Hazel Tutor: Guiding Novices Through Type-Driven Development Strategies

HANNAH POTTER, University of Michigan

CYRUS OMAR, University of Michigan

Hazel Tutor is a work-in-progress editor service for the Hazel programming environment designed to interactively help novices learn type-driven development strategies as they construct, edit, and debug code. The system provides both feedback and, when requested, suggestions on the basis of type information available at the cursor. Hazel is able to supply the Hazel Tutor with the necessary information at all times, including when there are *holes* in the program. When the cursor is on an empty hole, the system organizes the suggestions based on the type-driven development strategy that we aim to teach students. We are beginning to work on presenting debugging strategies when on an error hole. We outline our hypotheses, specific research questions of interest, and planned future studies.

1 INTRODUCTION

Type-driven functional programming is powerful, but it can also be challenging to both novices and experts. Novices struggle in particular with developing the correct mental model of the type system of the language and using that mental model strategically to narrow down the search space as they construct and manipulate programs [16, 18]. Even more experienced programmers can be uncertain about their mental models, especially when learning a new language, using advanced features, or working with complex code. Additionally, when working with unfamiliar APIs, both novices and experts can benefit from an understanding of the type structure of the API.

Compilers provide feedback, and occasionally suggestions, related to type structure, but only once there is a type error in the program. Modern IDE services provide feedback and suggestions even in situations where there is not an error. However, these services are limited and often entirely unavailable when there is a parse error or type error anywhere in the program, even far away from the cursor. These situations are quite common during the development process, and particularly so when the programmer is inexperienced. Moreover, feedback and assistance is most useful exactly in situations where the program is incomplete.

The Hazel is a live functional programming environment that solves this so-called *gap problem* by assigning both static and dynamic meaning to programs with holes. Empty holes stand for missing pieces of the program, and non-empty holes are membranes around errors. Holes are added automatically. This eliminates semantic gaps: every editor state in Hazel has both a type and a result. This information is intended to be used by editor services to provide feedback and assistance throughout the development process [12–14].

This paper introduces one such editor service, the Hazel Tutor, which is a pop-up window that follows the cursor around and provides information about the expected and actual types of the expression or pattern that the cursor is on. This serves to move the burden of determining this information out of the programmer's head—instead, it is always in their eyeline. This information is critical when following a type-driven development strategy. The expected type constrains the space of possible expressions, often quite substantially, and the actual type is useful both to verify that the expression has the intended type and, moreover, to help when debugging type errors.

Take the following general functional code snippet as an example, where the programmer's cursor (indicated by the | character) is at the `else` branch of the `if` expression:

Authors' addresses: Hannah Potter, hkpotter@umich.edu, University of Michigan; Cyrus Omar, comar@umich.edu, University of Michigan.

```
let f = fun (x : Int, y : Bool) -> if y then foo x else |
```

To figure out the type of the expression that must be used to fill in that branch, the programmer may think about the types of foo, x, and foo applied to x. Even in a short, simple example such as this one, a programmer is burdened with the cognitive load of thinking through multiple expressions and their types to fill in the else branch with an expression of the correct type. This example would not even parse in most languages, thus the programmer would be forced to think through the types without help from the system.

We are in the process of designing and developing the Hazel Tutor (Section 2) to aide novices (and eventually expert programmers) as they engage in type-driven development tasks designed to teach correct mental models of the Hazel type system. We make explicit what is often only available implicitly, what the programmer often only becomes aware of once the compiler produces an error, or what is only available in a limited form from an IDE. This information is used in both parts of the Hazel Tutor user interface: the Cursor Inspector and the Strategy Guide. The Cursor Inspector allows a programmer to see the expected and actual types of every expression and pattern in a program. Additionally, when requested, the Strategy Guide gives suggestions of how to fill in a hole with an expression of the expected type, following a type-driven development strategy that we aim to teach students. We plan to evaluate our hypotheses and specific research questions about the usefulness of such a system (Section 3).

