Live Functional Programming with Typed Holes

ANONYMOUS AUTHOR(S)

Live programming environments aim to provide programmers (and sometimes audiences) with continuous feedback about a program's dynamic behavior as it is being edited. The problem is that programming languages typically assign dynamic meaning only to programs that are *complete*, i.e. syntactically well-formed and free of type errors, whereas during much of the editing process, the program is not yet complete. As a result, live feedback flickers in and out or goes stale, in many cases for substantial lengths of time.

This paper confronts this problem from type-theoretic first principles by developing a *dynamic semantics for incomplete functional programs*, based in part on the static semantics for incomplete functional programs developed in recent work by Omar et al. [2017a]. Incomplete functional programs are modeled as expressions with *holes*, with empty holes standing for missing expressions or types, and non-empty holes operating as "membranes" around static and dynamic type inconsistencies. Rather than aborting with an exception when evaluation encounters any of these holes (as in several existing systems), evaluation proceeds "around" the holes, performing as much of the remaining computation as is possible and tracking the closure around each hole instance as it flows through the program. These hole closures can be reported both directly and indirectly, via various editor services, to help the programmer decide how to fill a hole. They also enable a "fill-and-resume" feature that avoids the need to restart evaluation after edits that amount to hole filling. Formally, the semantics draws from both gradual type theory (to handle type holes) and contextual modal type theory (which provides a logical foundation for hole closures), and develops additional technical machinery necessary to continue evaluation past the various hole forms and to prove important metatheoretic properties.

We have mechanized the core formal development using the Agda proof assistant. We also describe a simple implementation, called Hazelnut Live, that inserts holes as necessary during the editing process to guarantee that every edit state has some (possibly incomplete) type, based in part on the Hazelnut edit action calculus in prior work by Omar et al. [2017a]. Taken together with the type safety property that this paper establishes, the result is a proof-of-concept live typed functional programming environment where dynamic feedback is truly continuous, i.e. it is available for every possible edit state.

1 Introduction

Programmers typically shift back and forth between program editing and evaluation many times before converging upon a program that behaves correctly. Live programming environments 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.

For example, read-evaluate-print loops (REPLs) and derivatives thereof, like the IPython/Jupyter lab notebooks popular in data science [Pérez and Granger 2007], allow the programmer to edit and immediately execute program fragments organized into a sequence of cells. Spreadsheets are live functional dataflow environments, with cells organized into a grid [Wakeling 2007]. More specialized examples include live direct manipulation programming environments like SuperGlue [McDirmid 2007], Sketch-n-Sketch [Chugh et al. 2016; Hempel and Chugh 2016], and the tools demonstrated by Victor [2012] in his lectures; live user interface frameworks [Burckhardt et al. 2013]; live image processing languages [Tanimoto 1990]; and live visual and auditory dataflow languages [Burnett et al. 1998], which can support live coding in an artistic performance. Editor-integrated debuggers [McCauley et al. 2008] and other systems that support editing the state of a running program, like Smalltalk environments [Goldberg and Robson 1983], are also live programming environments. Live programming, in its various incarnations [Tanimoto 1990, 2013], has been, and continues to be, an active area of research and development.

2018. 2475-1421/2018/3-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The problem at the heart of this paper 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 in and out", creating temporal gaps, 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 actually seeing and writing in the editor.

In some cases these gaps are momentary, like while the programmer enters a short expression. In other cases, these gaps can persist over substantial lengths of time, such as when there are many branches of a case analysis whose bodies are initially left blank or when the programmer makes a mistake. Novice programmers, of course, make more mistakes [Fitzgerald et al. 2008; McCauley et al. 2008]. The problem is particularly pronounced for languages with rich static type systems, where certain program changes, such as a change to a type definition, can cause type errors to propagate throughout the program. Throughout the process of addressing these errors, the program text remains formally meaningless. Overall, an analysis of edits performed by Java programmers using Eclipse found that about 40% of edit states were malformed [Omar et al. 2017a; Yoon and Myers 2014] and some additional number, which could not be determined from the data, were well-formed but ill-typed.

In recognition of this problem—that incomplete programs lack meaning—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* in both expression and type position. 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). Omar et al. [2017a,b] discuss several ways to determine an incomplete expression from an editor state. Briefly, error recovery mechanisms can insert the necessary holes implicitly [Aho and Peterson 1972; Charles 1991; Graham et al. 1979; Kats et al. 2009; Kats and Visser 2010], the programmer can insert explicit holes manually (see below) or semi-automatically via code completion techniques [Amorim et al. 2016], or a structure editor can insert holes fully automatically [Omar et al. 2017a] (we say more about this technique in Sec. 3.5). For the purposes of live programming, however, a static semantics for incomplete programs does not suffice—we also need a corresponding dynamic semantics for incomplete programs that specifies how to evaluate expressions with holes. This paper aims to address this need by developing a dynamic semantics for incomplete functional programs, starting from the static semantics of Omar et al. [2017a].

One approach would be to define a dynamic semantics that simply aborts with an error when evaluation reaches a hole. This mirrors a workaround to the problem that programmers commonly deploy: raising an exception where an expression has yet be determined, e.g. `raise Unimplemented`. Some languages provide more concise notation for this "encoding" of expression holes, e.g. Scala provides `???` and GHC Haskell with the `-fdefer-typed-holes` or `-fdefer-type-errors` flags uses `_u` where `u` serves as an optional hole name [Jones et al. 2014].[1]

Although better than nothing, this approach of stopping with an exception upon reaching an expression hole has limitations as the basis for a live programming environment because (1) it provides limited information about the dynamic state of the program when the exception occurs (typically only a stack trace); (2) 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); and (3) it provides no means by which to resume evaluation after filling a hole.

[1]Without these flags, holes cause compilation to fail. The compiler reports static 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.

Moreover, we need to be able to run programs containing not only expression holes but also type holes. The static semantics developed by Omar et al. [2017a] derives the machinery for reasoning statically about type holes from gradual type theory, identifying the type hole with the unknown type [Siek and Taha 2006; Siek et al. 2015]. As such, we can look to the dynamic semantics from gradual type theory to specify the dynamic semantics for incomplete programs, which selectively inserts dynamic casts as necessitated by missing type information to maintain type safety. However, when a cast fails, evaluation again stops with an exception. As a result, traditional gradual typing approaches still leave the live programming environment unable to provide feedback about the parts of the program that do not depend on the problematic sub-expression.

