Toward Semantic Foundations for Program Editors*

Cyrus Omar[1], Ian Voysey[2], Michael Hilton[3], Joshua Sunshine[4], Claire Le Goues[5], Jonathan Aldrich[6], and Matthew A. Hammer[7]

1 **Carnegie Mellon University, Pittsburgh, PA, USA**
`iev@cs.cmu.edu`
2 **Carnegie Mellon University, Pittsburgh, PA, USA**
`iev@cs.cmu.edu`
3 **Oregon State University, Corvallis, OR, USA**
`hiltonm@eecs.oregonstate.edu`
4 **Carnegie Mellon University, Pittsburgh, PA, USA**
`sunshine@cs.cmu.edu`
5 **Carnegie Mellon University, Pittsburgh, PA, USA**
`clegoues@cs.cmu.edu`
6 **Carnegie Mellon University, Pittsburgh, PA, USA**
`aldrich@cs.cmu.edu`
7 **University of Colorado Boulder, Boulder, CO, USA**
`matthew.hammer@colorado.edu`

Abstract

Programming language definitions assign formal meaning to complete programs. Programmers, however, spend a substantial amount of time interacting with *incomplete* programs – programs with holes, type inconsistencies and binding inconsistencies – using tools like program editors and live programming environments (which interleave editing and evaluation). Semanticists have done comparatively little to formally characterize (1) the static and dynamic semantics of incomplete programs; (2) the actions available to programmers as they edit and inspect incomplete programs; and (3) the behavior of editor services that suggest likely edit actions to the programmer.

This paper serves as a vision statement for a research program that seeks to develop these "missing" semantic foundations. Our hope is that these contributions, which will take the form of a series of simple formal calculi equipped with a tractable metatheory, will guide the design of a variety of current and future interactive programming tools, much as various lambda calculi have guided modern language designs. Our own research will apply these principles in the design of Hazel, an experimental *live lab notebook* programming environment designed for data science tasks. We plan to co-design the Hazel language with the editor so that we can explore concepts such as edit-time semantic conflict resolution mechanisms and mechanisms that allow library providers to install library-specific editor services.

1998 ACM Subject Classification D.3.1 Formal Definitions and Theory, D.2.6 Programming Environments

Keywords and phrases program editors; type systems; live programming; program prediction

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2017.11

* This work is supported in part through a gift from Mozilla; by NSF under grant numbers CCF-1619282, 1553741 and 1439957; by AFRL and DARPA under agreement #FA8750-16-2-0042; and by the NSA under lablet contract #H98230-14-C-0140. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of Mozilla, NSF, AFRL, DARPA or NSA.

1 Introduction

Language-aware program editors (like Eclipse or Emacs, with the appropriate extensions installed [13]) offer programmers a number of useful editor services. Simple examples include (1) syntax highlighting, (2) type inspection, (3) navigation to variable binding sites, and (4) refactoring services. More sophisticated editors provide context-aware code and action suggestions to the programmer (using various code completion, program synthesis and program repair techniques). Many editors also offer *live programming* [26, 5] services, e.g. by displaying the run-time value of an expression directly within the editor as the program runs.

When these editor services encounter *complete programs* – programs that are well-formed and semantically meaningful (i.e. assigned meaning) according to the definition of the language in use – they can rely on a variety of well-understood reasoning principles and program manipulation techniques. For example, a syntax highlighter for well-formed programs can be generated automatically from a context-free grammar [47] and the remaining editor services enumerated above can follow the language's type and binding structure as specified by a standard static semantics. Live programming services can additionally follow the language's dynamic semantics.

The problem, of course, is that many of the edit states encountered by a program editor do not correspond to complete programs. For example, the programmer may be in the midst of a transient edit, or the programmer may have introduced a type error somewhere in the program. Standard language definitions are silent about incomplete programs, so in these situations, simple program editors disable various editor services until the program is again in a complete state. In other words, useful editor services become unavailable when the programmer needs them most! More advanced editors attempt to continue to provide editor services during these incomplete states by using various *ad hoc* and poorly understood heuristics that rely on idiosyncratic internal representations of incomplete programs.

This paper advocates for a research program that seeks to understand both incomplete programs, and the editor services that interact with them, as semantically rich mathematical objects. This research program will broaden the scope of the "programming language theory" (PLT) tradition, which has made significant advances by treating complete programs, programming languages and logics as semantically rich mathematical objects.

