Live and Direct Functional Programming with Holes*

ANONYMOUS AUTHOR(S)

Live programming environments aim to provide programmers with continuous feedback about a program's behavior as it is being edited. The problem is that programming languages typically assign meaning only to programs that are *complete*, i.e. syntactically well-formed and free of type and binding errors. Consequently, the live feedback presented to the programmer suffers from temporal or perceptive gaps.

The Hazel live functional programming environment addresses this "gap problem" from first principles: Hazel assigns rich static and dynamic meaning to every incomplete program that the programmer can construct using Hazel's language of structured edit actions. An incomplete program is represented as an expression with *holes*, where empty holes stand for missing expressions or types, and non-empty holes operate as membranes around type inconsistencies. Rather than aborting evaluation when encountering a hole instance, our approach records the environment around the hole instance—forming a *hole closure*—and then evaluates "around" it. These hole closures enable a *fill-and-resume* operation that avoids the need to restart evaluation after edits that amount to hole filling. Furthermore, various editor services can report information from hole closures to help the programmer decide how to fill the corresponding holes; we describe two such examples in this paper. First, we discuss Hazel's *live context inspector*, which allows the programmer to explore values from relevant hole closures when the cursor is on a hole in the program. Second, we discuss early work on *live palettes*—user interfaces, which can themselves contain holes, that the programmer can directly manipulate to fill an expression hole. A live palette can use the closures associated with the hole being filled to provide specialized, concrete feedback about the dynamic consequences of the programmer's choices. Taken together, the result is a substantially more *live and direct* typed functional programming experience.

1 INTRODUCTION

Programmers typically shift back and forth between program editing and program evaluation many times before converging upon a program that behaves as is intended. Live programming environments aim to support this workflow by interleaving editing and evaluation so as to narrow what Burckhardt et al. [2013] call the "temporal and perceptive gap" between these activities.

The problem at the heart of this work is that programming languages typically assign meaning only to complete programs, i.e. programs that are syntactically well-formed and free of static type and binding errors. A program editor, however, frequently encounters incomplete, and therefore meaningless, editor states. As a result, live feedback either "flickers out", creating a temporal gap, or it "goes stale", i.e. it relies on the most recent complete editor state, creating a perceptive gap because the feedback may not accurately reflect what the programmer is seeing in the editor. In some cases, these gaps can extend over substantial lengths of time.

In recognition of this "gap problem", Omar et al. [2017a] develop a static semantics (i.e. a type system) for incomplete functional programs, modeling them formally as typed expressions with *holes*. Empty holes stand for missing expressions or types, and non-empty holes operate as "membranes" around static type inconsistencies (i.e. they internalize the "red underline" that editors commonly display under a type inconsistency). Holes can be inserted into the program in several ways. In systems where the editor state is a text buffer, error recovery mechanisms can insert holes implicitly [Aho and Peterson 1972; Charles 1991; Kats et al. 2009]. Alternatively, the language might provide explicit notation for holes, so that the programmer can insert them either manually or semi-automatically via code completion [Amorim et al. 2016]. For example, GHC Haskell supports the notation _u for empty holes, where u is an optional hole name [Jones et al. 2014]. Structure editors insert explicitly represented holes fully automatically [Omar et al. 2017a].

*Some text and figures in this submission are taken from a full paper by the authors, which is currently under review.

For the purposes of live programming, however, a static semantics like that developed by Omar et al. [2017a] does not suffice—we also need a corresponding dynamic semantics that specifies how to evaluate expressions with holes. The simplest approach would be to define a dynamic semantics that aborts with an error when evaluation reaches a hole. This mirrors a workaround that programmers commonly deploy: raising a placeholder exception, e.g. `raise Unimplemented`. GHC Haskell supports this mode of evaluation for programs with holes using the `-fdefer-typed-holes` flag.[1] Although better than nothing, this "exceptional approach" to evaluation with holes has limitations in the setting of a live programming environment because (1) it provides no information about the behavior of the remainder of program, parts of which may not depend on the missing or erroneous expression (e.g. later cells in a lab notebook, or tests involving complete components of the program); (2) it provides limited information about the dynamic state of the program where the exception occurs (typically only a stack trace); and (3) it provides no means by which to resume evaluation after filling a hole.

