the components just described. While interaction with suggestions is often simply selection from a menu, more involved interaction models include active code completion via palettes [33], interactive example augmentation [22], and work on the Read-Eval-Synthesize-Loop [34], which presents a Read-Eval-Print-Loop-inspired interaction model for driving human-in-the-loop synthesis.

III. Hazel Assistant

The Hazel programming environment [23] provides a compelling setting for explorations in programming assistant design. It is a structure editor with a formalized editor action semantics, and it avoids the language server gap problem described above, providing continual static and dynamic analyses even for program sketches [35].

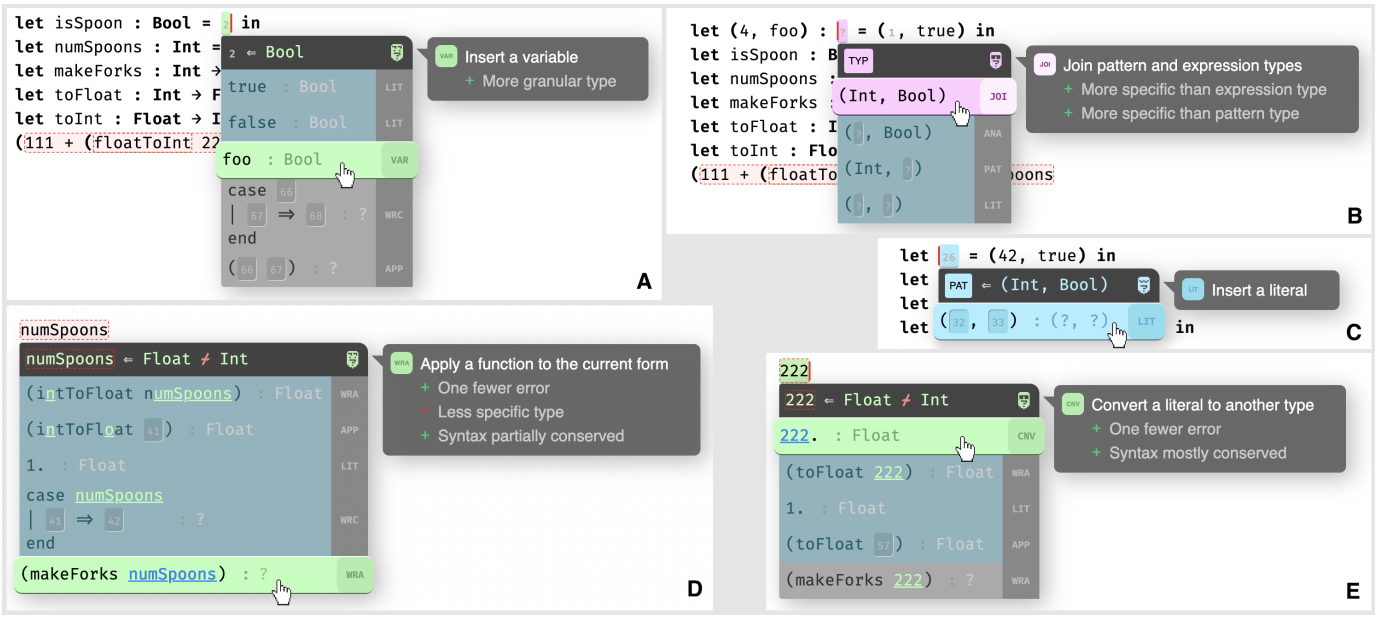

Fig. 2. The Hazel Assistant: (2.A): Completion menu for an expression hole of type `Bool`. The assistant is not limited to expression completion; it can also use type inference to refine type annotations (2.B) and patterns (2.C). (2.D): Wrapper synthesizer being used for code repair. Both the first and last options result in fewer errors, but the last (selected) is ranked down as it results in a less specific type. (2.E): Converter synthesizer targeting numeric type errors

The Hazel Assistant shown in Fig. 2 is a working prototype of a completion and repair assistant for Hazel, serving as a simple end-to-end instantiation of our architecture.