Contributions. This paper develops a theoretically well-grounded dynamic semantics for incomplete functional programs that addresses the limitations of the "exceptional approach" just described. In particular, rather than stopping with an exception when evaluation encounters an expression hole, evaluation continues "around" the hole, performing as much of the remaining computation as possible. Evaluation also proceeds past failed casts in much the same way. The system tracks the environment around each hole instance as evaluation proceeds. The live programming environment can feed relevant information from these hole environments to the programmer as they work to fill the holes in the program. Then, when the programmer performs an edit that fills an empty expression hole or that replaces a non-empty hole with a type-correct expression, evaluation can resume from the paused, i.e. *indeterminate*, evaluation state.

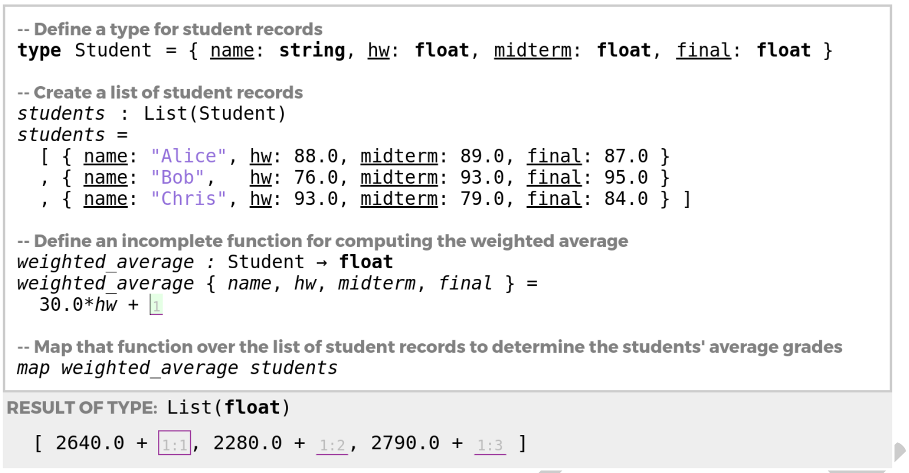
Paper Outline. We are integrating this approach into the Hazel programming environment being developed (separately from this paper) by Omar et al. [2017b]. We begin in Sec. 2 by detailing the approach informally, with several examples, in the setting of this specific design.

Sec. 3 then abstracts away orthogonal details of the language and user interface and makes these intuitions formally precise by detailing the primary contribution of this paper: a core calculus, Hazelnut Live, that supports evaluating incomplete expressions as just outlined, in a manner that admits clean type safety theorems (in particular, a Progress theorem). The semantics of Hazelnut Live borrows machinery from gradual type theory [Siek et al. 2015] (to handle incomplete types) and contextual modal type theory [Nanevski et al. 2008] (CMTT, which provides a logical foundation for hole environment tracking). These non-obvious connections to well-established existing systems help support our claim that this approach is theoretically well-grounded.

Sec. 3.4 outlines our mechanization of the core calculus using the Agda proof assistant. Sec. 3.5 outlines our simple implementation of the core calculus, which implements the live programming features from Sec. 2, but for the more austere language of Sec. 3. The editor component of this implementation provides a language of structured edit actions, based on the Hazelnut structure editor calculus developed by [Omar et al. 2017a], that guarantees that every editor state has some, possibly incomplete, type. Together with the theorems established earlier in Sec. 3, the result is a proof-of-concept live functional programming environment where every possible editor state has both non-trivial static and dynamic meaning, i.e. live feedback never "flickers" or "goes out". The supplemental material includes both the mechanization and the implementation.

Sec. 4 defines the fill-and-resume feature, which is rooted in the contextual substitution operation from CMTT. We establish the correctness of fill-and-resume with a novel commutativity property and discuss how it provides a semantic interpretation of cells in a live lab notebook environment.

Sec. 5 describes related work in more detail and simultaneously discusses directions for future work. Sec. 6 briefly concludes. The appendix includes some straightforward auxiliary definitions and proofs mentioned in the paper, some extensions to the core calculus, and some screenshots and additional details on the implementation.



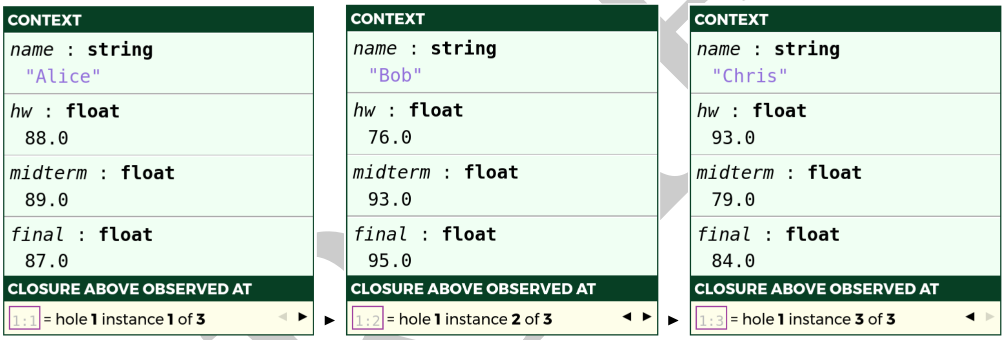
```
-- Define a type for student records
type Student = { name: string, hw: float, midterm: float, final: float }

-- Create a list of student records
students : List(Student)
students =
  [ { name: "Alice", hw: 88.0, midterm: 89.0, final: 87.0 }
  , { name: "Bob",   hw: 76.0, midterm: 93.0, final: 95.0 }
  , { name: "Chris", hw: 93.0, midterm: 79.0, final: 84.0 } ]

-- Define an incomplete function for computing the weighted average
weighted_average : Student → float
weighted_average { name, hw, midterm, final } =
  30.0*hw + 1

-- Map that function over the list of student records to determine the students' average grades
map weighted_average students
RESULT OF TYPE: List(float)
 [ 2640.0 + 1:1, 2280.0 + 1:2, 2790.0 + 1:3 ]
```

(a) Evaluating an incomplete functional program past the first hole



```
CONTEXT                       CONTEXT                       CONTEXT
name : string                 name : string                 name : string
  "Alice"                       "Bob"                         "Chris"
hw : float                    hw : float                    hw : float
 88.0                          76.0                          93.0
midterm : float               midterm : float               midterm : float
 89.0                          93.0                          79.0
final : float                 final : float                 final : float
 87.0                          95.0                          84.0
CLOSURE ABOVE OBSERVED AT     CLOSURE ABOVE OBSERVED AT     CLOSURE ABOVE OBSERVED AT
1:1 = hole 1 instance 1 of 3  1:2 = hole 1 instance 2 of 3  1:3 = hole 1 instance 3 of 3
```

(b) The programming environment communicates relevant static *and* dynamic information in a sidebar.