In following the PLT tradition, we intend to start by developing a series of minimal calculi that build upon well-understood typed lambda calculi to capture the essential character of incomplete programs and various editor services of interest. Editor designers will be able to apply the insights gained from studying these calculi (together with insights gained from the study of human factors and other topics) to design more sophisticated program editors. Some of these editors will evolve directly from editors already in use today. In parallel with these efforts, we plan to design a "clean-slate" programming environment, Hazel, based directly on these first principles. This will allow researchers to explore the frontier of what is possible when one considers languages and editors within a common theoretical framework. Such a clean-slate design will also likely prove useful in certain educational settings, and even some day evolve into a practical tool.

Figure 1 shows a mockup of the Hazel user interface, which is loosely modeled after the widely adopted IPython / Jupyter lab notebook interface [36]. This figure will serve as our running example throughout the remainder of the paper. Each section below briefly summarizes a fundamental problem that we must confront as we seek to develop a semantic foundation for advanced program editors. For each problem, we discuss existing approaches, including those advanced by our own recent research, and suggest a number of promising future research directions that we hope that the community will pursue.

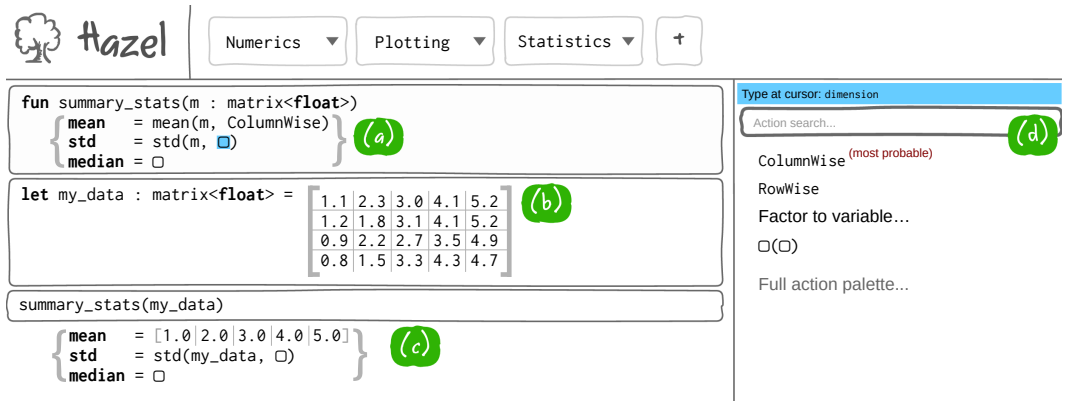

Figure 1 A mockup of Hazel.

2 Problem 1: Syntactically Malformed Edit States

Textual program editors frequently encounter edit states that are not well-formed with respect to the textual syntax of complete programs. For example, consider a programmer constructing a call to a function `std`:

```
std(m,
```

There is a syntax error, so editor services that require a syntactically complete program must be disabled. This is unsatisfying.

Sophisticated editors like Eclipse, and editor generators like Spoofax [20], use *error recovery* heuristics that silently insert tokens so that the editor-internal representation is well-formed [1, 7, 14, 19]. These heuristics are typically provided manually by the grammar designer, though certain heuristics can be generated semi-automatically by tools that are given a description of the scoping conventions of the language or of secondary notational conventions (e.g. whitespace) [19, 12]. Error recovery heuristics require guessing at the programmer's intent, so they are fundamentally *ad hoc* and can confuse the programmer [19].

A more systematic alternative approach, and the approach that we plan to explore with Hazel, is to build a *structure editor* – a program editor where every edit state maps onto a syntax tree, with *holes* representing leaves of the tree that have not yet been constructed. This representation choice sidesteps the problem of syntactically malformed edit states. Notice that in Figure 1, the program fragment in cell **(a)** contains holes, appearing as squares. This design also permits non-textual *projections* of expressions, e.g. the 2D projection of a matrix value in cell **(b)**. We will return to the topic of non-textual projections below.