The prototype integrates various Synthesizers that focus on using cursor-local syntactic and static Analyzers to suggest local code transformations. As we are operating on a program sketch–a program with explicit syntactic holes–this task characterization covers both code completion (when the term is an empty hole) and code repair (where the term has a non-empty hole around it, indicating a type error). For type-correct terms, the assistant still provide suggestions for possible transformations, providing lightweight ambient awareness of implementation alternatives.

- **Type Analyzer**: Determines the expected (analytic) type and current (synthetic) type at the cursor
- **Binding Analyzer**: Collects bound variables and uses
- **Syntactic Context Analyzer**: Manages an ascending list of enclosing syntactic forms, rooted at the cursor term and including its parent and ancestors

These Analyzers are queried by various Synthesizers:

- **Inserter Synthesizers**: These implement basic type-aware code completion, ignoring cursor term content and suggesting a wholesale replacement. For example, at a hole with expected type `Bool`, the Literal Synthesizer suggests `true` and `false` (Fig. 2.A)
- **Wrapper Synthesizers**: These take into account the type of the current term to suggest wrapping it within a larger term. For example the application synthesizer may suggest wrapping a term by applying a function which consumes that term's type and produces the expected type
- **Converter Synthesizers**: These suggest conversions between types, e.g. `Float`/`Int` conversions (Fig. 2.E)

Each synthesizer emits a set of suggestions, each equipped with an explanation related to the synthesizer that suggested it. Suggestions are pooled together and fed to the Scorers, who rate each suggestion, possibly with further Language Server consultation. The Hazel Assistant currently employs the following Scorers:

- **Error Delta Scorer**: Determines the integer change in the number of static errors (Fig. 2.D-E)
- **Idiomaticity Scorer**: Performs a heuristic assessment of the idiomaticity of the result, based on a comparison against a fixed list of non-idiomatic syntactic patterns such as using a lambda directly in an application
- **Type Specificity Scorer**: Compares the current and resulting types. This is positive if the resulting type is more specific, such as moving from ? (unknown) to `Int`
- **Syntax Conservation Scorer**: Compares the string representation of the current term and its suggested replacement via Levenshtein distance

Each suggestion's scores are collected in a ScoreReport and passed to the Ranker, which uses a set score weighting to sort the set of suggestions. The Presenter surfaces the ranked suggestions as a scrollable list. On hover, a suggestion's explanation is shown as well as an explanation for the ranking in terms of the individual scores. Selecting a suggestion triggers the corresponding EditAction and closes the interaction loop. Interactions with the Presenter, e.g. moving up and down, are Assistant Actions and update only the Assistant State.