Fig. 1. Example 1: Grades

2 Live Functional Programming with Typed Holes in Hazel

We begin with an example-driven overview of our proposed approach within the Hazel programming environment Omar et al. [2017b]. Hazel is an open source web lab notebook environment based roughly on IPython/Jupyter [Perez and Granger 2007]. The language of Hazel is tracking toward parity with Elm, a popular pure functional programming language similar to "core ML" [Czaplicki 2012, 2018]. We assume familiarity with the basic concepts of ML/Elm (see http://elm-lang.org/).

Both the user interface and the language of Hazel are under active development. For the sake of exposition, we have "post-processed" the examples in this section to make use of several convenient language mechanisms from Elm that are not yet implemented in Hazel as of this writing. These mechanisms are orthogonal to the contributions of this paper—we have implemented all of the live programming features described in this section, both in Hazel and in the implementation of the core calculus included in the supplement (see Sec. 3.5).

2.1 Example 1: Evaluating Past Holes and Hole Closures

Consider the perspective of a teacher in the midst of developing a Hazel notebook to compute final student grades at the end of a course. The top of Figure 1 depicts the cell containing the incomplete program that the teacher has written so far (we omit irrelevant parts of the UI).

At the top of this program, the teacher defines a record type, `Student`, for recording a student's course data—here, the student's name, of type **`string`**, and, for simplicity, three grades, each of type **`float`**. The variable `students` binds a list of such student records. For simplicity, we include only three example students. At the bottom of the program, the teacher maps a function `weighted_average` over this student data (`map` is the standard map function over lists, not shown), intending to compute a final weighted average for each student. However, the program is incomplete because the teacher has not yet completed the body of the `weighted_average` function. This pattern is well-established: programmers often consume a function before implementing it.

Thusfar in Fig. 1, the teacher has decomposed the function argument into variables by record destructuring, then multiplied the homework grade, `hw`, by `30.0` and finally inserted the `+` operator. The cursor now sits at an *empty hole*, as indicated by the vertical bar and the green background. Each hole has a unique name (generated automatically in Hazel), here simply `1`.

Let us briefly digress: in a conventional "batch" programming system, writing `30.0*hw +` by itself would simply cause parsing to fail and there would be no static or dynamic feedback available to the programmer. In response, the programmer might insert a temporary universal placeholder, e.g. by raising an exception as suggested in Sec. 1 (note however that Elm does not support exceptions, by design). This would cause typechecking to succeed. However, when the programming environment attempts to evaluate the program, evaluation would proceed only as far as the first `map` iteration, which would call into `weighted_average` and then fail when attempting to evaluate the hole.[2] While the stack trace would confirm that `weighted_average` has been called, this is not entirely satisfying.

Hazel does not take this "exceptional" interpretation of holes. Instead, evaluation continues past the hole, treating it as an opaque expression of the appropriate type. The result, shown at the bottom of Fig. 1a, is a list of length 3, confirming that `map` does indeed behave as expected in this regard despite the teacher having provided an incomplete argument. Furthermore, each element of the resulting list has been evaluated as far as possible, i.e. the arithmetic expression `30.0*hw` has been evaluated for each corresponding value of `hw`, as expected. Evaluation cannot proceed any further because holes appear as addends. We say that each of these addition expressions is an *indeterminate* sub-expression, and the result as a hole is therefore also indeterminate, because it is not yet a value, nor can it take a step due to holes in critical positions.

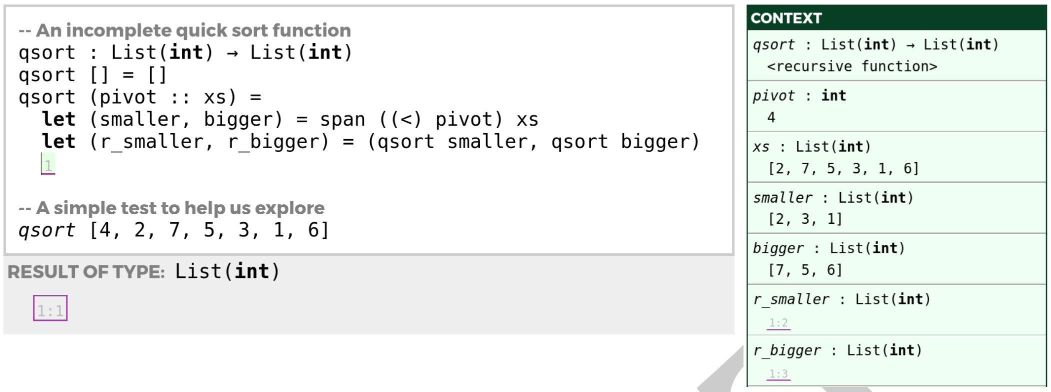
Given this indeterminate result, the teacher might take notice of the magnitude of the numbers being computed, e.g. `2640.0` and `2280.0`, and decide immediately that a mistake has been made: the teacher wants to compute a weighted average between `0.0` and `100.0`, and so the correct constant is `0.30` rather than `30.0`. (There are of course other possible fixes.)

Although these observations might save only a small amount of time in this case, it demonstrates the broader motivations of live programming: continuous feedback about the dynamic behavior of the program can help confirm the mental model that the programmer has developed (in this case, regarding the behavior of `map`), or else help quickly dispel inconsistencies between the programmer's expectations and the actual behavior of the program (in this case, the arithmetic expression).