2 HAZEL TUTOR BY EXAMPLE

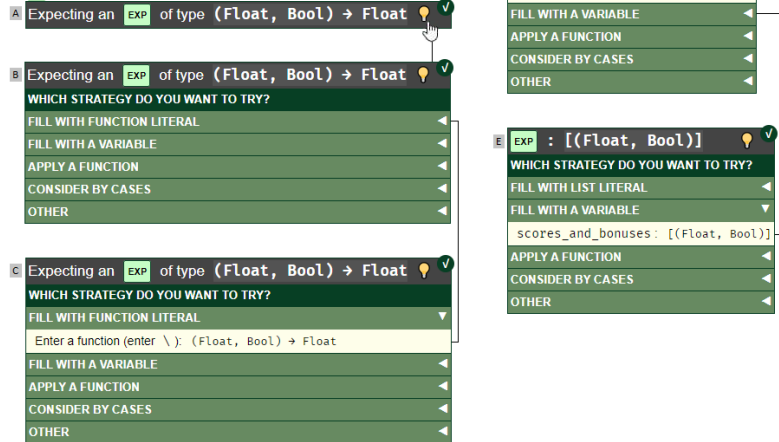Fig. 1. The student starter code for the convert_scores problem. Each step is described in the text.

Let us start with an example to walk through the use and functionality of the Hazel Tutor. We will start with a typical programming problem for an introductory functional programming class. The students in this course are familiar with programming, but not functional programming. The class has covered a few weeks worth of material and the students are learning how to use the map function on Lists. Figure 1 shows the starter code for implementing a function named convert_scores.

The function should take a list of submission data for an assignment and produce a list of final scores. The submission data is represented as pairs of a raw submission score (that has type `Float`) and an indicator of whether or not a bonus question was answered correctly (that has type `Bool`). The function should work such that a submission earns 2.5 extra points over the raw score if the bonus question was answered correctly; otherwise, the the submission simply earns the raw score. `convert_scores` must use the given `map` function (appropriately specialized for this problem).[1]

2.1 Code Comprehension

A student begins the problem by investigating the starter code. Two empty holes (numbered automatically 26 and 24) appear as the two arguments to `map`. This student does not remember the order of arguments to `map`, therefore they move their cursor to the hole for the first argument. They see using the Cursor Inspector (Figure 1A) that the expected type for the first argument is `(Float, Bool) -> Float`. The student then moves their cursor to the hole for the second argument and sees that the expected type is `[(Float, Bool)]` (Figure 1D).

Note that this information is available not just for holes, but also when the cursor is on any expression or pattern in the program (further discussed below). Note also that this type information is not only useful to novices, but also to experts, for instance when using an unfamiliar API.

To help make the information displayed by the Cursor Inspector more digestible for novices as well as concise for experts, a programmer is able to easily toggle between two modes of messaging by clicking on the message. Novice specific messaging has been suggested by other works [1, 9, 10], and we follow the same principle here. The novice version of messaging uses more words over symbols, such as using "of type" (see Figure 1A) instead of ":" (see Figure 1D). As programmers become more familiar with the symbols through tutorials or course instruction, or simply by toggling between the two modes of the Cursor Inspector, they can transition to using and understanding the more concise, expert mode.

2.2 Editing

The student now knows the expected type of the first argument to `map`, but is unsure of how to fill in this hole. The student selects the hint icon on the Cursor Inspector, and the Strategy Guide is opened (Figure 1B). They see ordered steps for exploring different expressions of the correct type that can be used to fill in the hole, following a type-driven development strategy. The first option is to use literals of the correct type followed by the second set of options to use variables of the correct type. The next set of options are function applications that will result in an expression of the correct type. Next is the set of options to consider by cases, followed by the last set of other, more complex options. By grouping the choices in this way, we prompt the user to go from simpler options (variables, introductory forms) to more complex options (elimination forms), while filtering out the large number of constructs that are not likely to be relevant, such as literals and variables of the incorrect type. The student decides to make a new function (Figure 1C), but chooses to create a new let binding over inlining the function. The Strategy Guide does not serve as an auto-fill tool, but rather an understanding aide, so the student types out their choice.