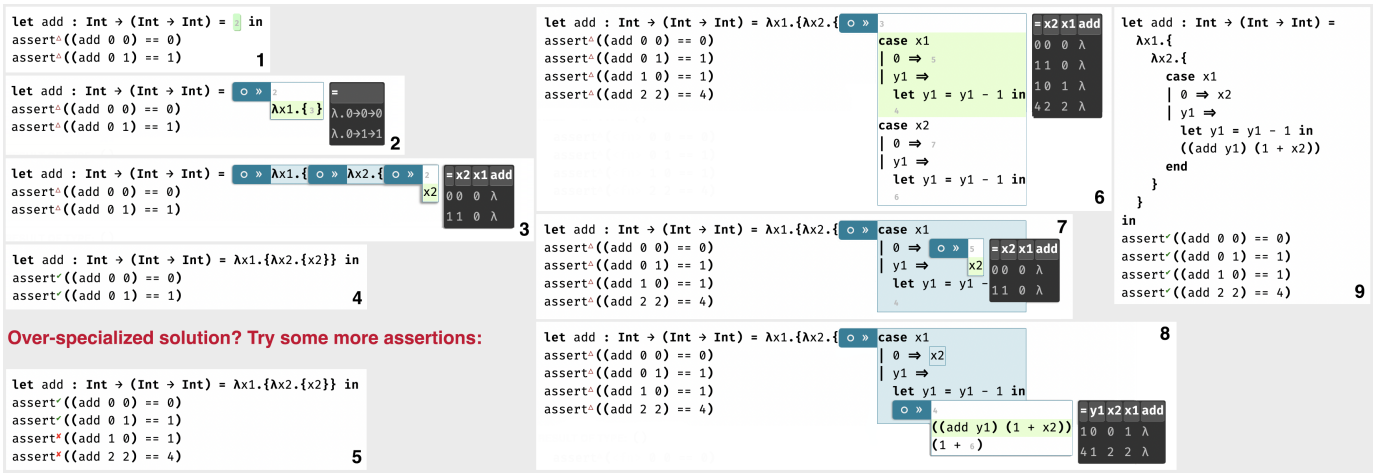

Fig. 3. Hazel Live Assistant: Here we collaborate with the Smyth synthesizer to write a function to add Peano-representation integers. Here we are working around the fact that the Smyth synthesizer supports only algebraic data types, which are not supported by Hazel; we translate the successor constructor to "+ 1" and destructure a successor by subtracting 1. (3.1) portrays a stubbed-out function with two user-provided examples. (3.2-3.4) show the process of *stepping through* a synthesis refinement tree: The user is offered a menu of options; at these stages there is only one suggested completion. The black panel displays the unevaluated constraints which must be satisfied. (3.5) shows a finished but overspecialized solution; the user resolves this by stepping out of synthesis and adding two more examples. (3.6-3.8) represent the result of deleting the "x2" reference and resuming synthesis. This time the user has more options; either casing on x1 or x2 (3.6), and adding 1 before or after the recursive call (3.8). (3.9) shows the completed function

IV. Hazel Live Assistant

The Hazel Live Assistant extends the Base Assistant by integrating with a more sophisticated external type-and-example-directed synthesizer, Smyth [36]. To support human-in-the-loop synthesis it employs a more complex interaction, with the Smyth synthesizer incorporating non-trivial retained state.

In the Hazel editor, all expressions, including incomplete ones, have well-defined evaluation results. Via a process called live evaluation [35], empty holes are propagated as placeholders into the evaluated result, and ill-typed expressions are partially evaluated by placing them in *non-empty holes* and evaluating around them. Smyth additionally uses *live unevaluation* to propagate the values from assertions backward through the program as additional synthesis constraints. Thus the Live Assistant demonstrates extending the reach of Analyzers into program execution.

In the small, the Live Assistant behaves similarly to the Base Assistant. When activated on a hole, it presents a list of candidate hole fillings which may be navigated via the up/down arrows. Each entry in this case is not only type compatible, but represents a refinement step which, possibly after further refinements, may result in a term satisfying the provided assertions.

Like the Base Assistant, the Live Assistant explains suggestions. The rows of the black boxes in Figure 3 represent constraints. The column marked '=' indicates the values the current term must take to satisfy the assertions when the variables take on the values indicated in the other columns. Unlike the Base Assistant, the Live Assistant is a suggestion synthesizer that maintains nontrivial state. If the user accepts a non-terminal refinement – that is, a refinement which contains holes – the suggestion is not immediately committed. Rather, the suggestion menu advances to the first contained hole, stepping deeper into the refinement tree. This UI expedites user-directed backtracking, allowing easy exploration of forks in the synthesis process by binding the left/right arrow keys to forward/backward movement in the refinement tree.

This process of flexible exploration is also exploited in the synthesis back-end, as the refinement tree is lazily generated, allowing the possibility of the human in the loop manually resolving chokepoints where the constraints in the code itself do not sufficiently restrict the search space.

V. Discussion

This paper attempts to organize the burgeon area of programming assistant design with a high level architecture and terminology, and then demonstrates the feasibility of this architecture for design explorations with two prototype assistants, both integrated into the Hazel platform. These design explorations highlight design criteria that are perhaps understudied: semantic ranking, integration, explanations, and human-in-the-loop interactions with synthesizers. We hope that this work helps to bring together various communities working on individual components of the overall system design and ultimately to tap the creative and collaborative potential of human-in-the-loop programming assistants.

