Toward a Live Stepper for Typed Expressions with Holes

CYRUS OMAR, Carnegie Mellon University
IAN VOYSEY, Carnegie Mellon University
MATTHEW A. HAMMER, University of Colorado Boulder

To understand the dynamic behavior of an expression, programmers can use a "stepper" to interactively simplify the expression according to the dynamic semantics of the programming language. The problem that motivates this work is that a standard dynamic semantics assigns meaning only to complete, well-typed expressions, but there are situations where a programmer might want to explore the dynamic behavior of an expression well before it is complete. This paper proposes the development of a dynamic semantics for *incomplete expressions*, which we take to mean expressions with *holes*. Holes indicate portions of the expression that have yet to be filled in, or, following our recent work (Omar et al. 2017a), that have a local type inconsistency that has yet to be resolved. The result would be a program editor where the programmer has access to the stepper at all times, not just when the program is in a complete state, and where evaluation does not stop immediately at the hole, but rather proceeds as far as possible past the hole.

Even this would not be entirely satisfying when engaging in live programming, where editing and evaluation are interleaved, because naïvely, the programmer would need to restart the stepper after each edit. To address this problem, this paper further proposes a mechanism that tracks the dynamic environment around each hole as well as its evaluation status. This allows for the specification of a *live stepper*, i.e. a stepper where evaluation can continue where it left off even after holes are filled in, because the current state of the stepper can be "patched" to accurately reflect each edit that was made.

This paper reports early conceptual progress in these directions. Many details have yet to be considered. We hope to discuss both the human aspects and the formal aspects of the design with the workshop participants.

ACM Reference format:
Cyrus Omar, Ian Voysey, and Matthew A. Hammer. 2017. Toward a Live Stepper for Typed Expressions with Holes. *PACM Progr. Lang.* 1, 1, Article 1 (January 2017), 7 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

Broadly speaking, live programming environments are those that granularly interleave editing and evaluation (Burckhardt et al. 2013; McDirmid 2007; Tanimoto 1990, 2013). In the words of Burckhardt et al. (2013), live programming environments "promise to narrow the temporal and perceptive gap between program development and code execution". Examples of live programming environments include lab notebook environments, e.g. the popular IPython/Jupyter (Pérez and Granger 2007), 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 like SuperGlue (McDirmid 2007), Sketch-n-Sketch (Chugh et al. 2016) and the tools demonstrated by Bret Victor in his lectures (Victor 2012); the TouchDevelop live UI framework (Burckhardt et al. 2013); live mobile application development systems like Flutter (Flutter Developers 2017); and live visual and auditory dataflow languages (Burnett et al. 1998), to name a few prominent examples.

The problem that has motivated much of our recent work is that most programming environments, live programming environments included, provide feedback via various editor services only once the program being edited is syntactically well-formed and, when relevant, well-typed. This leaves a "temporal and perceptive gap", because programmers sometimes leave a program malformed or

2017. 2475-1421/2017/1-ART1 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

ill-typed for extended periods of time, e.g. as they think about what to enter at the cursor, or as they work on a different part of the program.

In view of this general problem, we recently developed a *structure editor calculus* called Hazelnut where every edit state consists of a well-formed and statically meaningful, i.e. well-typed, incomplete expression, which we take to mean an expression with holes (Omar et al. 2017a). This calculus addressed fundamental questions relevant to editor services that operate statically, but there was no solution in that paper to the problems faced by editor services that also require knowledge of the dynamic meaning of an incomplete program, as would be relevant to a live programming environment. For example, consider a *stepper* (a.k.a. a *stepwise debugger*), like that available to Haskell programmers in the GHCi system (GHC Team 2017) and other systems (Marlow et al. 2007; Wallace et al. 2001), to OCaml programmers with recent work on `ocamli` by Whitington and Ridge (2017) and to Standard ML programmers (Tolmach and Appel 1995). A stepper requires that the expression being stepped be assigned dynamic meaning according to a small-step operational semantics (Harper 2016; Plotkin 2004), but no such semantics was defined for incomplete expressions that arise when using Hazelnut, or any of these other systems. Defining such a semantics was left as future work in the Hazelnut paper, and in a subsequent "vision paper" (Omar et al. 2017b). The purpose of this paper is to sketch out our progress toward a theoretically well-grounded solution.