Structure editors have a long history. For example, the Cornell Program Synthesizer was developed in the early 1980s [45]. Although text-based syntax continues to predominate, there remains significant interest in structure editors today, particularly in practice. For example, Scratch is a structure editor that has achieved success as a tool for teaching children how to program [40]. `mbeddr` is an editor for a C-like language [51], built using the commercially supported MPS structure editor workbench [50]. TouchDevelop is an editor for an object-oriented language [46]. Lamdu [24] and Unison [8] are open source structure editors for functional languages similar to Haskell. Most work on structure editors has focused on the user interfaces that they present. This is important work – presenting a fluid user interface involving higher-level edit actions is a non-trivial problem, and some aspects of this problem remain open even after many years of research. There is reason to be optimistic, however,

$$\begin{array}{lrcl} \mathsf{HTyp}: & \dot{\tau} & ::= & \dot{\tau} \to \dot{\tau} \mid \texttt{num} \mid \text{⦇⦈} \\ \mathsf{HExp}: & \dot{e} & ::= & x \mid \lambda x.\dot{e} \mid \dot{e}(\dot{e}) \mid \underline{n} \mid \dot{e} + \dot{e} \mid (\dot{e} : \dot{\tau}) \mid \text{⦇⦈} \mid \text{⦇}\dot{e}\text{⦈} \end{array}$$

Figure 2 Syntax of H-types and H-expressions in the Hazelnut calculus [34].

with recent studies suggesting that programmers experienced with a modern keyboard-driven structure editor (e.g. `mbeddr`) can be highly productive [3, 52].

Researchers have also explored various "hybrid" approaches, which incorporate holes into an otherwise textual program editor. These hybrid approaches are appealing in part because tools for interacting with text, like regular expressions and various differencing techniques used by version control systems, are already well-developed. For example, recent work on *syntactic placeholders* envisions a textual program editor where edit actions cause textual placeholders (a.k.a. holes) of various sorts to appear, rather than leaving the program transiently malformed [2]. This "approximates" the experience of a structure editor in common usage, while allowing the programmer to perform arbitrary text edits when necessary. Some programming systems, e.g. recent iterations of the Glasgow Haskell Compiler (GHC) [42] and the Agda proof assistant [32], support a workflow where the programmer places holes manually at locations in the program that remain under construction. Another hybrid approach would be to perform error recovery by attempting to insert holes into the internal representation used by the program editor, without including them in the surface syntax exposed to programmers. If "pure" structure editing proves too rigid as we design Hazel, we will explore hybrid approaches.

3 Problem 2: Statically Meaningless Edit States

No matter how an editor confronts syntactically malformed edit states, it must also confront edit states that are syntactically well-formed but statically meaningless. For example, the following value member definition (assuming an ML-like language) has a type inconsistency:

```
val x : float = std(m, ColumnWise)
```

because `std` has type `matrix(float) * dimension -> vec(float)`, but the type annotation on `x` is `float`, rather than `vec(float)`. This leaves the entire surrounding program formally meaningless according to a standard static semantics.

In the presence of syntactic holes, the problem of reasoning statically about incomplete programs becomes even more interesting. Consider the incomplete expression `std(m, ☐)` from cell **(a)** in Figure 1. Although it is intuitively apparent that the type of this expression, after hole instantiation, could only be `vec(float)` (the return type of `std`), and that the hole must be instantiated with values of type `dimension`, the static semantics of complete expressions is again silent about these matters.

Various heuristic approaches are implemented in Eclipse and other sophisticated tools, but the formal character of these heuristics are obscure, buried deep within their implementations. What is needed is a clear static semantics for incomplete programs, i.e. programs that contain holes (in both expressions and types), type inconsistencies, binding inconsistencies (i.e. unbound variables), and other static problems. Such a static semantics is necessary for Hazel to be able to provide type inspection services. For example, in the right column of Figure 1, Hazel is informing the programmer that the expression at the cursor, highlighted in blue in cell **(a)**, must be of type `dimension`). Similarly, Hazel must be able to assign the incomplete function `summary_stats` an incomplete function type for it to be able to

understand subsequent applications of `summary_stats`. Here, the function body has been filled out enough to be able to assign the function the following incomplete function type:

```
matrix(float) -> { mean :  vec(float), std :  vec(float), median : □ }
```

We have investigated a subset of this problem in recent work [34] by defining a static semantics for a simply typed lambda calculus (with a single base type, `num`, for simplicity) extended with holes and type inconsistencies (but no binding inconsistencies). Figure 2 defines the syntactic objects of this calculus – *H-types*, $\dot{\tau}$, are types with holes (⦇⦈), and *H-expressions*, $\dot{e}$, are expressions with holes (⦇⦈), and marked type inconsistencies, (⦇$\dot{e}$⦈). We call marked type inconsistencies *non-empty holes*, because they mark portions of the syntax tree that remain incomplete and behave semantically much like empty holes. Types and expressions that contain no holes are *complete types* and *complete expressions*, respectively.