Dynamic Semantics for Functional Programs with Holes. This proposal is based on our ongoing effort to develop a dynamic semantics for incomplete functional programs, starting from the static semantics developed by Omar et al. [2017a], that addresses these three limitations of the exceptional approach. In particular, rather than stopping when evaluation encounters an expression hole, evaluation continues "around" the hole so that dynamic feedback about other parts of the program remains available. Uniquely, the system tracks the closure (i.e. variable environment) around each expression hole instance as evaluation proceeds.

Hole closure tracking serves several purposes. First, when the programmer performs an edit that fills a hole, evaluation can resume from the paused, i.e. *indeterminate*, evaluation state, rather than restarting evaluation from the beginning; we call this operation *fill-and-resume*. Second, the live programming environment can feed relevant information from the hole closures to the programmer, via various editor services, as they work to fill the holes in the program.

LIVE Talk Proposal. We will demonstrate the basics of our approach as well as two examples of live, hole-driven editor services that we are integrating into Hazel, a live functional programming environment being developed by Omar et al. [2017b] (more background on Hazel below).

The first editor service is Hazel's *live context inspector* (described in Sec. 2), which displays static information together with relevant hole closure information. Programmers can rapidly switch between different closures for the selected hole and interactively explore nested hole closures, which arise from incomplete recursive functions.

The second editor service is Hazel's *live palettes* (described using mockups in Sec. 3), which are user-defined direct manipulation user interfaces that appear within Hazel programs and generate hole fillings. A palette has access to the environment captured by associated hole closures, so it can provide concrete feedback about the dynamic implications of choices made by the programmer in the palette UI. Palettes may themselves contain holes, which may be filled using nested palettes.

The live context inspector has been implemented already, and we expect an initial version of live palettes to be ready for demonstration at the workshop.

2 LIVE CONTEXT INSPECTOR IN HAZEL

This section gives an example-driven overview of our approach as implemented in Hazel, a live programming environment being developed by Omar et al. [2017b]. The Hazel user interface is based roughly on IPython/Jupyter [Pérez and Granger 2007], with results appearing below code

[1] Without this flag, holes cause compilation to fail with an error message that reports information about each hole's type and typing context. Proof assistants like Agda [Norell 2007, 2009] and Idris [Brady 2013] also respond to holes in this way.

```
-- An incomplete quicksort function
qsort : List(int) → List(int)
qsort [] = []
qsort (pivot :: xs) =
  let (smaller, bigger) = partition ((<) pivot) xs
  let (r_smaller, r_bigger) = (qsort smaller, qsort bigger)
  1

-- A simple test to help us explore
qsort [4, 2, 6, 5, 3, 1, 7]
```

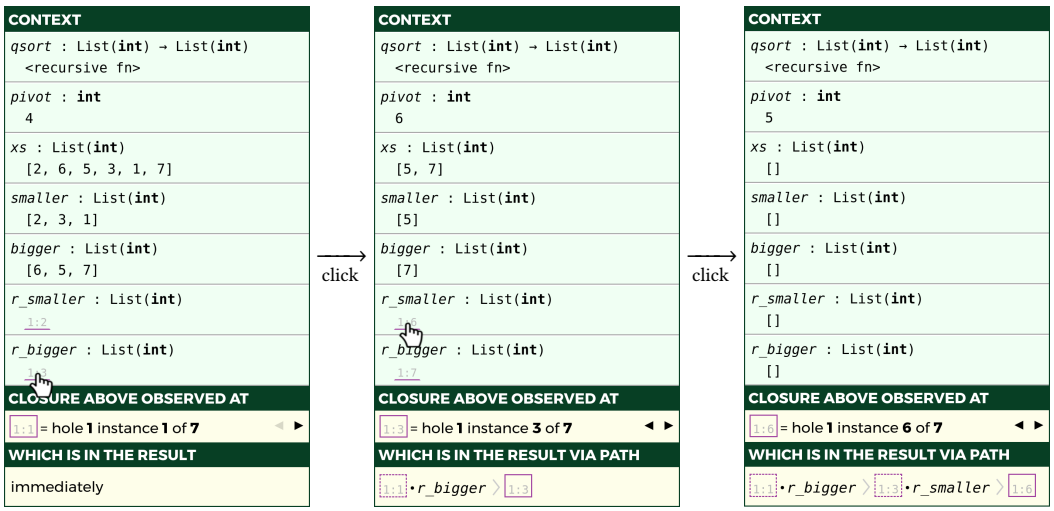
RESULT OF TYPE: `List(int)`

1:1

(a) The result of evaluation is a hole closure.