Hazel also helps programmers reason about the dynamic behavior of bound variables in scope at a hole via the *live context inspector*, normally displays it as a sidebar but shown disembodied in three states in Fig. 1b. In all three states, the live context inspector displays the names and types of the variables that are in scope, as is a common feature of programming environments.

New to our approach are the values associated with each binding. These values come from the environments associated with each hole instance in the result. In this case, there are three instances of hole `1` in the result, numbered sequentially: `1:1`, `1:2` and `1:3`, arising from the three calls to `weighted_averages`. An environment is a partial mapping of each variable in scope to a value. We

[2]In a lazy language, like Haskell, the result would be much the same because the environment forces the result for printing.



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

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

```
RESULT OF TYPE: List(int)
 1:1
```

```
CONTEXT
qsort : List(int) → List(int)
  <recursive function>
pivot : int
  4
xs : List(int)
  [2, 7, 5, 3, 1, 6]
smaller : List(int)
  [2, 3, 1]
bigger : List(int)
  [7, 5, 6]
r_smaller : List(int)
  1:2
r_bigger : List(int)
  1:3
```

Fig. 2. Example 2: Incomplete Quicksort

call a hole instance paired with an environment a *hole closure*, by analogy to function closures (and also due to the logical foundations detailed in Sec. 3). The programmer can select different closures by clicking on a hole instance (here, instance `1:1` was selected by default, as indicated by the purple outline), or cycle rapidly through all of the closures for the hole at the cursor by clicking the arrows, e.g. ▶, at the bottom of the context inspector, or corresponding hotkeys.

2.2 Example 2: Recursive Functions

The first example was purposely simple. Let us briefly consider a second more sophisticated example of these concepts: an incomplete implementation of quicksort as a recursive function, shown in Fig. 2. Here, the programmer has selected a pivot, split the remainder of the list relative to this pivot and made the appropriate recursive calls. A hole appears in the return position of the function body as the programmer contemplates how to combine the results from the recursive calls.

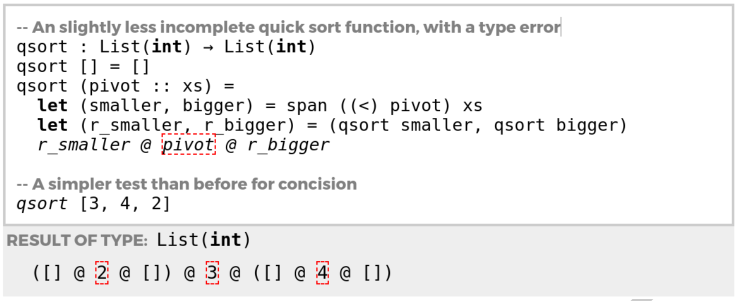
The programmer has called `qsort` on an example list. However, the indeterminate result in this case is rather uninteresting: it simply confirms that for this example, evaluation went through the recursive case of `qsort`, which ends with hole `1`. The live context inspector, on the other hand, provides a wealth of feedback about the dynamic behavior of this incomplete program. For example, the programmer can confirm that the lists `smaller` and `bigger` are appropriately named.

The results from the recursive calls, `r_smaller` and `r_bigger`, are again hole instances, `1:2` and `1:3`. The programmer can click on these hole instances to reveal the environments at the corresponding recursive calls. For example, clicking on `1:2` reveals that the pivot in that case is `7` (not shown). In exploring these paths rooted at the result, the programmer can develop intuitions about the recursive structure of the computation before the program is complete. (We leave to future work the pedagogical question of whether such early feedback might help students with recursion.)

2.3 Example 3: Live Programming with Static Type Errors

The previous examples were incomplete because of *missing* expressions. Now, we discuss programs that are incomplete, and therefore conventionally meaningless, because of *type inconsistencies*. Let us return to the quicksort example just described, but assume that the programmer has filled in the previous hole as shown in Fig. 3. (In Sec. 4, we discuss how the programming environment might avoid restarting evaluation after such edits, but for small examples like this, restarting works fine.)

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 out: the `@` operator here performs list concatenation, but the pivot is an integer. Most compilers and editors will report a static error message, of varying utility, to the programmer in this case



```
-- An slightly less incomplete quick sort function, with a type error
qsort : List(int) → List(int)
qsort [] = []
qsort (pivot :: xs) =
  let (smaller, bigger) = span ((<) pivot) xs
  let (r_smaller, r_bigger) = (qsort smaller, qsort bigger)
  r_smaller @ pivot @ r_bigger

-- A simpler test than before for concision
qsort [3, 4, 2]
RESULT OF TYPE: List(int)
  ([] @ 2 @ []) @ 3 @ ([] @ 4 @ [])
```

Fig. 3. Example 3: Ill-Typed Quicksort

(and Hazel follows suit, not shown). Our argument is that this should not cause all feedback about the dynamic behavior of the program to “flicker out” or “go stale”. After all, the error is localized and there is perfectly good code elsewhere in the program (if not nearby, then perhaps far away).

Our approach, following the prior work of Omar et al. [2017a], is to semantically internalize the “red outline” around type inconsistencies as a *non-empty hole*. From the outside, a non-empty hole behaves statically much like an empty hole. Our approach takes the same approach for the dynamic semantics, i.e. evaluation safely proceeds past a non-empty hole just as if it were an empty hole. The semantics also associates an environment with each instance of a non-empty hole (not shown). The difference is that evaluation also proceeds inside the hole, so that feedback about an expression that might “almost” be correct is available. In this case, the result at the bottom of Fig. 3 reveals that the programmer is on the right track: the list elements (here, for a shorter example for the sake of exposition) are in the right order. They simply have not been combined correctly.

2.4 Example 4: Gradual Type Errors

So far, we have only discussed programs where a hole appears within an expression. Hazel also allows holes to appear in types. Fortunately, Omar et al. [2017a] confirms that the substantial literature on *gradual type systems* [Siek and Taha 2006; Siek et al. 2015] is directly relevant to the problem of reasoning statically around type holes. Unsurprisingly, it is also relevant to the problem of running programs with incomplete types. Indeed, that is the very purpose of gradual typing. As such, let us consider only a small synthetic example to demonstrate what is different in Hazel.

```
1  f : ? -> int
2  f x = if x then 1 else 2
3  (f true, f 3, f false)
```

Evaluation produces the following result:

```
(1, if 3<int => ? =/> bool> then 1 else 2, 2)
```

Here, we have left a hole, ?, in the type of f. On Line 3, we call f three times, with two different types of arguments. The static semantics of gradual typing permits all of these calls (see next section) and indeed, the first call, with true as the argument, evaluates unproblematically to 1.