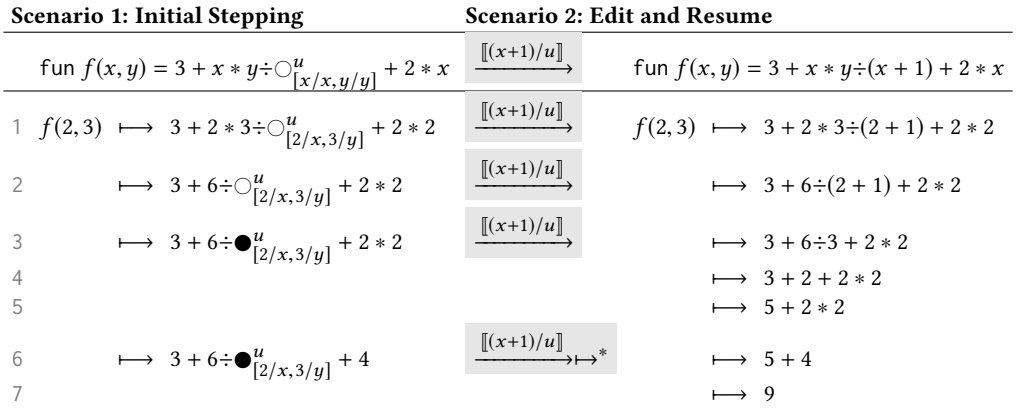

Fig. 1. An example demonstrating (1) stepping of an incomplete program; (2) support for "edit-and-resume" when transitioning between edit states related by an edit that can be understood as hole instantiation.

Scenario 1: Initial Stepping. Consider the definition of the incomplete function f on the top left of Fig. 1. This function is incomplete because a hole, notated $\bigcirc$, appears in its body. In our previous work, holes were unadorned (Omar et al. 2017a).[1] For the purposes of this paper, however, it is helpful to adorn each hole with a unique name, u. In addition, each hole is adorned with an *environment*, indicated by a subscript. We will return to consider environments shortly. Our notation here is for the purposes of exposition. In practice, the hole name might be indicated in some other way, e.g. using different colors for different visible holes, and the environment would be maintained internally, rather than shown explicitly to the programmer.

The cell below the definition of f applies f to 2 and 3 (because the programmer intends to explore the behavior of f for this choice of input.) Normally, the fact that f is incomplete would prevent the programmer from being able to step this function application – the programmer would first need to fill in the hole in f with a well-typed term before the stepper could consult the language's operational semantics to proceed with stepping. This is despite the fact that according to a static

[1] In our previous work, we notated holes (⦇⦈), but $\bigcirc$ is simpler for our present purposes.

semantics for incomplete expressions following that described in our previous work, f can still be assigned type $(\texttt{num}, \texttt{num}) \rightarrow \texttt{num}$ and $f(2,3)$ can therefore be assigned type num (Omar et al. 2017a). Our interest here is in eliminating this restriction, so that we can step the expression $f(2,3)$ even before completing the definition of f. We can do so as shown on Lines 1-3,6 (left) of Fig. 1. Let us consider each step in turn.

The first step (Line 1) operates in essentially the usual way: we substitute 2 for x and 3 for y in the body of f. The only novelty is that substitution proceeds also into the environment associated with each hole. An environment is an n-ary substitution mapping variables, x, to expressions, e, notated in the standard way as $[e_1/x_1, ..., e_n/x_n]$. When stepping starts, the environment associated with each hole is simply the identity environment, here $[x/x, y/y]$, indicating that no substitutions have yet occurred around the hole. Once we apply f, the environment for the hole u becomes $[2/x, 3/y][x/x, y/y] = [2/x, 3/y]$. There are two main reasons to maintain an environment around each hole. One is so that the programmer or an editor service in the live programming environment can select a hole in the evaluation trace and inspect the current environment there, to aid in determining how to fill the hole. The second reason has to do with edit-and-resume functionality, which we will return to in Scenario 2 below.

Assuming the usual associativity and left-to-right evaluation order for arithmetic expressions, the second step of evaluation (Line 2) proceeds to reduce the subexpression $2 * 3$ to 6 in exactly the usual way.

Now we arrive at a critical point in evaluation. The next step, following the usual evaluation order, would divide 6 by the evaluated divisor, except that the divisor is absent – there is a hole, u, in its place.