Before filling the function body, the student moves to the second argument of `map`. The student decides that a list literal (Figure 1D), though having the correct type, would not fit the desired functionality. They choose to look at the next set of options and see a variable that has the correct type and will give the correct functionality: `scores_and_bonuses` (Figure 1E).

While having the type information of the expression at the cursor can be helpful in guiding the programmer on how to fill in holes correctly and edit a program, some programmers may not

---

[1]At the time of writing, Hazel does not support polymorphism.

```
let bonus = 2.5 in
let convert_scores : [(Float, Bool)] → [Float] = ↵
  fun scores_and_bonuses.{
    let apply_bonus : (Float, Bool) → Float = ↵
      fun (raw_score, bonus_question).{ ↵
        let additional_points = 43  in
        raw_score + additional_points
      }                        EXP : Float ↪ Int  ⚠
    in
    map apply_bonus scores_and_bonuses
  }
in
  19
```
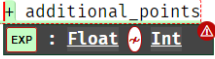
Fig. 2. The Cursor Inspector shows that there is a type error because the type of the expression is Int, which is inconsistent with the expected type Float. (Novice mode, not shown, makes the order of expected and actual types clear.)

want to see this information at all times. In consideration of this, programmers are able to easily show and hide the Hazel Tutor, using a simple keyboard shortcut. This is one of the efforts made to minimize distraction. Additionally, minimally-changing background colors and icons are used to avoid drawing unwarranted attention to the Hazel Tutor user interface. This is true even when there is a type error, discussed below. To further help minimize distraction while still keeping the user interface in a position of attention, the Cursor Inspector moves with the cursor at a token level as opposed to a character level. We mention in Section 3 plans to evaluate these design choices.

2.3 Fixing Type Errors

Now, the student moves on to filling the body of the function that the Hazel Tutor helped them create (Figure 2). This function takes an element of the list, which is a tuple of the raw_score and whether or not the bonus_question was answered correctly. The student knows that they need to somehow calculate the number of additional_points that a submission earned and add that to the raw_score, so they stub this out. When adding this code, the student sees an error in the Cursor Inspector. As shown in Figure 2, we see that the student mistakenly used integer addition (the + operator) as opposed to floating point addition (the +. operator).[2] The student is quickly able to inspect the types of the expressions involved and correct the error.

The type information displayed by the Cursor Inspector aides programmers in such tasks, identifying and correcting type errors, by presenting information about the expected and actual types of expressions. In future work, we plan to use the Strategy Guide in some form to help programmers work through a debugging strategy to correct type errors.

2.4 Branching

Now the student moves on to filling in how to calculate the additional_points. The student is unsure how to fill in this hole, so they again make use of the Strategy Guide, as seen in Figure 3. The student looks down the list of options. The student sees in the "Consider by Cases" section the option to branch on bonus_question and recognizes that this is the functionality they want.[3]

Now the student moves to filling in the branches of the case expression. After an initial attempt, the student can clearly see that they have a type error in their case expression because it is outlined in red, as shown in Figure 4. The student selects the case expression to see what the error is. They

---

[2]The + operator is not overloaded in Hazel, just as it is not in OCaml.