The second call, however, is problematic: 3 is of type **int** but it flows into boolean guard position. The usual semantics for gradual typing would simply abort evaluation here with a dynamic type error. However, this deprives the programmer of live feedback about other parts of the program that might not depend at all on this problematic function call—in this case, the third call to f. Instead, we follow the approach we took above for static type inconsistencies: we treat a failed cast like a hole. A failed cast, notated textually and in red 3<**int** => ? =/> bool> (see next section), reports both the expected type, here bool, and the assigned type, here **int**. Consequently, the second call to f stops as if the guard were an empty hole, but the third call can progress safely.

$$
\begin{array}{rcll}
\mathsf{HTyp} & \tau & ::= & b \mid \tau \to \tau \mid \llparenthesis\rrparenthesis \\
\mathsf{HExp} & e & ::= & c \mid x \mid \lambda x{:}\tau.e \mid \lambda x.e \mid e(e) \mid \llparenthesis\rrparenthesis^u \mid \llparenthesis e \rrparenthesis^u \mid e : \tau \\
\mathsf{IHExp} & d & ::= & c \mid x \mid \lambda x{:}\tau.d \mid d(d) \mid \llparenthesis\rrparenthesis^u_\sigma \mid \llparenthesis d \rrparenthesis^u_\sigma \mid d\langle \tau \Rightarrow \tau \rangle \mid d\langle \tau \Rightarrow \llparenthesis\rrparenthesis \nRightarrow \tau \rangle
\end{array}
$$

$$d\langle \tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3 \rangle \overset{\text{def}}{=} d\langle \tau_1 \Rightarrow \tau_2 \rangle \langle \tau_2 \Rightarrow \tau_3 \rangle$$

Fig. 4. Syntax of types, τ, external expressions, e, and internal expressions, d. We write x to range over variables, u over hole names, and σ over finite substitutions (i.e., environments) which map variables to internal expressions, written $[d_1/x_1, \cdots, d_n/x_n]$ for $n \geq 0$.

$\boxed{\Gamma \vdash e \Rightarrow \tau}$ e synthesizes type τ

$$
\text{SConst}\ \frac{}{\Gamma \vdash c \Rightarrow b}
\qquad
\text{SVar}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x \Rightarrow \tau}
\qquad
\text{SLam}\ \frac{\Gamma, x : \tau_1 \vdash e \Rightarrow \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.e \Rightarrow \tau_1 \to \tau_2}
\qquad
\text{SAp}\ \frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \quad \Gamma \vdash e_2 \Leftarrow \tau_2 \quad \tau_1 \blacktriangleright_{\to} \tau_2 \to \tau}{\Gamma \vdash e_1(e_2) \Rightarrow \tau}
$$

$$
\text{SEHole}\ \frac{}{\Gamma \vdash \llparenthesis\rrparenthesis^u \Rightarrow \llparenthesis\rrparenthesis}
\qquad
\text{SNEHole}\ \frac{\Gamma \vdash e \Rightarrow \tau}{\Gamma \vdash \llparenthesis e \rrparenthesis^u \Rightarrow \llparenthesis\rrparenthesis}
\qquad
\text{SAsc}\ \frac{\Gamma \vdash e \Leftarrow \tau}{\Gamma \vdash e : \tau \Rightarrow \tau}
$$

$\boxed{\Gamma \vdash e \Leftarrow \tau}$ e analyzes against type τ

$$
\text{ALam}\ \frac{\tau \blacktriangleright_{\to} \tau_1 \to \tau_2 \qquad \Gamma, x : \tau_1 \vdash e \Leftarrow \tau_2}{\Gamma \vdash \lambda x.e \Leftarrow \tau}
\qquad
\text{ASubsume}\ \frac{\Gamma \vdash e \Rightarrow \tau \qquad \tau \sim \tau'}{\Gamma \vdash e \Leftarrow \tau'}
$$

Fig. 5. Bidirectional Typing of External Expressions

3 Hazelnut Live, Formally

We will now make the intuitions developed in the previous section formally precise by specifying a core calculus, which we call Hazelnut Live, and developing its metatheory.

Overview. The syntax of the core calculus given in Fig. 4 consists of types and expressions with holes. We distinguish between external expressions, e, and internal expressions, d. External expressions correspond to programs as entered by the programmer (see Sec. 1 for discussion of implicit, manual, semi-automated and fully automated hole entry methods). Each well-typed external expression (as specified in Sec. 3.1 below) expands to a well-typed internal expression (see Sec. 3.2) before it is evaluated (see Sec. 3.3). We take this approach, notably also taken in the "redefinition" of Standard ML by Harper and Stone [2000], because (1) the external language supports type inference and explicit type ascriptions, $e : \tau$, but it is formally simpler to eliminate ascriptions and specify a type assignment system when defining the dynamic semantics; and (2) we need additional syntactic machinery during evaluation for tracking hole closures and dynamic type casts. This machinery is inserted by the expansion step, rather than entered explicitly by the programmer. In this regard, the internal language is analogous to the cast calculus in the gradually typed lambda calculus [Siek and Taha 2006; Siek et al. 2015], though as we will see the Hazelnut Live internal language goes beyond the cast calculus in several respects. We have mechanized these formal developments using the Agda proof assistant [Norell 2007, 2009] (see Sec. 3.4). Rule names in this section, e.g. SVar, correspond to variable names used in the mechanization. We have also implemented Hazelnut Live in a manner that maintains a novel end-to-end well-definedness invariant (see Sec. 3.5).