One approach here would be to throw our hands up and raise an exception at this point, following the common programmer practice of raising an exception named something like `Unimplemented` at points in a program that remain to be written. This is the approach that the hole system in the Glasgow Haskell Compiler takes (Peyton Jones et al. 2016).

This approach is unsatisfying, however, because there is computation that remains to be done in other parts of the program. We'd like to step *past* the hole and evaluate as much as we can elsewhere (at least, in a pure functional setting.) We do so by way of the third step (Line 3), which simply marks the hole as having been evaluated by coloring it in, ●. We will return to why tracking the evaluation status of each hole instance is helpful in Scenario 2 below.

Because the hole appears as a divisor, we cannot reduce the sub-expression $6 \div \bullet^{u}_{[2/x,3/y]}$ any further. That, in turn, also prevents us from reducing the sub-expression $3 + 6 \div \bullet^{u}_{[2/x,3/y]}$ any further. However, we can move on as shown on Line 6 to the sub-expression $2 * 2$, which reduces to 4. At this point, there are no further steps that can be taken. This seems to violate the classical notion of Progress, which is one half of the classical Type Safety theorem (Harper 2016; Milner 1978), in that we claimed that the expression being stepped is well-typed, but evaluation can neither proceed, nor has it produced a value. However, this violation is not due to missing rules in our semantics – there is simply nothing more we can do. We conjecture that this theoretical problem can therefore be solved by (1) positively characterizing *indeterminate* evaluation states, i.e. 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, in addition to a value. This "fix" is in some ways analagous to the fix needed when introducing run-time errors into a language (Harper 2016). A careful evaluation of this conjecture using the Agda proof assistant (Norell 2009) is ongoing work.

Before moving on to Scenario 2, there is one more important issue to consider. In the example in Fig. 1, there were only expression holes, but in the full calculus, there can also be type holes. An

observation made in our previous work was that the machinery for reasoning about type holes coincides with that for reasoning about unknown types in gradual type theory (Siek and Taha 2006). Our conjecture is therefore that the machinery needed to step gradually typed programs will also allow us to step incomplete programs that have type holes. In particular, there will be a need for run-time type checking based on cast insertion, so as to handle well-typed but erroneous programs like $(3 : \bigcirc)(4)$.

Scenario 2: Edit and Resume. Suppose now that the programmer decides that the hole u should be filled by the expression $x + 1$, and, through some sequence of edit actions (considered formally in our prior work), arrives at the new definition of f shown on the top right of Figure 1. This function f is now complete – no holes remain – so the live programming environment could restart evaluation of $f(2, 3)$ and proceed in the usual way by taking the steps shown on the right of Fig. 1, ultimately arriving on Line 7 at the final result of 9.

The problem here is that when live programming, it might not be desirable to restart stepping from the beginning on each such edit, both for reasons of convenience and performance. A programmer particularly interested in some intermediate evaluation state, e.g. any of Lines 1, 2, 3 or 6 on the left of Fig. 1, would have to engage with the stepper to return to the corresponding edit state on the right of Fig. 1 after the edit, and incur the dynamic cost of doing so.

A better design would be one where if the editor has already computed an evaluation state from the version of f with a hole, and two edit states differ only up to *hole instantiation*, written $[\![(x + 1)/u]\!]$, then it can take advantage of an important commutativity property that we aim to prove about our dynamics: that *hole instantiation commutes with evaluation*.

Hole instantiation, $[\![e/u]\!]e'$ is similar to substitution, except that it acts on hole(s) named u in $\dot{e}'$. At each such hole, the corresponding substitution is applied to $\dot{e}$. For example, on Line 1, we replace the hole u on the left with $[2/x, 3/y](x + 1) = 2 + 1$ on the right to arrive at the corresponding evaluation state. The same can be done on Line 2, allowing us to skip Line 1 entirely. On Line 3, hole instantiation operates a bit differently because the hole u has been evaluated – we now instantiate the hole with the evaluated value of $[2/x, 3/y](x + 1)$, which is 3.

More generally, our conjecture is that it suffices to start from any indeterminate evaluation state previously computed and perform hole instantiation on it. After doing so, evaluation can resume. The end result is guaranteed to coincide with that of evaluating the new version of f from scratch. This might require some number of additional steps, as indicated by the $\mapsto^*$ connective on Line 6.