Our own design efforts remain ongoing. One key direction is to directly incorporate AI techniques, in particular deep reinforcement learning, which also is oriented around an action space and a reward/scoring structure (human acceptance, tests passing, type errors resolution). We are also working to incorporate more complex multistage refactorings via interactive monadic edit actions, extending work on edit-time tactics in proof assistants [37].

VI. Acknowledgements

We would like to thank Xinyu Wang for his encouragement and support; the Hazel Live Assistant began as a project in his program synthesis class. We would also like to thank Justin Lubin for his patient and comprehensive advice in properly integrating the Smyth synthesizer.

References

[1] C. Rich, H. E. Shrobe, and R. C. Waters, "Overview of the Programmer's Apprentice," in *Sixth International Joint Conference on Artificial Intelligence, IJCAI 79*, 1979, pp. 827–828.

[2] C. Rich and H. E. Shrobe, "Initial Report on a Lisp Programmer's Apprentice," *IEEE Trans. Software Eng.*, vol. 4, no. 6, pp. 456–467, 1978. [Online]. Available: https://doi.org/10.1109/TSE.1978.233869

[3] H. E. Shrobe, B. Katz, and R. Davis, "Towards a Programmer's Apprentice (Again)," in *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015, pp. 4062–4066. [Online]. Available: http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9672

[4] D. Hou and D. M. Pletcher, "An Evaluation of the Strategies of Sorting, Filtering, and Grouping API Methods for Code Completion," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2011, pp. 233–242.

[5] R. Robbes and M. Lanza, "How Program History Can Improve Code Completion," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2008, pp. 317–326.

[6] M. Bruch, M. Monperrus, and M. Mezini, "Learning from Examples to Improve Code Completion Systems," in *7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium*, 2009, pp. 213–222.

[7] G. C. Murphy, M. Kersten, and L. Findlater, "How Are Java Software Developers Using the Eclipse IDE?" *IEEE Software*, vol. 23, no. 4, pp. 76–83, 2006.

[8] S. Gulwani, O. Polozov, and R. Singh, "Program Synthesis," *Found. Trends Program. Lang.*, vol. 4, no. 1-2, pp. 1–119, 2017. [Online]. Available: https://doi.org/10.1561/2500000010

[9] P.-M. Osera and S. Zdancewic, "Type-and-Example-Directed Program Synthesis," *Programming Language Design and Implementation (PLDI)*, vol. 50, no. 6, pp. 619–630, 2015.

[10] J. Frankle, P.-M. Osera, D. Walker, and S. Zdancewic, "Example-Directed Synthesis: A Type-Theoretic Interpretation," in *Symposium on Principles of Programming Languages (POPL)*, 2016.

[11] A. Solar-Lezama, "Program Sketching," *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 5, pp. 475–495, 2013.

[12] A. Miltner, S. Gulwani, V. Le, A. Leung, A. Radhakrishna, G. Soares, A. Tiwari, and A. Udupa, "On the Fly Synthesis of Edit Suggestions," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, oct 2019. [Online]. Available: https://doi.org/10.1145/3360569

[13] T. A. Lau, P. M. Domingos, and D. S. Weld, "Version Space Algebra and its Application to Programming by Demonstration." in *ICML*, 2000, pp. 527–534.

[14] N. Polikarpova, I. Kuraj, and A. Solar-Lezama, "Program Synthesis From Polymorphic Refinement Types," *Programming Language Design and Implementation (PLDI)*, vol. 51, no. 6, pp. 522–538, 2016.

[15] M. P. Gissurarson, "Suggesting Valid Hole Fits for Typed-Holes (Experience Report)," *ACM SIGPLAN International Symposium on Haskell 11*, vol. 53, no. 7, pp. 179–185, 2018.

[16] ——, "The Hole Story: Type-Driven Synthesis and Repair," Licentiate Thesis, 2022.

[17] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating Large Language Models Trained on Code," 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[18] E. C. R. Shin, M. Allamanis, M. Brockschmidt, and A. Polozov, "Program Synthesis and Semantic Parsing with Learned Code Idioms," in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems (NeurIPS)*, H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett, Eds., 2019, pp. 10 824–10 834.