$\boxed{\tau_1 \sim \tau_2}$ τ_1 is consistent with τ_2

$$\text{TCHole1}\ \frac{}{⦇⦈ \sim \tau} \qquad \text{TCHole2}\ \frac{}{\tau \sim ⦇⦈} \qquad \text{TCRefl}\ \frac{}{\tau \sim \tau} \qquad \text{TCArr}\ \frac{\tau_1 \sim \tau_1' \qquad \tau_2 \sim \tau_2'}{\tau_1 \to \tau_2 \sim \tau_1' \to \tau_2'}$$

$\boxed{\tau \blacktriangleright_{\to} \tau_1 \to \tau_2}$ τ has matched arrow type $\tau_1 \to \tau_2$

$$\text{MAHole}\ \frac{}{⦇⦈ \blacktriangleright_{\to} ⦇⦈ \to ⦇⦈} \qquad \text{MAArr}\ \frac{}{\tau_1 \to \tau_2 \blacktriangleright_{\to} \tau_1 \to \tau_2}$$

Fig. 6. Type Consistency and Matching

3.1 Static Semantics of the External Language

We start with the type system of the Hazelnut Live external language, which closely follows the Hazelnut type system [Omar et al. 2017a]; we summarize the minor differences as they come up.

Fig. 5 defines the type system in the *bidirectional* style with two mutually defined judgements [Chlipala et al. 2005; Christiansen 2013; Dunfield and Krishnaswami 2013; Pierce and Turner 2000]. The type synthesis judgement $\Gamma \vdash e \Rightarrow \tau$ synthesizes a type τ for external expression e under typing context Γ, which tracks typing assumptions of the form $x : \tau$ in the usual manner [Harper 2016; Pierce 2002]. The type analysis judgement $\Gamma \vdash e \Leftarrow \tau$ checks expression e against a given type τ. Algorithmically, analysis accepts a type as input, and synthesis gives a type as output. We start with synthesis for the programmer's "top level" external expression.

The primary benefit of specifying the Hazelnut Live external language bidirectionally is that the programmer need not annotate each hole with a type. An empty hole is written simply $⦇⦈^u$, where u is the hole name, which we tacitly assume is unique (holes in Hazelnut were not named). Rule SEHole specifies that an empty hole synthesizes hole type, written $⦇⦈$. If an empty hole appears where an expression of some other type is expected, e.g. under an explicit ascription (governed by Rule SAsc) or in the argument position of a function application (governed by Rule SAp, discussed below), we apply the *subsumption rule*, Rule ASubsume, which specifies that if an expression e synthesizes type τ, then it may be checked against any *consistent* type, τ'.

Fig. 6 specifies the type consistency relation, written $\tau \sim \tau'$, which specifies that two types are consistent if they differ only up to type holes in corresponding positions. The hole type is consistent with every type, and so, by the subsumption rule, expression holes may appear where an expression of any type is expected. The type consistency relation here coincides with the type consistency relation from gradual type theory by identifying the hole type with the unknown type [Siek and Taha 2006]. Type consistency is reflexive and symmetric, but it is *not* transitive. This stands in contrast to subtyping, which is anti-symmetric and transitive; subtyping may be integrated into a gradual type system following Siek and Taha [2007].

Non-empty expression holes, written $⦇e⦈^u$, behave similarly to empty holes. Rule SNEHole specifies that a non-empty expression hole also synthesizes hole type as long as the expression inside the hole, e, synthesizes some (arbitrary) type. Non-empty expression holes therefore internalize the "red underline/outline" that many editors display around type inconsistencies in a program.

For the familiar forms of the lambda calculus, the rules again follow prior work. For simplicity, the core calculus includes only a single base type b with a single constant c, governed by Rule SConst (i.e. b is the unit type). By contrast, Omar et al. [2017a] instead defined a number type with a single operation. That paper also defined sum types as an extension to the core calculus. We follow suit on both counts in Appendix B.

Rule SVar synthesizes the corresponding type from Γ. For the sake of exposition, Hazelnut Live includes "half-annotated" lambdas, $\lambda x{:}\tau.e$, in addition to the unannotated lambdas, $\lambda x.e$, from Hazelnut. Half-annotated lambdas may appear in synthetic position according to Rule SLam, which is standard [Chlipala et al. 2005]. Unannotated lambdas may only appear where the expected type is known to be either an arrow type or the hole type, which is treated as if it were $(\!|\!) \to (\!|\!)$. To avoid the need for two separate rules, Rule ALam uses the matching relation $\tau \blacktriangleright_{\to} \tau_1 \to \tau_2$ defined in Fig. 6, which produces the matched arrow type $(\!|\!) \to (\!|\!)$ given the hole type, and operates as the identity on arrow types [Garcia and Cimini 2015a; Siek et al. 2015].[3]

The rule governing function application, Rule SAp, similarly treats an expression of hole type in function position as if it were of type $(\!|\!) \to (\!|\!)$ using the same matched arrow type judgement.

3.2 Expansion

Each well-typed external expression e expands to a well-typed internal expression d, for evaluation. Fig. 7 specifies expansion, and Fig. 8 specifies type assignment for internal expressions.

As with the type system for the external language (above), we specify expansion bidirectionally [Ferreira and Pientka 2014]. The synthetic expansion judgement $\Gamma \vdash e \Rightarrow \tau \leadsto d \dashv \Delta$ produces an expansion d and a hole context Δ when synthesizing type τ for e. We describe hole contexts, which serve as "inputs" to the type assignment judgement $\Delta; \Gamma \vdash d : \tau$, further below. The analytic expansion judgement $\Gamma \vdash e \Leftarrow \tau \leadsto d : \tau' \dashv \Delta$, produces an expansion d of type τ', and a hole context Δ, when checking e against τ. The following theorem establishes that expansions are well-typed and in the analytic case that the assigned type, τ', is consistent with provided type, τ.

Theorem 3.1 (Typed Expansion).

(1) If $\Gamma \vdash e \Rightarrow \tau \leadsto d \dashv \Delta$ then $\Delta; \Gamma \vdash d : \tau$.

(2) If $\Gamma \vdash e \Leftarrow \tau \leadsto d : \tau' \dashv \Delta$ then $\tau \sim \tau'$ and $\Delta; \Gamma \vdash d : \tau'$.

The reason that τ' is only consistent with the provided type τ is because the subsumption rule permits us to check an external expression against any type consistent with the type that the expression *actually* synthesizes, whereas every internal expression can be assigned at most one type, i.e. the following standard unicity property holds of the type assignment system.