This relates to ongoing research into combining general-purpose incremental computation (IC) with static analysis (Hammer et al. 2016). Currently, IC research focuses on input changes (Chen et al. 2011, 2014; Fisher et al. 2016; Hammer and Acar 2008; Hammer et al. 2009; Hammer and Dunfield 2016; Hammer et al. 2015, 2014, 2011), whereas the mechanism proposed here considers incremental *program changes*.

The notion of holes being associated with unique names and substitutions, and the notion of hole instantiation just described, is borrowed directly from work in *contextual modal type theory (CMTT)* (Nanevski et al. 2008). Hole names correspond to *metavariables* and holes themselves to *closures*. CMTT is, in turn, the Curry-Howard interpretation of contextual modal logic. This gives us confidence that our approach is not *ad hoc*, but rather rooted in the established logical tradition.

Next Steps. The preliminary work outlined above, with its roots in gradual type theory and CMTT, suggests a theoretical foundation for moving forward. However, there remain some major missing pieces, on both the theoretical and implementation sides.

First, CMTT does not come equipped with a dynamic semantics that supports evaluation of terms with free metavariables, which is precisely what we require (see Scenario 1, above). As such, we need to formally develop the notion of an *indeterminate evaluation state*, define a dynamics

that can handle free metavariables, and formally state and prove our Indeterminate Progress and commutativity conjectures. We also need to carefully consider how non-termination affects the commutativity property. Our work so far has focused on pure, functional languages, but it is important also to consider these issues for impure functional languages, e.g. ML-like languages with reference cells and external effects. It is likely that the commutativity property will be weaker in this setting.

Second, CMTT's closures nicely handle empty expression holes, but non-empty holes, type holes, and other problems that we plan to internalize with our statics need to be considered carefully. Non-empty holes can likely be understood as a simple variation on empty holes. In the previous section, we discussed the relationship between type holes and gradual typing. Work in gradual typing appears to provide one solution to the problem of evaluating terms with type holes (by inserting run-time casts (Siek and Taha 2006).) This suggests that a comprehensive dynamics for incomplete programs, i.e. one that assigns dynamic meaning to every statically meaningful incomplete program, will require developing a *gradual contextual modal type theory* (GCMTT).

We also need to develop a semantics that characterizes when two edit states are related by hole instantiation. There are two ways to approach this: as a function of the syntactic difference between the two edit states; and 2) as a function of an edit action that was actually performed.

There are several more practical design questions that we aim to explore after developing the initial foundations just described. First is of course that we need a useful baseline stepper for Hazelnut, with support for skipping tedious intermediate steps and displaying large terms in a readable manner. Recent work by Whitington and Ridge (2017) on `ocamli` has a similar goal, and we plan to follow many of the same techniques. We also plan to look into work by Perera et al. (2012) on using a (slightly different) notion of holes to hide portions of the program that are irrelevant to selected portions of the output. We are calling our ongoing design Hazelnut Live. Ultimately, this will be merged into the full-scale design that we call Hazel (Omar et al. 2017b).

It would be useful for the programmer to be able to select a hole that appears in an indeterminate state and 1) be taken to its original location; 2) be able to inspect the *value* of a subexpression under the cursor in the environment of the selected hole (rather than just its type.) This corresponds to "watches" and "stack inspection" in standard debuggers.

IPython/Jupyter (Pérez and Granger 2007) supports 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 new value of the cell is recomputed. It would be useful to be able to use the mechanisms just described to incrementalize parts of this recomputation automatically.

Ultimately, we believe that this work will provide a foundation for live programming environments for statically typed functional programming languages that behave far less rigidly than one might expect, because holes in both expressions and types do not prevent us from dynamically exploring the program being written.

ACKNOWLEDGMENTS

The authors would like to thank Jonathan Aldrich, Michael Hilton, Roly Perera and the anonymous referee for their thoughtful feedback on this work. The last author is supported in part through gifts from Mozilla and Facebook; 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.

REFERENCES

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 *PLDI*.

Margaret M. Burnett, John W. Atwood Jr., and Zachary T. Welch. 1998. Implementing Level 4 Liveness in Declarative Visual Programming Languages. In *IEEE Symposium on Visual Languages*.

Yan Chen, Joshua Dunfield, Matthew A. Hammer, and Umut A. Acar. 2011. Implicit self-adjusting computation for purely functional programs. In *ICFP*.

Yan Chen, Joshua Dunfield, Matthew A. Hammer, and Umut A. Acar. 2014. Implicit self-adjusting computation for purely functional programs. *J. Funct. Program.* 24, 1 (2014), 56–112.