We will not reproduce further details here. Instead, let us simply note some interesting connections with other work.

First, type holes behave much like unknown types, ?, from Siek and Taha's pioneering work on gradual typing [41]. This discovery is quite encouraging, given that gradual typing is also motivated by a desire to make sense of one class of "incomplete program" – programs that have not been fully annotated with types.

Empty expression holes have also been studied formally, e.g. as the *metavariables* of contextual modal type theory (CMTT) [31]. In particular, expression holes can have types and are surrounded by contexts, just as metavariables in CMTT are associated with types and contexts. This begins to clarify the logical meaning of a typing derivation in Hazelnut – it conveys well-typedness relative to an (implicit) modal context that extracts each expression hole's type and context. The modal context must be emptied – i.e. the expression holes must be instantiated with expressions of the proper type in the proper context – before the expression can be considered complete. This relates to the notion of modal necessity in contextual modal logic.

For interactive proof assistants that support a tactic model based directly on hole filling, the connection to CMTT and similar systems is quite salient. For example, Beluga [37] is based on dependent CMTT and aspects of Idris' editor support [4] are based on a similar system – McBride's OLEG [25]. As we will discuss in Sec. 5, our notion of a program editor supports actions beyond hole filling.

There are a number of future research directions that are worth exploring.

Binding inconsistencies. In the simple calculus developed so far, all variables must be bound before they are used, including those in holes. We plan extend Hazelnut to support reasoning when a variable is mentioned without having been bound (as is a common workflow). Dagenais and Hendren also studied how to reason statically about programs with binding errors using a constraint system, focusing on Java programs whose imports are not completely known [11]. They neither considered programs with holes or other type inconsistencies, nor did they formally specify their technique. However, they provide a useful starting point.

Expressiveness. The simple calculus discussed above is only as expressive as the typed lambda calculus with numbers. We must scale up the semantics to handle other modern language features. Our plan is to focus initially on functional language constructs (so that Hazel can be used to teach courses that are today taught using Standard ML, OCaml or Haskell). This will include recursive and polymorphic functions, recursive types, and labeled product (record) and sum types. We also propose to investigate ML-style structural pattern

matching. All of these will require defining new sorts of holes and static inconsistencies, including: (1) non-empty holes at the type level, to handle kind inconsistencies; (2) holes in label position; and (3) holes and type inconsistencies in patterns.

Automation. Although we plan to explore some of these language extensions "manually," extending our existing mechanized metatheory, we ultimately plan to *automatically* generate a statics for incomplete terms from a standard statics for complete terms, annotated perhaps with additional information. There is some precedent for this in recent work on the Gradualizer, which is capable of producing a gradual type system from a standard type system with lightweight annotations that communicate the intended polarities of certain constructs [10]. However, although it provides a good starting point, gradual type systems only consider the problem of holes in types. Our plan is to build upon existing proof automation techniques, e.g. Agda's reflection [48] (in part because our present mechanization effort is in Agda).

4 Problem 3: Dynamically Meaningless Edit States

Modern programming tools are increasingly moving beyond simple "batch" programming models by incorporating *live programming* features that interleave editing and evaluation [44, 43, 26]. These tools provide programmers with rapid feedback about the dynamic behavior of the program they are editing, or selected portions thereof [27]. Examples include *lab notebooks*, e.g. the popular IPython/Jupyter [36], which allow the programmer to interactively edit and evaluate program fragments organized into a sequence of cells (an extension of the read-eval-print loop (REPL)); spreadsheets; live graphics programming environments, e.g. SuperGlue [26], Sketch-n-Sketch [9] and the tools demonstrated by Bret Victor in his lectures [49]; the TouchDevelop live UI framework [5]; and live visual and auditory dataflow languages [6]. In the words of Burckhardt et al. [5], live programming environments "capture the imagination of today's programmers and promise to narrow the temporal and perceptive gap between program development and code execution".