EXPECTING AN EXPRESSION OF TYPE
`List(int)`
GOT CONSISTENT TYPE
?2

(b) The type inspector provides static feedback about the term at the cursor. Currently, the hole should be filled by a list expression. Holes have the hole (i.e. unknown) type, ?, which is universally consistent (see [Omar et al. 2017a; Siek and Taha 2006]).



(c) The programmer can explore the recursive structure of the computation by clicking on hole instances.

Fig. 1. Example 1: Incomplete Quicksort

cells. The Hazel language is tracking toward feature parity with Elm (elm-lang.org) [Czaplicki 2012, 2018b], a popular pure functional programming language similar to "core ML", with which we assume basic familiarity. Hazel is intended initially for use by students and instructors in introductory functional programming courses (where Elm has been successful [D'Alves et al. 2017]). For the sake of exposition, we have post-processed the screenshots in this section after generating them in Hazel to make use of some "syntactic and semantic sugar" from Elm that was not available in Hazel as of this writing, e.g. pattern matching in function arguments, list notation, and record labels (currently there are only tuples). These conveniences are orthogonal to the focus of this paper; all of the user interface features demonstrated in this section have been implemented and will be demonstrated in action during the workshop presentation.

2.1 Example 1: Incomplete Quicksort

For our first example, consider the incomplete implementation of the recursive quicksort function in Fig. 1a. So far, the programmer (perhaps a student, or a lecturer using Hazel as a presentational aid [Paxton 2002]) has filled in the base case, and in the recursive case, partitioned the remainder of the list and recursed. A hole, numbered 1, appears in return position as the programmer decides how to fill it with an appropriate list expression, as indicated by the *type inspector* in Fig. 1b.

```
-- A less incomplete quicksort function, now with a type error
qsort : List(int) → List(int)
qsort [] = []
qsort (pivot :: xs) =
  let (smaller, bigger) = partition ((<) pivot) xs
  let (r_smaller, r_bigger) = (qsort smaller, qsort bigger)
  r_smaller @ pivot @ r_bigger

-- A simple test to help us explore
qsort [4, 2, 6, 5, 3, 1, 7]
```

EXPECTING AN EXPRESSION OF TYPE
`List(int)`
GOT INCONSISTENT TYPE
`int`

RESULT OF TYPE: `List(int)`

```
(([] @ 1 @ []) @ 2 @ ([] @ 3 @ [])) @ 4 @ (([] @ 5 @ []) @ 6 @ ([] @ 7 @ []))
```

Fig. 2. Example 2: Ill-Typed Quicksort

At the bottom of the cell in Fig. 1a, the programmer has applied `qsort` to an example list. However, the indeterminate result of this function application is simply an instance of hole `1`, which serves only to confirm that evaluation went through the recursive case of `qsort`. More interesting is the live context inspector, shown in three states in Fig. 1c, which provides feedback about the values of the variables in scope at hole `1` from the the various instances of hole `1` that appear in the result, either immediately or within another closure. For example, in its initial state (Fig. 1c, left) it shows the closure at the instance of hole `1` that appears immediately in the result due to the outermost application of `qsort`. From this, the programmer can confirm (or the lecturer can visually point out) that the lists `smaller` and `bigger` computed by the call to `partition` are appropriately named, and observe that they are not yet themselves sorted.

The results from the subsequent recursive calls, `r_smaller` and `r_bigger`, are again hole instances, `1:2` and `1:3`. The programmer can click on either of these hole instances to reveal the associated closures from the corresponding recursive calls. For example, clicking on `1:3` reveals the hole closure from the `r_bigger` recursive call as shown in Fig. 1c (middle). From there, the programmer can click another hole closure, e.g. `1:6` to reveal the hole closure from the subsequent `r_smaller` call as shown in Fig. 1c (right). Notice in each case that the path from the result to the selected hole closure is reported as shown at the bottom of the context inspector in Fig. 1c. In exploring these paths rooted at the result, the programmer can develop concrete intuitions about the recursive structure of the computation even before the program is complete. Note the distinction between this hole closure approach and tracing approaches, which record the environment at every call of a function (as in Lamdu [Lotem and Chuchem 2016]). Tracing approaches do not situate each recorded environment within the result. An interactive debugger might situate recorded environments within an evaluation trace, but this is a more complex object than the result (a single irreducible expression).