[3]if expressions are not currently in the Hazel language. When such a form is added, we plan to make the Hazel Tutor choose the most appropriate branching form in a type-directed way, based on the scrutinee.
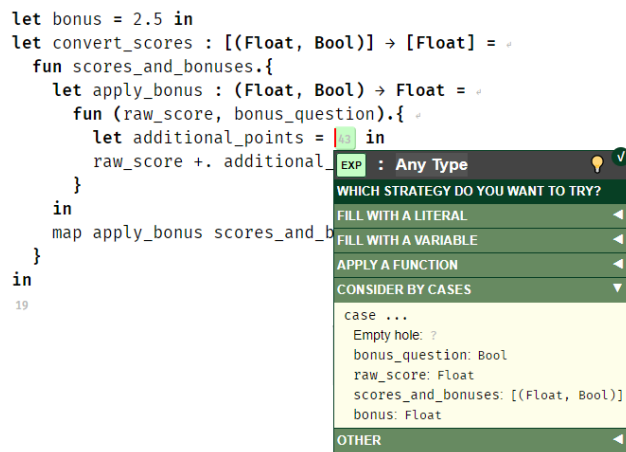
Fig. 3. The Strategy Guide in use to help the student fill in the hole to correctly calculate the number of additional points that need to be added to the raw score based on whether or not the bonus question was answered correctly.
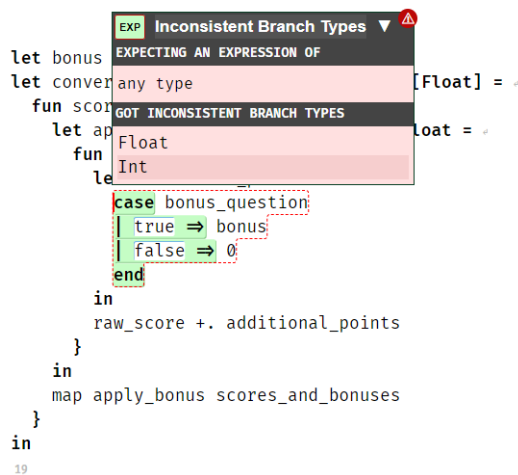

Fig. 4. The Cursor Inspector shows the programmer the type error. When the error is due to inconsistent branch types in a case expression, the programmer can inspect the type of each of the branches.

see from the Cursor Inspector that the error is that the `case` expression has inconsistent branch types. The student expands the message to get more detailed information about the error. Some forms of messages in the Cursor Inspector have expanded versions that can be used to provide extra, relevant information. Now the student can see that the issue is that the first branch has type `Int` and the second branch has type `Float`. When there are inconsistent branch types, Hazel does not guess which should be used as the correct type (as is the protocol in other systems), but instead wraps the whole `case` expression in a non-empty hole, marking the error, and the Cursor Inspector shows the type of all the branches. The student now identifies the error as that each of the branches should have type `Float` and fixes the erroneous second branch to be `0.0`, completing

the `convert_scores` function. Throughout the assignment, we see that the student was able to make use of the Cursor Inspector and the Strategy Guide to follow a type-driven strategy.

3   HYPOTHESES AND STUDIES

We have formulated some hypotheses about the effectiveness of our tool. We are in the process of planning studies to test these hypotheses and develop new ones.

3.1   Hypotheses

Our research of the literature around compiler error messaging, type debugging, teaching introductory programming, as well as our own experiences teaching novices to functional programming, has guided our design of the Cursor Inspector and Strategy Guide and influenced our hypotheses of the effectiveness and results of using our tool. Our main hypotheses are:

(H1) The Hazel Tutor system will help students develop a more accurate mental model of the Hazel language and type system.

(H2) Students will develop effective strategies for filling in holes and correcting type errors by using the Hazel Tutor.

(H3) Programmers will more quickly understand existing code and new APIs when using the Hazel Tutor's Cursor Inspector.

The Cursor Inspector makes always available the expected and actual types of any expression. We believe that when programmers see this information frequently over an extended period of time, they will become more familiar with the type system and be able to better understand and predict how it works. Secondly, we believe that because the Cursor Inspector allows the programmer to easily see the expected type for any hole, they will more accurately fill in holes with expressions of the correct type and quickly correct type errors. Additionally, programmers can make use of the Strategy Guide which gives suggestions of expressions of the correct type to fill a hole, organized into categories that the student can use to develop appropriate mental strategies. Lastly, we believe that the type information that is made available at the cursor will help programmers understand existing code and learn how to correctly use new APIs more quickly.

3.2   Research Questions and Planned Future Studies

To investigate our hypotheses and other concerns around our design, we have developed specific research questions to evaluate in future studies, focused initially on experienced programmers who are novice to functional programming.

(Q1) How does the tutoring system help programmers during type driven development tasks?

    (a) Are programmers able to fill in holes with an expression of the correct type more often when using the Cursor Inspector and Strategy Guide than without it?