Theorem 3.2 (Type Assignment Unicity). *If $\Delta; \Gamma \vdash d : \tau$ and $\Delta; \Gamma \vdash d : \tau'$ then $\tau = \tau'$.*

Consequently, analytic expansion reports the type actually assigned to the expansion it produces. For example, we can derive that $\Gamma \vdash c \Leftarrow (\!|\!) \leadsto c : b \dashv \emptyset$.

Before describing the rules in detail, let us state two other guiding theorems. The following theorem establishes that an expansion exists for every well-typed external expression. The mechanization also establishes that when an expansion exists, it is unique (not shown).

Theorem 3.3 (Expandability).

(1) If $\Gamma \vdash e \Rightarrow \tau$ then $\Gamma \vdash e \Rightarrow \tau \leadsto d \dashv \Delta$ for some d and Δ.

(2) If $\Gamma \vdash e \Leftarrow \tau$ then $\Gamma \vdash e \Leftarrow \tau \leadsto d : \tau' \dashv \Delta$ for some d and τ' and Δ.

The following theorem establishes that expansion generalizes external typing.

Theorem 3.4 (Expansion Generality).

(1) If $\Gamma \vdash e \Rightarrow \tau \leadsto d \dashv \Delta$ then $\Gamma \vdash e \Rightarrow \tau$.

(2) If $\Gamma \vdash e \Leftarrow \tau \leadsto d : \tau' \dashv \Delta$ then $\Gamma \vdash e \Leftarrow \tau$.

The rules governing expansion of constants, variables and lambda expressions — Rules ESConst, ESVar, ESLam and EALam — mirror the corresponding type assignment rules — Rules TAConst, TAVar and TALam — and in turn, the corresponding bidirectional typing rules from Fig. 5. To support

[3] A system supporting ML-style type reconstruction [Damas and Milner 1982] might also include a synthetic rule for unannotated lambdas, e.g. as outlined by Dunfield and Krishnaswami [2013], but we stick to this simpler "Scala-style" local type inference scheme in this paper [Odersky et al. 2001; Pierce and Turner 2000].

$\boxed{\Gamma \vdash e \Rightarrow \tau \rightsquigarrow d \dashv \Delta}$ e synthesizes type τ and expands to d

ESConst
$$\frac{}{\Gamma \vdash c \Rightarrow b \rightsquigarrow c \dashv \emptyset}$$

ESVar
$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x \Rightarrow \tau \rightsquigarrow x \dashv \emptyset}$$

ESLam
$$\frac{\Gamma, x : \tau_1 \vdash e \Rightarrow \tau_2 \rightsquigarrow d \dashv \Delta}{\Gamma \vdash \lambda x{:}\tau_1.e \Rightarrow \tau_1 \to \tau_2 \rightsquigarrow \lambda x{:}\tau_1.d \dashv \Delta}$$

ESAp
$$\frac{\begin{array}{c}\Gamma \vdash e_1 \Rightarrow \tau_1 \qquad \tau_1 \blacktriangleright_{\to} \tau_2 \to \tau \\ \Gamma \vdash e_1 \Leftarrow \tau_2 \to \tau \rightsquigarrow d_1 : \tau_1' \dashv \Delta_1 \qquad \Gamma \vdash e_2 \Leftarrow \tau_2 \rightsquigarrow d_2 : \tau_2' \dashv \Delta_2\end{array}}{\Gamma \vdash e_1(e_2) \Rightarrow \tau \rightsquigarrow (d_1\langle \tau_1' \Rightarrow \tau_2 \to \tau\rangle)(d_2\langle \tau_2' \Rightarrow \tau_2\rangle) \dashv \Delta_1 \cup \Delta_2}$$

ESEHole
$$\frac{}{\Gamma \vdash \llparenthesis\rrparenthesis^u \Rightarrow \llparenthesis\rrparenthesis \rightsquigarrow \llparenthesis\rrparenthesis^u_{\mathrm{id}(\Gamma)} \dashv u :: \llparenthesis\rrparenthesis[\Gamma]}$$

ESNEHole
$$\frac{\Gamma \vdash e \Rightarrow \tau \rightsquigarrow d \dashv \Delta}{\Gamma \vdash \llparenthesis e\rrparenthesis^u \Rightarrow \llparenthesis\rrparenthesis \rightsquigarrow \llparenthesis d\rrparenthesis^u_{\mathrm{id}(\Gamma)} \dashv \Delta, u :: \llparenthesis\rrparenthesis[\Gamma]}$$

ESAsc
$$\frac{\Gamma \vdash e \Leftarrow \tau \rightsquigarrow d : \tau' \dashv \Delta}{\Gamma \vdash e : \tau \Rightarrow \tau \rightsquigarrow d\langle \tau' \Rightarrow \tau\rangle \dashv \Delta}$$

$\boxed{\Gamma \vdash e \Leftarrow \tau_1 \rightsquigarrow d : \tau_2 \dashv \Delta}$ e analyzes against type τ_1 and expands to d of consistent type τ_2

EALam
$$\frac{\tau \blacktriangleright_{\to} \tau_1 \to \tau_2 \qquad \Gamma, x : \tau_1 \vdash e \Leftarrow \tau_2 \rightsquigarrow d : \tau_2' \dashv \Delta}{\Gamma \vdash \lambda x.e \Leftarrow \tau \rightsquigarrow \lambda x{:}\tau_1.d : \tau_1 \to \tau_2' \dashv \Delta}$$

EASubsume
$$\frac{\begin{array}{c}e \neq \llparenthesis\rrparenthesis^u \qquad e \neq \llparenthesis e'\rrparenthesis^u \\ \Gamma \vdash e \Rightarrow \tau' \rightsquigarrow d \dashv \Delta \qquad \tau \sim \tau'\end{array}}{\Gamma \vdash e \Leftarrow \tau \rightsquigarrow d : \tau' \dashv \Delta}$$

EAEHole
$$\frac{}{\Gamma \vdash \llparenthesis\rrparenthesis^u \Leftarrow \tau \rightsquigarrow \llparenthesis\rrparenthesis^u_{\mathrm{id}(\Gamma)} : \tau \dashv u :: \tau[\Gamma]}$$