Our proposed design for Hazel combines aspects of several of these designs to form a *live lab notebook interface*. It will use the edit state of each cell to continuously update the output value displayed for that cell and subsequent cells that depend on it. Uniquely, rather than providing meaningful feedback about the dynamic behavior only once a cell becomes complete, Hazel will provide meaningful feedback also about the dynamic behavior of incomplete cells (and thereby further tighten Burckhardt's "perceptive gap").

For example, in cell **(c)** of Figure 1, the programmer applies the incomplete function `summary_stats` to the matrix `my_data`, and the editor is still able to display a result. The value of the column-wise mean is fully determined, because evaluation does not encounter any holes, whereas the standard deviation and median computations cannot be fully evaluated. Notice, however, that the standard deviation computation does communicate the substitution of the applied argument, `my_data`, for the variable `m`.[1]

To realize this functionality, we need a dynamic semantics for incomplete programs that builds upon our proposed static semantics. There is some precedent for this: research in gradual typing considers the dynamic semantics of programs with holes in types, and our proposed static semantics for incomplete programs borrows technical machinery from

[1] To avoid exposing the internals of imported library functions, evaluation does not step into functions, like `std`, that have been imported from external libraries indicated by the row at the top of Figure 1 (unless specifically requested, not shown).

theoretical work on gradual typing [41]. However, we need a dynamic semantics for incomplete programs that also have expression holes (and in the future, other sorts of holes).

Research on CMTT has not yet considered the problem of evaluating expressions under a non-empty metavariable context. Normally, this would violate the classical notion of Progress – evaluation can neither proceed, nor has it produced a value. We conjecture that this is resolved by (1) positively characterizing *indeterminate* evaluation states, those where a hole blocks progress at all locations within the expression, and (2) defining a notion of Indeterminate Progress that allows for evaluation to stop at an indeterminate evaluation state. By gradualizing CMTT and defining these notions, we believe we can achieve the basic functionality described above.

There are several more applications that we aim to explore after developing these initial foundations. For example, it would be useful for the programmer to be able to select a hole that appears in an indeterminate state and be taken to its original location. There, they should be able to inspect the *value* of a subexpression under the cursor in the environment of the selected hole (rather than just its type). Again, CMTT's closures provide a theoretical starting point for this debugger service.

It would also be useful to be able to continue evaluation where it left off after making an edit to the program that corresponds to hole instantiation. This would require proving a commutativity property regarding hole instantiation. Fortunately, initial research on commutativity properties for holes has been conducted for CMTT, which will serve as a starting point for this work [31]. There are likely to be interesting new theoretical questions (and, likely, some limitations) that arise if one adds non-termination and memory effects.

Relatedly, IPython/Jupyter [36] support a feature whereby numeric variable(s) in cells can be marked as being "interactive", which causes the user interface to display a slider. As the slider value changes, the value of the cell is recomputed. It would be useful to be able to use the mechanisms just proposed to incrementalize parts of this recomputation.

5 Problem 4: A Calculus of Edit Actions

The previous sections considered the structure and meaning of intermediate edit states. However, to understand the act of *editing* itself, we need *a calculus of edit actions* that governs transitions between these edit states.

In a structure editor, the ideal would be for every possible edit state to be both statically and dynamically meaningful according to the semantics proposed in the previous two sections. This corresponds formally to proving a metatheorem about the action semantics: when the initial edit state is semantically meaningful, the edit state that results from performing an action is as well. In a textual or hybrid setting, these structured edit actions would need to be supplemented by lower-level text edit actions that may not maintain this invariant. In addition to this crucial metatheorem, which we call *sensibility*, there are a number of other metatheorems of interest that establish the expressive power of the action semantics, e.g. that every well-typed term can be constructed by some sequence of edit actions.