2.2 Example 2: Ill-Typed Quicksort

The previous example was incomplete because of *missing* expressions. Now, we discuss programs that are incomplete, and therefore conventionally meaningless, because of *type inconsistencies*. In particular, let us assume that the programmer has filled in hole `1` in the previous example as shown in Fig. 2. The result computed in Fig. 1a recorded the environment around every instance of hole `1`, so the new result can be computed from the previous result by performing contextual substitution (a concept from contextual modal type theory [Nanevski et al. 2008]) and resuming evaluation, rather than restarting evaluation entirely. For larger examples, this *fill-and-resume* operation could significantly reduce evaluation time, which is important to perceived liveness [Tanimoto 2013].

The programmer appears to be on the right track conceptually in recognizing that the pivot needs to appear between the smaller and bigger elements. However, the types do not quite work

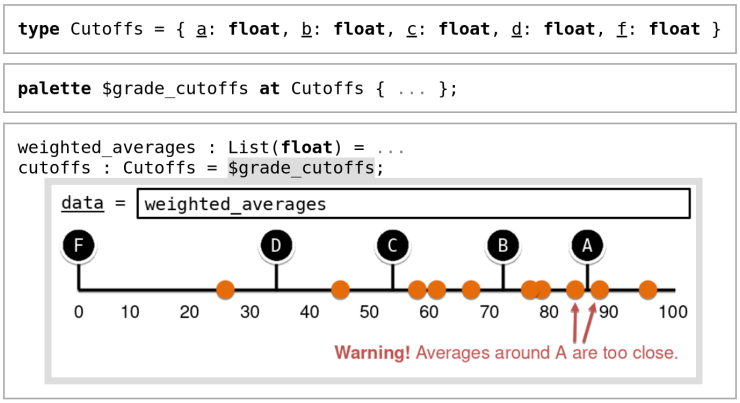


Fig. 3. An example where the teacher assigns cutoffs for letter grades using a domain-specific live palette.

out: the `@` operator here performs list concatenation, but the pivot is an integer. Most compilers and editors will report a static error message to the programmer in this case, and Hazel follows suit in the type inspector (shown inset in Fig. 2). However, in our approach, the presence of a static type error need not cause all feedback about the dynamic behavior of the program to "flicker out" or "go stale"–after all, there are perfectly meaningful parts of the program (both nearby and far away from the error) whose dynamic behavior may be of interest. Concrete values from evaluation can also help explain the type error [Seidel et al. 2016].

Our approach, following the prior work of Omar et al. [2017a], is to semantically internalize the "red outline" around a type inconsistency, representing it as a *non-empty hole*. Evaluation safely proceeds past a non-empty hole just as if it were an empty hole. Evaluation also proceeds inside the hole, so that feedback about the type-inconsistent expression, which might "almost" be correct, is available. In this case, the result at the bottom of Fig. 2 reveals concretely that the programmer is on the right track: the list elements appear in the correct order. They simply have not been combined correctly. The semantics also associates an environment with each instance of a non-empty hole, so we can use the live context inspector essentially as in Fig. 1c (not shown).

3 LIVE PALETTES IN HAZEL

Another editor service that we are designing for integration into Hazel is the *live palette* service. Palettes, which were introduced in work on *active code completion* by Omar et al. [2012], allow programmers to fill expression holes by directly manipulating specialized graphical user interfaces rather than only with text edits. For example, a palette might allow the programmer to fill a hole of type `Color` using a color selection user interface. Omar et al. [2012] elicited several examples of data types where alternative GUIs for constructing values might be useful. The system in that paper, called Graphite, is designed to be extensible by library providers.

Palettes in Graphite are ephemeral, operating as a form of code completion, but projectional editors, e.g. Barista [Ko and Myers 2006] and `mbeddr` [Voelter et al. 2012], often support a small fixed number of palettes (called *projections*) that persist, i.e. that appear within the code itself. Projectional palettes also improve upon Graphite's palettes in that they are compositional, i.e. code can appear within holes that appear inside the palette (e.g. the entries in a matrix palette).