       (i) Do students hide the Cursor Inspector or leave it visible?

      (ii) Do students interact with the Strategy Guide? How often? Does this change as they become more proficient?

    (b) Are programmers able to correct type errors more quickly and accurately when using the system than without it?

       (i) How do programmers use the Cursor Inspector when type debugging?

      (ii) Are there features not available in Hazel Tutor that the programmers felt would have been helpful for debugging?

    (c) Does the Cursor Inspector help programmers understand existing code more quickly or accurately?

    (d) With what sorts of programming tasks is the tutoring system most helpful? Least helpful?

(e) Do users find the Cursor Inspector distracting when editing?

(Q2) Does long-term use of the tutoring system improve student performance on assessments of program understanding?

These research questions (numbered independently from the hypotheses) cannot all be answered with a single study, thus we are planning on a series of initial studies to gather early results and insights about our design and implementation upon which we can iterate.

For our initial studies, our participants will be students in an undergraduate Programming Languages courses. These are primarily upper-division computer science students, but are typically novices to functional programming, making them a reasonable population from which to draw based on our research questions. Additionally, we have gathered logging data about code compilations and submissions of assignments done for this course using Learn OCaml [5]. We believe we might be able to gain insights comparing data between the two learning environments, recognizing that there are many confounding variables to consider.

We plan to run multiple studies, with different sets of tasks. One study will be focused on editing tasks, where participants will be given partial implementations and be asked to fill in holes in the program. Another will be focused on correcting type errors, where participants will be given programs with type-errors and be asked to fix the errors. During both of these studies, we plan to incorporate measures of how distracting participants find the tools, including asking questions as part of a post-task survey. We also plan to use Hazel and the Hazel Tutor in future iterations of the previously mentioned Programming Languages course and make comparisons between the iterations of the course to draw comparisons, particularly looking at type error logging information and assessment scores.

4 FUTURE WORK

We have plans for improving the tutoring system and adding more functionality. Currently, the Cursor Inspector is mostly implemented, and the Strategy Guide is in the process of being implemented. One improvement planned for the Cursor Inspector is to make it pinnable: allow a programmer to "pin" an instance of the Cursor Inspector to an expression. This would allow programmers to view type information about multiple expressions at the same time. We also plan to make the tutor more tailored to individual users by tracking their knowledge of the language to introduce and show documentation of unfamiliar forms. We would also like to improve the Strategy Guide by having it guide programmers through a debugging strategy when the cursor is on an error hole. Additionally, we would like to intelligently prune and order the suggestions given in each step of the Strategy Guide. We are also considering additional tools for the tutoring system, such as links to documentation, a type debugger, and a type derivation visualizer.

5 RELATED WORK

The term *notional machine* is used to describe the abstract model of a computer and is accurate up to the point necessary for the level of abstraction, whereas a student's mental model of the computer may be inaccurate; the goal is to align the student's mental model to the notional machine [2]. Recent work has been done in investigating language semantics as a notional machine in the context of programming education [4]. Misconceptions and difficulties of students learning programming have been partially attributed to incorrect mental models [8, 16, 18]. As such, suggestions have been made that notional machines should be taught explicitly [2]. Our aim is to have the Hazel tutoring system build and confirm student mental models of the type system by surfacing to the UI information about the types of expressions instead of forcing programmers to (perhaps incorrectly) derive this information without help from their programming environment.

In bringing type information to the surface of the UI, we also aim to help programmers fix type errors in their programs. Work has gone into exploring type errors and error messages [1], particularly in how students interact with messages [9, 10], and common novice errors and behaviors of students fixing type errors [19]. Some results have found that longer messages with more information do not necessarily help novices understand messages [11] and that error messages should reduce the cognitive load of the programmer [1].