In our recent work on Hazelnut, we have developed an action calculus for the minimal calculus of H-types and H-expressions described in Section 3 [34]. We have mechanically proven the sensibility invariant, as well as expressivity metatheorems, using the Agda proof assistant. What remains is to investigate *action composition principles*. For example, it would be worthwhile to investigate the notion of an *action macro*, whereby functional programs could themselves be lifted to the level of actions to compute non-trivial compound actions. Such compound actions would give a uniform description of transformations ranging from

the simple – like "move the cursor to the next hole to the right" – to quite complex whole program refactorings, while remaining subject to the core semantics. Using proof automation, it should be possible to prove that an action macro implements derived action logic that is admissible with respect to the core semantics. This would eliminate the possibility of "edit-time" errors. This is closely related to work on tactic languages in proof assistants, e.g. the Mtac typed macro language for Coq [53], differing again in that the action language involves notions other than hole filling.

6 Problem 5: Meaningful Suggestion Generation and Ranking

The simplest edit actions will be bound to keyboard shortcuts. However, Hazel will also provide suggestions to help the programmer edit incomplete programs by providing a *suggestion palette*, marked **(d)** in Figure 1. This palette will suggest semantically relevant code snippets when the cursor is on an empty hole. It will also suggest other relevant edit actions, including high-level edit actions implemented by imported action macros (e.g. the refactoring action in Figure 1). When the cursor is on a non-empty hole, indicating a static error, it will suggest bug fixes. We plan to also consider bugs that do not correspond to static errors, including those identified explicitly by the programmer, and those related to assertion failures or exceptions encountered when using the live programming features of Hazel. In these situations, we plan to build on existing automated fault localization techniques [18, 38, 39].

Note that features like these are not themselves novel. Many editors provide contextually relevant suggestions. Indeed, suggestion generation is closely related to several major research areas: code completion [30, 17], program synthesis [15], and program repair [22, 28, 23, 21].

The problems that such existing systems encounter is exactly the problem we have been discussing throughout this proposal: when attempting to integrate these features into an editor, it is difficult to reason about malformed or meaningless edit states. Many of these systems therefore fall back onto tokenized representations of programs [17]. Because Hazel will maintain the invariant that every edit state is a syntactically and semantically meaningful formal structure, we can develop a more principled solution to the problem of generating meaningful suggestions. In particular, we will be able to *prove* that every action suggestion generated for a particular edit state is meaningful for that edit state.

In addition to investigating the problem of populating the suggestion palette with semantically valid actions, we will consider the problem of evaluating the statistical likelihood of the suggestions. This requires developing a statistical model over actions. We will prove that this statistical model is a proper probability distribution (e.g. that it "integrates" to 1), and that it assigns zero probability to semantically invalid actions. We will also develop techniques for estimating the parameters of these distributions from a corpus of code or a corpus of edit actions. Collectively, we refer to these contributions as a *statistical action suggestion semantics*.

Ultimately, we envision this work as being the foundation for an *intelligent programmer's assistant* that is able to integrate semantic information gathered from the incomplete program with statistics gathered from other programs and interactions that the system has observed to do much of the "tedious" labor of programming, without hiding the generated code from the programmer (as is the case with fully automated program synthesis techniques).

7 Language-Editor Co-Design

In designing Hazel, we are intentionally blurring the line between the programming language and the program editor. This opens up a number of interesting research directions in language-editor co-design. For example, it may be possible to recast "tricky" language mechanisms, like function overloading, type classes [16], implicit values, and unqualified imports, as editor mechanisms. Because we will be treating programming as a structured conversation between the programmer and the programming environment, the editor can simply ask the programmer to resolve ambiguities when they arise. The programmer's choice is then stored unambiguously in the underlying syntax tree.

Another important research direction lies in exploring how types can be used to control the presentation of expressions in the editor. In the textual setting, we have developed *type-specific languages* (TSLs) [33]. It should be possible to define an analagous notion of *type-specific projections* (TSPs) in the setting of a structure editor. For example, the matrix projection shown in Figure 1 need not be built in to Hazel. Instead, the `Numerics` library provider will be able to introduce this logic. In particular, TSPs will define not only derived visual forms, but also derived edit actions (e.g. "add new column" for the example just given.) It should be possible to switch between multiple projections (including purely textual projections) while editing code and interacting with values. This line of research is also related to our work on *active code completion*, which investigated type-specific code generation interfaces in a textual program editor (Eclipse) [35].

Another interesting direction is that of semantic, interactive documentation. In particular, in Hazel, references to program structures that appear in documentation will be treated in the same way as other references and be subject to renaming and other operations. Documentation will also be capable of containing expressions of arbitrary types (e.g. of the `Image` or `Link` type). Together with the type-specific projection mechanism mentioned above, we hope that this will allow Hazel to function not only as a structured programming environment, but also as a structured document authoring environment! By understanding hyperlinks as variable references (in, perhaps, a different modality [29]), we may be able to blur the line between a module and a webpage.