Like Graphite, our current design for Hazel's palettes is extensible. Like projectional palettes, Hazel's palettes are also persistent and compositional. Holes that appear inside palettes are governed by a hygiene discipline based on recent work on splicing in user-defined literal macros [Omar and Aldrich 2018]. Uniquely, Hazel's palette system is live: the program can be evaluated before the

palette has generated code and the palette can use the hole closures associated with the hole that it is filling to provide concrete feedback to the programmer within the palette UI.

Let us consider an illustrative domain-specific example that demonstrates all of these qualities: Fig. 3 shows a mockup of a palette that allows the teacher to determine grade cutoffs, represented collectively by a value of the record type `Cutoffs`, by dragging markers visually along a number line, with student grades superimposed. The grades (in this case `weighted_averages`) are entered by filling a hole in the UI. The program is initially evaluated as if the hole where the palette name appears, here `$grade_cutoffs`, is empty so that any hole closures in the result can be made available to the palette. In this case, there is only one hole closure, from which the actual value of the variable `weighted_averages` can be used by the palette to display the grades as orange points. It can also use this information in other ways, e.g. to display a warning in red that the "A" cutoff does not have a very large gap around it. If there were multiple closures available, the user could toggle between them using the live palette inspector as discussed in the previous section.

We omit the implementation of `$grade_cutoffs` for concision, but to briefly summarize: the implementation of each palette is required to implement a Model-Update-View interface, following the Elm architecture [Czaplicki 2018a]. In the `view` function that generates the user interface as a value of type `Html`, the palette may request that a nested hole with a specified expected type be rendered using the `HtmlHole` constructor of the `Html` type. When the cursor is in this hole, Hazel provides all of the usual editor services based on the specified type. The expressions in these holes can be evaluated relative to the current user-selected closure and the resulting values are available for use by the `view` function (here, to generate the orange dots and red warning). Each palette must also implement a `to_exp` function that produces the final expression to fill the hole, much like a macro (see [Omar and Aldrich 2018]).

4 DISCUSSION

In summary, Hazel provides a full solution to the gap problem: all of Hazel's editor services operate free of temporal or perceptive gaps, because every editor state has non-trivial static and dynamic meaning. In particular, Hazel can evaluate programs with holes and track the closure around each hole instance. The editor services that make use of these hole closures, in turn, provide Hazel programmers with a uniquely *live and direct* typed functional programming experience.

In a separate technical paper, we have developed a formal semantics, mechanized in the Agda proof assistant, that specifies how to evaluate programs with holes while tracking hole closures, and that assigns formal meaning to the fill-and-resume operation. The static semantics is based on the Hazelnut calculus developed recently by Omar et al. [2017a], and the dynamic semantics combines and extends previous work on gradual type theory (to handle type holes) [Siek and Taha 2006; Siek et al. 2015] and contextual modal type theory (which provides a Curry-Howard interpretation of hole closures) [Nanevski et al. 2008].

We believe that the insights we have developed in the setting of Hazel will be immediately relevant to other live functional programming environments, and with further development, perhaps also to imperative programming environments (though certain features, like fill-and-resume, rely on the fact that order of evaluation does not matter in the pure functional setting).

Going forward, we hope to scale up Hazel's live programming features to encompass the full Elm programming language (and perhaps further extensions), and to fully realize our work on live palettes, which we outlined with mockups in Sec. 3. Other editor services might also be able to make use of hole closures, e.g. type-and-example based program synthesis techniques [Osera and Zdancewic 2015] could allow examples to be given at hole instances rather than only at function boundaries. Ultimately, we believe that a principled treatment of incomplete programs will be essential to the operation of the next generation of live programming environments.

REFERENCES

Alfred V. Aho and Thomas G. Peterson. 1972. A Minimum Distance Error-Correcting Parser for Context-Free Languages. *SIAM J. Comput.* 1, 4 (1972), 305–312. https://doi.org/10.1137/0201022

Luís Eduardo de Souza Amorim, Sebastian Erdweg, Guido Wachsmuth, and Eelco Visser. 2016. Principled Syntactic Code Completion Using Placeholders. In *ACM SIGPLAN International Conference on Software Language Engineering (SLE)*.

Edwin Brady. 2013. Idris, A General-Purpose Dependently Typed Programming Language: Design and Implementation. *Journal of Functional Programming* 23, 05 (2013), 552–593.