Type debuggers help programmers locate and understand type errors in their programs. The OCaml Type Debugger is designed keeping novice programmers in mind, providing information needed to help programmers see why an error message occurred [6]. The Cursor Inspector allows programmers to view the types of *all* of the expressions in a program. Additionally, like the OCaml Type Debugger, Hazel provides expression specific error messages. However, at this stage of the design and implementation process, the Cursor Inspector does not guide programmers through this investigation process of type debugging. Such guidance is part of our future work. Another tool designed to interactively allow programmers to inspect the types of expressions is Typeview [17]. However, this tool is limited in functionality for programs that do not type check, whereas the Cursor Inspector does not face this issue because there are no meaningless editor states in Hazel. Another feature of Typeview is that it allows programmers to see the types of multiple expressions at the same time, which is functionality we plan to add by making the Cursor Inspector pinnable.

There are other lines of work aimed at helping programmers other than fixing type errors. Recent work on guiding programmers though explicit programming strategies has shown some promising results about keeping programmers more organized and solving problems more systematically [7]. The current design of Hazel Tutor does not give the same level of step-by-step instruction as the system presented in that work. However, the Strategy Guide still presents information in an ordered manner, suggesting to programmers the order in which they should think through options of expressions to fill holes. Generated hints are another way to help programmers, such as iSNAP [15]. In a pilot study of that tool, the authors recognized the need to prevent abuse of excessive hint requests and to help students understand the goal behind such hints. Making users type their chosen option from the Strategy Guide is one attempt to prevent blind following of suggestions.

Tutoring systems are another way to provide help and guidance to programmers. Ask-Elle is a functional tutoring system for Haskell that sets up programming exercises as a series of hole refinements, using model tracing to compare student programs with solutions [3]. The Hazel tutoring system is similar in that it helps programmers fill in holes, but is more flexible in that it does not need preset solutions to provide hints and guidance.

6 CONCLUSION

In this paper, we presented our work-in-progress for the Hazel Tutor, designed to help programmers with type-driven development in Hazel. Main features of this tool include the Cursor Inspector that provides always available type information at the cursor and, in the case of a type error, error messaging. Additionally, we designed the Strategy Guide to provide suggestions for filling in empty holes with expressions of the correct type following a type-driven strategy. Studies will be necessary to evaluate the effectiveness of these tools in aiding programmers in understanding existing code, editing programs, fixing type errors, understanding the type system, and learning to follow a type-driven development strategy.

REFERENCES

[1] Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. 2019. Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education* (Aberdeen, Scotland Uk) *(ITiCSE-WGR '19)*. Association for Computing Machinery, New York, NY, USA, 177–210. https://doi.org/10.1145/3344429.3372508

[2] Paul E. Dickson, Neil C. C. Brown, and Brett A. Becker. 2020. Engage Against the Machine: Rise of the Notional Machines as Effective Pedagogical Devices. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education* (Trondheim, Norway) *(ITiCSE '20)*. Association for Computing Machinery, New York, NY, USA, 159–165. https://doi.org/10.1145/3341525.3387404

[3] Alex Gerdes, Bastiaan Heeren, Johan Jeuring, and L. Thomas van Binsbergen. 2017. Ask-Elle: an Adaptable Programming Tutor for Haskell Giving Automated Feedback. *International Journal of Artificial Intelligence in Education* 27, 1 (01 Mar 2017), 65–100. https://doi.org/10.1007/s40593-015-0080-x

[4] Mark Guzdial, Shriram Krishnamurthi, Juha Sorva, and Jan Vahrenhold. 2019. Notional Machines and Programming Language Semantics in Education (Dagstuhl Seminar 19281). *Dagstuhl Reports* 9, 7 (2019), 1–23. https://doi.org/10.4230/DagRep.9.7.1

[5] Aliya Hameer and Brigitte Pientka. 2019. Teaching the Art of Functional Programming Using Automated Grading (Experience Report). *Proc. ACM Program. Lang.* 3, ICFP, Article 115 (July 2019), 15 pages. https://doi.org/10.1145/3341719