8 Conclusion

To summarize, there are a number of interesting semantic questions that come up in the design of program editors. We advocate a research program that studies these problems using mathematical tools previously used to study programming languages and complete programs. This work will both demystify the design of program editors and open up the doors for a number of advanced editor services. Ultimately, we envision an intelligent programmer's assistant that combines a deep semantic understanding of incomplete programs with a broad statistical understanding of common idioms to help humans author both programs and documents (as one and the same sort of artifact.)

Acknowledgments. We thank the SNAPL 2017 reviewers and our paper shepherd Nate Foster for the thoughtful comments and suggestions.

References

1 Alfred V. Aho and Thomas G. Peterson. A minimum distance error-correcting parser for context-free languages. *SIAM J. Comput.*, 1(4):305–312, 1972. doi:10.1137/0201022.

2 Luís Eduardo de Souza Amorim, Sebastian Erdweg, Guido Wachsmuth, and Eelco Visser. Principled syntactic code completion using placeholders. In *Software Language Engineering (SLE)*, 2016. doi:10.1145/2997364.2997374.

3 Dimitar Asenov and Peter Müller. Envision: A fast and flexible visual code editor with fluid interactions (Overview). In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2014.

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

5 Sebastian Burckhardt, Manuel Fahndrich, Peli de Halleux, Sean McDirmid, Michal Moskal, Nikolai Tillmann, and Jun Kato. It's alive! continuous feedback in UI programming. In *PLDI*, 2013. doi:10.1145/2491956.2462170.

6 Margaret M. Burnett, John W. Atwood Jr., and Zachary T. Welch. Implementing level 4 liveness in declarative visual programming languages. In *IEEE Symposium on Visual Languages*, 1998.

7 Philippe Charles. *A practical method for constructing efficient LALR (K) parsers with automatic error recovery*. PhD thesis, New York University, 1991.

8 Paul Chiusano. Unison. http://www.unisonweb.org/. Accessed: 2016-04-25.

9 Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. Programmatic and direct manipulation, together at last. In *PLDI*, 2016.

10 Matteo Cimini and Jeremy G. Siek. The gradualizer: a methodology and algorithm for generating gradual type systems. In *POPL*, 2016.

11 Barthélémy Dagenais and Laurie J. Hendren. Enabling static analysis for partial Java programs. In *OOPSLA*, 2008.

12 Maartje de Jonge, Emma Nilsson-Nyman, Lennart C. L. Kats, and Eelco Visser. Natural and flexible error recovery for generated parsers. In *Software Language Engineering (SLE)*, 2009.

13 Erich Gamma and Kent Beck. *Contributing to Eclipse: principles, patterns, and plug-ins*. Addison-Wesley Professional, 2004.

14 Susan L. Graham, Charles B. Haley, and William N. Joy. Practical lr error recovery. *ACM SIGPLAN Notices*, 14(8), 1979. doi:10.1145/872732.806967.

15 Sumit Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, PPDP'10, pages 13–24, New York, NY, USA, 2010. ACM. doi:10.1145/1836089.1836091.

16 Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, 1996.

17 Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *International Conference on Software Engineering (ICSE)*, pages 837–847, 2012.

18 James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *International Conference on Software Engineering (ICSE)*, pages 467–477, Orlando, FL, USA, 2002. doi:10.1145/581339.581397.

19 Lennart C. L. Kats, Maartje de Jonge, Emma Nilsson-Nyman, and Eelco Visser. Providing rapid feedback in generated modular language environments: adding error recovery to scannerless generalized-LR parsing. In *OOPSLA*, 2009. doi:10.1145/1640089.1640122.

20 Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *OOPSLA*, 2010.

21 Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. Repairing programs with semantic code search. In *International Conference On Automated Software Engineering (ASE)*, 2015.

22 Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering (TSE)*, 38:54–72, 2012. doi:10.1109/TSE.2011.104.

23 Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *POPL*, 2016. doi:10.1145/2837614.2837617.

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

25 Conor McBride. *Dependently typed functional programs and their proofs*. PhD thesis, University of Edinburgh. College of Science and Engineering. School of Informatics., 2000.