Sebastian Burckhardt, Manuel Fähndrich, Peli de Halleux, Sean McDirmid, Michal Moskal, Nikolai Tillmann, and Jun Kato. 2013. It's alive! continuous feedback in UI programming. In *Conference on Programming Language Design and Implementation (PLDI)*.

Philippe Charles. 1991. *A Practical Method for Constructing Efficient LALR(K) Parsers with Automatic Error Recovery.* Ph.D. Dissertation. New York, NY, USA. UMI Order No. GAX91-34651.

Evan Czaplicki. 2012. Elm: Concurrent frp for functional guis. *Senior thesis, Harvard University* (2012).

Evan Czaplicki. 2018a. Elm Architecture. (2018). https://guide.elm-lang.org/architecture/. Retrieved Aug 14, 2018.

Evan Czaplicki. 2018b. An Introduction to Elm. (2018). https://guide.elm-lang.org/. Retrieved Apr. 7, 2018.

Curtis D'Alves, Tanya Bouman, Christopher Schankula, Jenell Hogg, Levin Noronha, Emily Horsman, Rumsha Siddiqui, and Christopher Kumar Anand. 2017. Using Elm to Introduce Algebraic Thinking to K-8 Students. In *TFPIE@TFP (EPTCS)*, Vol. 270. 18–36.

Simon Peyton Jones, Sean Leather, and Thijs Alkemade. 2014. Language options — Glasgow Haskell Compiler 8.4.1 User's Guide (Typed Holes). http://downloads.haskell.org/~ghc/latest/docs/html/users_guide/glasgow_exts.html. Retrieved Apr 16, 2018.. (2014).

Lennart C. L. Kats, Maartje de Jonge, Emma Nilsson-Nyman, and Eelco Visser. 2009. Providing rapid feedback in generated modular language environments: adding error recovery to scannerless generalized-LR parsing. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

Andrew J. Ko and Brad A. Myers. 2006. Barista: An Implementation Framework for Enabling New Tools, Interaction Techniques and Views in Code Editors. In *SIGCHI Conference on Human Factors in Computing Systems (CHI)* (2006).

Eyal Lotem and Yair Chuchem. 2016. Project Lamdu. http://www.lamdu.org/. (2016). Accessed: 2016-04-08.

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual modal type theory. *ACM Trans. Comput. Log.* 9, 3 (2008). https://doi.org/10.1145/1352582.1352591

Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory.* Ph.D. Dissertation. Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden.

Ulf Norell. 2009. Dependently typed programming in Agda. In *Advanced Functional Programming*. Springer, 230–266.

Cyrus Omar and Jonathan Aldrich. 2018. Reasonably Programmable Literal Notation. *Proc. ACM Program. Lang.* 2, ICFP (2018), 106:1–106:32. https://doi.org/10.1145/3236801

Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew Hammer. 2017a. Hazelnut: A Bidirectionally Typed Structure Editor Calculus. In *Principles of Programming Languages (POPL)*.

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

Cyrus Omar, YoungSeok Yoon, Thomas D. LaToza, and Brad A. Myers. 2012. Active Code Completion. In *International Conference on Software Engineering (ICSE)*.

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

John Paxton. 2002. Live Programming As a Lecture Technique. *J. Comput. Sci. Coll.* 18, 2 (Dec. 2002), 51–56. http://dl.acm.org/citation.cfm?id=771322.771332

Fernando Pérez and Brian E. Granger. 2007. IPython: a System for Interactive Scientific Computing. *Computing in Science and Engineering* 9, 3 (May 2007), 21–29. http://ipython.org

Eric L. Seidel, Ranjit Jhala, and Westley Weimer. 2016. Dynamic Witnesses for Static Type Errors (or, Ill-typed Programs Usually Go Wrong). In *International Conference on Functional Programming (ICFP)*.

Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*. http://scheme2006.cs.uchicago.edu/13-siek.pdf

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*. 274–293. https://doi.org/10.4230/LIPIcs.SNAPL.2015.274

Steven L. Tanimoto. 2013. A perspective on the evolution of live programming. In *International Workshop on Live Programming (LIVE)*.

Markus Voelter, Daniel Ratiu, Bernhard Schaetz, and Bernd Kolb. 2012. Mbeddr: An Extensible C-based Programming Language and IDE for Embedded Systems. In *SPLASH* (2012).