[19] V. Raychev, M. Vechev, and E. Yahav, "Code Completion with Statistical Language Models," in *Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, 2014, p. 419–428. [Online]. Available: https://doi.org/10.1145/2594291.2594321

[20] J. Li, Y. Wang, I. King, and M. R. Lyu, "Code Completion with Neural Attention and Pointer Networks," *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI'18)*, 2017. [Online]. Available: http://arxiv.org/abs/1711.09573

[21] T. Zhang, Z. Chen, Y. Zhu, P. Vaithilingam, X. Wang, and E. L. Glassman, "Interpretable Program Synthesis," in *2021 CHI Conference on Human Factors in Computing Systems*, ser. CHI '21. Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3411764.3445646

[22] T. Zhang, L. Lowmanstone, X. Wang, and E. L. Glassman, *Interactive Program Synthesis by Augmented Examples*. UIST, 2020, p. 627–648. [Online]. Available: https://doi.org/10.1145/3379337.3415900

[23] C. Omar, I. Voysey, M. Hilton, J. Aldrich, and M. A. Hammer, "Hazelnut: A Bidirectionally Typed Structure Editor Calculus," in *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2017.

[24] H. Potter and C. Omar, "Hazel Tutor: Guiding Novices Through Type-Driven Development Strategies," *Human Aspects of Types and Reasoning Assistants (HATRA)*, 2020.

[25] F. Bour, T. Refis, and G. Scherer, "Merlin: A Language Server for OCaml (Experience Report)," *Proc. ACM Program. Lang.*, vol. 2, no. ICFP, jul 2018. [Online]. Available: https://doi.org/10.1145/3236798

[26] C. Omar, I. Voysey, M. Hilton, J. Sunshine, C. Le Goues, J. Aldrich, and M. A. Hammer, "Toward Semantic Foundations for Program Editors," in *Summit on Advances in Programming Languages (SNAPL)*, 2017.

[27] S. L. Graham, C. B. Haley, and W. N. Joy, "Practical LR Error Recovery," in *SIGPLAN Symposium on Compiler Construction (CC)*, 1979.

[28] M. de Jonge, E. Nilsson-Nyman, L. C. L. Kats, and E. Visser, "Natural and Flexible Error Recovery for Generated Parsers," in *Software Language Engineering (SLE)*, 2009.

[29] R. Robbes and M. Lanza, "How Program History Can Improve Code Completion," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008, pp. 317–326.

[30] S. Han, D. R. Wallace, and R. C. Miller, "Code Completion from Abbreviated Input," in *2009 IEEE/ACM International Conference on Automated Software Engineering*, 2009, pp. 332–343.

[31] H. Finkel and I. Laguna, "Report of the Workshop on Program Synthesis for Scientific Computing," 2021. [Online]. Available: https://arxiv.org/abs/2102.01687

[32] D. Doran, S. Schulz, and T. R. Besold, "What Does Explainable AI Really Mean? A New Conceptualization of Perspectives," *arXiv preprint arXiv:1710.00794*, 2017.

[33] C. Omar, Y. S. Yoon, T. D. LaToza, and B. A. Myers, "Active Code Completion," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 859–869.

[34] H. Peleg, R. Gabay, S. Itzhaky, and E. Yahav, "Programming with a Read-Eval-Synth Loop," *OOPSLA*, vol. 4, nov 2020. [Online]. Available: https://doi.org/10.1145/3428227

[35] C. Omar, I. Voysey, R. Chugh, and M. A. Hammer, "Live Functional Programming with Typed Holes," *Proceedings of the ACM on Programming Languages (PACMPL), Issue POPL*, 2019.

[36] J. Lubin, N. Collins, C. Omar, and R. Chugh, "Program Sketching with Live Bidirectional Evaluation," *Proceedings of the ACM on Programming Languages*, vol. 4, no. ICFP, pp. 1–29, 2020.

[37] J. Korkut, "Edit-Time Tactics in Idris," Master's thesis, Wesleyan University, 2018.