26 Sean McDirmid. Living it up with a live programming language. In *OOPSLA*, 2007.

27 Sean McDirmid. Usable live programming. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, 2013.

28 Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *International Conference on Software Engineering (ICSE)*, 2016.

29 Tom Murphy VII, Karl Crary, and Robert Harper. Type-safe distributed programming with ML5. In *International Symposium on Trustworthy Global Computing*, pages 108–123. Springer, 2007.

30 Kıvanç Muşlu, Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Improving IDE Recommendations by Considering Global Implications of Existing Recommendations. In *New Ideas and Emerging Results Track at the 34th International Conference on Software Engineering (ICSE)*, 2012.

31 Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Trans. Comput. Log.*, 9(3), 2008.

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

33 Cyrus Omar, Darya Kurilova, Ligia Nistor, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Safely composable type-specific languages. In *ECOOP*, 2014.

34 Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew Hammer. Hazelnut: A bidirectionally typed structure editor calculus. In *POPL*, 2017. URL: https://arxiv.org/abs/1607.04180.

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

36 Fernando Pérez and Brian E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007. URL: http://ipython.org.

37 Brigitte Pientka. Beluga: Programming with dependent types, contextual data, and contexts. In *International Symposium on Functional and Logic Programming (FLOPS)*, 2010. doi:10.1007/978-3-642-12251-4_1.

38 Yuhua Qi, Xiaoguang Mao, Yan Lei, and Chengsong Wang. Using automated program repair for evaluating the effectiveness of fault localization techniques. In *International Symposium on Software Testing and Analysis*, 2013.

39 Manos Renieris and Steven Reiss. Fault localization with nearest neighbor queries. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2003.

40 Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: Programming for All. *Commun. ACM*, 52(11):60–67, November 2009. doi:10.1145/1592761.1592779.

41 Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, 2006.

42 Simon Peyton Jones, Sean Leather and Thijs Alkemade. Typed holes in GHC. `https://wiki.haskell.org/GHC/Typed_holes`.

43 Steven L. Tanimoto. VIVA: A visual language for image processing. *J. Vis. Lang. Comput.*, 1(2):127–139, 1990. URL: `http://dx.doi.org/10.1016/S1045-926X(05)80012-6`, `doi:10.1016/S1045-926X(05)80012-6`.

44 Steven L. Tanimoto. A perspective on the evolution of live programming. In *1st International Workshop on Live Programming, (LIVE)*, 2013.

45 Tim Teitelbaum and Thomas Reps. The Cornell Program Synthesizer: A Syntax-directed Programming Environment. *Commun. ACM*, 24(9):563–573, 1981. `doi:10.1145/358746.358755`.

46 Nikolai Tillmann, Michal Moskal, Jonathan de Halleux, and Manuel Fahndrich. TouchDevelop: Programming Cloud-connected Mobile Devices via Touchscreen. In *SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, 2011. `doi:10.1145/2048237.2048245`.

47 Mark G. J. van den Brand, Arie van Deursen, Jan Heering, H. A. de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. The ASF+SDF meta-environment: A component-based language development environment. In *International Conference on Compiler Construction (CC)*, 2001.

48 Paul Van Der Walt and Wouter Swierstra. Engineering proof by reflection in Agda. In *Symposium on Implementation and Application of Functional Languages*, 2012.

49 Bret Victor. Inventing on principle. Invited talk, Canadian University Software Engineering Conference (CUSEC), 2012.

50 Markus Voelter. Language and IDE Modularization and Composition with MPS. In *International Summer School on Generative and Transformational Techniques in Software Engineering*, pages 383–430. Springer, 2011.

51 Markus Voelter, Daniel Ratiu, Bernhard Schaetz, and Bernd Kolb. Mbeddr: An Extensible C-based Programming Language and IDE for Embedded Systems. In *SPLASH*, 2012. `doi:10.1145/2384716.2384767`.

52 Markus Voelter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. Towards User-Friendly Projectional Editors. In *Software Language Engineering (SLE)*, 2014. `doi:10.1007/978-3-319-11245-9_3`.

53 Beta Ziliani, Derek Dreyer, Neelakantan R Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. Mtac: A monad for typed tactic programming in Coq. *Journal of Functional Programming*, 25:e12, 2015.