[6] Yuki Ishii and Kenichi Asai. 2014. Report on a User Test and Extension of a Type Debugger for Novice Programmers. In *Proceedings 3rd International Workshop on Trends in Functional Programming in Education, TFPIE 2014, Soesterberg, The Netherlands, 25th May 2014 (EPTCS, Vol. 170)*, James Caldwell, Philip K. F. Hölzenspies, and Peter Achten (Eds.). 1–18. https://doi.org/10.4204/EPTCS.170.1

[7] Thomas D. LaToza, Maryam Arab, Dastyni Loksa, and Amy J. Ko. 2020. Explicit Programming Strategies. *Empirical Software Engineering* 25, 4 (01 Jul 2020), 2416–2449. https://doi.org/10.1007/s10664-020-09810-1

[8] Andrew Luxton-Reilly, Simon, Ibrahim Albluwi, Brett A. Becker, Michail Giannakos, Amruth N. Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. 2018. Introductory Programming: A Systematic Literature Review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education* (Larnaca, Cyprus) *(ITiCSE 2018 Companion)*. Association for Computing Machinery, New York, NY, USA, 55–106. https://doi.org/10.1145/3293881.3295779

[9] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011. Measuring the Effectiveness of Error Messages Designed for Novice Programmers. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education* (Dallas, TX, USA) *(SIGCSE '11)*. Association for Computing Machinery, New York, NY, USA, 499–504. https://doi.org/10.1145/1953163.1953308

[10] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011. Mind Your Language: On Novices' Interactions with Error Messages. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Portland, Oregon, USA) *(Onward! 2011)*. Association for Computing Machinery, New York, NY, USA, 3–18. https://doi.org/10.1145/2048237.2048241

[11] Marie-Hélène Nienaltowski, Michela Pedroni, and Bertrand Meyer. 2008. Compiler Error Messages: What Can Help Novices? *SIGCSE Bull.* 40, 1 (March 2008), 168–172. https://doi.org/10.1145/1352322.1352192

[12] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live Functional Programming with Typed Holes. *Proc. ACM Program. Lang.* 3, POPL, Article 14 (Jan. 2019), 32 pages. https://doi.org/10.1145/3290327

[13] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. 2017. Hazelnut: A Bidirectionally Typed Structure Editor Calculus. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 86–99. https://doi.org/10.1145/3009837.3009900

[14] Cyrus Omar, Ian Voysey, Michael Hilton, Joshua Sunshine, Claire Le Goues, Jonathan Aldrich, and Matthew A. Hammer. 2017. Toward Semantic Foundations for Program Editors. *CoRR* abs/1703.08694 (2017). arXiv:1703.08694 http://arxiv.org/abs/1703.08694

[15] Thomas W. Price, Yihuan Dong, and Dragan Lipovac. 2017. ISnap: Towards Intelligent Tutoring in Novice Programming Environments. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (Seattle, Washington, USA) *(SIGCSE '17)*. Association for Computing Machinery, New York, NY, USA, 483–488. https://doi.org/10.1145/3017680.3017762

[16] Yizhou Qian and James Lehman. 2017. Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Trans. Comput. Educ.* 18, 1, Article 1 (Oct. 2017), 24 pages. https://doi.org/10.1145/3077618

[17] Axel Simon, Olaf Chitil, and Frank Huch. 2000. Typeview: A Tool for Understanding Type Errors, In Draft Proceedings of the 12th International Workshop on Implementation of Functional Languages, Markus Mohnen and P. Koopman

(Eds.). *Draft Proceedings of the 12th International Workshop on Implementation of Functional Languages*, 63–69. https://kar.kent.ac.uk/21959/

[18] Ville Tirronen. 2014. Study on Difficulties and Misconceptions with Modern Type Systems. In *Proceedings of the 2014 Conference on Innovation Technology in Computer Science Education* (Uppsala, Sweden) *(ITiCSE '14)*. Association for Computing Machinery, New York, NY, USA, 303–308. https://doi.org/10.1145/2591708.2591726

[19] Baijun Wu and Sheng Chen. 2017. How Type Errors Were Fixed and What Students Did? *Proc. ACM Program. Lang.* 1, OOPSLA, Article 105 (Oct. 2017), 27 pages. https://doi.org/10.1145/3133929