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

Dakota Fisher, Matthew A. Hammer, William E. Byrd, and Matthew Might. 2016. miniAdapton: A Minimal Implementation of Incremental Computation in Scheme. *CoRR* abs/1609.05337 (2016). http://arxiv.org/abs/1609.05337

Flutter Developers. 2017. Technical Overview - Flutter. (2017). https://flutter.io/technical-overview/. Retrieved Sep. 21, 2017.

The GHC Team. 2017. The GHCi Debugger. https://downloads.haskell.org/~ghc/7.2.1/docs/html/users_guide/ghci-debugger.html. Accessed: Sep. 21, 2017. (2017).

Matthew A. Hammer and Umut A. Acar. 2008. Memory management for self-adjusting computation. In *7th International Symposium on Memory Management (ISMM)*.

Matthew A. Hammer, Umut A. Acar, and Yan Chen. 2009. CEAL: a C-based language for self-adjusting computation. In *PLDI*.

Matthew A Hammer, Bor-Yuh Evan Chang, and David Van Horn. 2016. A Vision for Online Verification-Validation. *International Conference on Generative Programming: Concepts and Experiences (GPCE)* (2016).

Matthew. A. Hammer and Joshua Dunfield. 2016. Typed Adapton: Refinement types for nominal memoization of purely functional incremental programs. *ArXiv e-prints* (Oct. 2016). arXiv:cs.PL/1610.00097

Matthew A. Hammer, Joshua Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S. Foster, Michael W. Hicks, and David Van Horn. 2015. Incremental computation with names. In *OOPSLA*.

Matthew A. Hammer, Yit Phang Khoo, Michael Hicks, and Jeffrey S. Foster. 2014. Adapton: composable, demand-driven incremental computation. In *PLDI*.

Matthew A. Hammer, Georg Neis, Yan Chen, and Umut A. Acar. 2011. Self-adjusting stack machines. In *OOPSLA*.

Robert Harper. 2016. *Practical Foundations for Programming Languages* (2nd ed.). https://www.cs.cmu.edu/~rwh/plbook/2nded.pdf

Simon Marlow, José Iborra, Bernard Pope, and Andy Gill. 2007. A lightweight interactive debugger for Haskell. In *Workshop on Haskell*.

Sean McDirmid. 2007. Living It Up with a Live Programming Language. In *OOPSLA*.

Robin Milner. 1978. A theory of type polymorphism in programming. *Journal of computer and system sciences* 17, 3 (1978), 348–375.

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

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

Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew Hammer. 2017a. Hazelnut: A Bidirectionally Typed Structure Editor Calculus. In *POPL*. https://arxiv.org/abs/1607.04180

Cyrus Omar, Ian Voysey, Michael Hilton, Joshua Sunshine, Claire Le Goues, Jonathan Aldrich, and Matthew Hammer. 2017b. Toward Semantic Foundations for Program Editors. In *Summit on Advances in Programming Languages (SNAPL)*. https://arxiv.org/abs/1607.04180

Roly Perera, Umut A. Acar, James Cheney, and Paul Blain Levy. 2012. Functional programs that explain their work. In *ICFP*.

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

Simon Peyton Jones, Sean Leather, and Thijs Alkemade. 2016. Typed holes in GHC. https://wiki.haskell.org/GHC/Typed_holes. (2016). Accessed: 2016-04-08.

Gordon D. Plotkin. 2004. A structural approach to operational semantics. *J. Log. Algebr. Program.* 60-61 (2004), 17–139.

Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*.

Steven L. Tanimoto. 1990. VIVA: A visual language for image processing. *J. Vis. Lang. Comput.* 1, 2 (1990), 127–139.

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

Andrew P. Tolmach and Andrew W. Appel. 1995. A Debugger for Standard ML. *J. Funct. Program.* 5, 2 (1995), 155–200.

Bret Victor. 2012. Inventing on principle. Invited talk, Canadian University Software Engineering Conference (CUSEC). (2012). https://www.youtube.com/watch?v=PUv66718DII

Malcolm Wallace, Olaf Chitil, Thorsten Brehm, and Colin Runciman. 2001. Multiple-View Tracing for Haskell: a New Hat. In *Haskell Workshop*.

John Whitington and Tom Ridge. 2017. Visualizing the Evaluation of Functional Programs for Debugging. In *SLATE*.