EANEHole
$$\frac{\Gamma \vdash e \Rightarrow \tau' \rightsquigarrow d \dashv \Delta}{\Gamma \vdash \llparenthesis e\rrparenthesis^u \Leftarrow \tau \rightsquigarrow \llparenthesis d\rrparenthesis^u_{\mathrm{id}(\Gamma)} : \tau \dashv \Delta, u :: \tau[\Gamma]}$$

Fig. 7. Expansion

$\boxed{\Delta; \Gamma \vdash d : \tau}$ d is assigned type τ

TAConst
$$\frac{}{\Delta; \Gamma \vdash c : b}$$

TAVar
$$\frac{x : \tau \in \Gamma}{\Delta; \Gamma \vdash x : \tau}$$

TALam
$$\frac{\Delta; \Gamma, x : \tau_1 \vdash d : \tau_2}{\Delta; \Gamma \vdash \lambda x{:}\tau_1.d : \tau_1 \to \tau_2}$$

TAAp
$$\frac{\Delta; \Gamma \vdash d_1 : \tau_2 \to \tau \qquad \Delta; \Gamma \vdash d_2 : \tau_2}{\Delta; \Gamma \vdash d_1(d_2) : \tau}$$

TAEHole
$$\frac{u :: \tau[\Gamma'] \in \Delta \qquad \Delta; \Gamma \vdash \sigma : \Gamma'}{\Delta; \Gamma \vdash \llparenthesis\rrparenthesis^u_\sigma : \tau}$$

TANEHole
$$\frac{\begin{array}{c}\Delta; \Gamma \vdash d : \tau' \\ u :: \tau[\Gamma'] \in \Delta \qquad \Delta; \Gamma \vdash \sigma : \Gamma'\end{array}}{\Delta; \Gamma \vdash \llparenthesis d\rrparenthesis^u_\sigma : \tau}$$

TACast
$$\frac{\Delta; \Gamma \vdash d : \tau_1 \qquad \tau_1 \sim \tau_2}{\Delta; \Gamma \vdash d\langle \tau_1 \Rightarrow \tau_2\rangle : \tau_2}$$

TAFailedCast
$$\frac{\Delta; \Gamma \vdash d : \tau_1 \qquad \tau_1 \text{ ground} \qquad \tau_2 \text{ ground} \qquad \tau_1 \neq \tau_2}{\Delta; \Gamma \vdash d\langle \tau_1 \Rightarrow \llparenthesis\rrparenthesis \not\Rightarrow \tau_2\rangle : \tau_2}$$

Fig. 8. Type Assignment for Internal Expressions

type assignment, all lambdas in the internal language are half-annotated—Rule `EALam` inserts the annotation when expanding an unannotated external lambda based on the given type. The rules governing hole expansion, and the rules that perform *cast insertion*—those governing function application and type ascription—are more interesting. Let us consider each of these two groups of rules in turn in Sec. 3.2.1 and Sec. 3.2.2, respectively.

3.2.1 Hole Expansion. Rules `ESEHole`, `ESNEHole`, `EAEHole` and `EANEHole` govern the expansion of empty and non-empty expression holes to empty and non-empty *hole closures*, $⦇⦈^u_\sigma$ and $⦇d⦈^u_\sigma$. The hole name u on a hole closure identifies the external hole to which the hole closure corresponds. While we assume each hole name to be unique in the external language, once evaluation begins, there may be multiple hole closures with the same name due to substitution. For example, the result from Fig. 1 shows three closures for the hole named 1. There, we numbered each hole closure for a given hole sequentially, `1:1`, `1:2` and `1:3`, but this is strictly for the sake of presentation, so we omit hole closure numbers from the core calculus.

For each hole, u, in an external expression, the hole context generated by expansion, Δ, contains a hypothesis of the form $u :: \tau[\Gamma]$, which records the hole's type, τ, and the typing context, Γ, from where it appears in the original expression.[4] We borrow this hole context notation from contextual modal type theory (CMTT) [Nanevski et al. 2008], identifying hole names with metavariables and hole contexts with modal contexts (we say more about the connection with CMTT below). In the synthetic hole expansion rules `ESEHole` and `ESNEHole`, the generated hole context assigns the hole type $⦇⦈$ to hole name u, as in the external typing rules. However, the first two premises of the expansion subsumption rule `EASubsume` disallow the use of subsumption for holes in analytic position. Instead, we employ separate analytic rules `EAEHole` and `EANEHole`, which each record the checked type τ in the hole context. Consequently, we can use type assignment for the internal language — the type assignment rules `TAEHole` and `TANEHole` in Fig. 8 assign a hole closure for hole name u the corresponding type from the hole context.

Each hole closure also has an associated environment σ which consists of a finite substitution of the form $[d_1/x_1, \cdots, d_n/x_n]$ for $n \geq 0$. The closure environment keeps a record of the substitutions that occur around the hole as evaluation occurs. Initially, when no evaluation has yet occurred, the hole expansion rules generate the identity substitution for the typing context associated with hole name u in hole context Δ, which we notate $\text{id}(\Gamma)$, and define as follows.

Definition 3.5 (Identity Substitution). $\text{id}(x_1 : \tau_1, \cdots, x_n : \tau_n) = [x_1/x_1, \cdots, x_n/x_n]$

The type assignment rules for hole closures TAEHole and TANEHole each require that the hole closure environment σ be consistent with the corresponding typing context, written as $\Delta; \Gamma \vdash \sigma : \Gamma'$. Formally, we define this relation in terms of type assignment as follows:

Definition 3.6 (Substitution Typing). $\Delta; \Gamma \vdash \sigma : \Gamma'$ *iff* $dom(\sigma) = dom(\Gamma')$ *and for each* $x : \tau \in \Gamma'$ *we have that* $d/x \in \sigma$ *and* $\Delta; \Gamma \vdash d : \tau$.

It is easy to verify that the identity substitution satisfies this requirement, i.e. that $\Delta; \Gamma \vdash \text{id}(\Gamma) : \Gamma$.

Empty hole closures, $⦇⦈^u_\sigma$, correspond to the metavariable closures (a.k.a. deferred substitutions) from CMTT, $\text{clo}(u, \sigma)$. Sec. 3.3 defines how these closure environments evolve during evaluation. Non-empty hole closures $⦇d⦈^u_\sigma$ have no direct correspondence with a notion from CMTT (see Sec. 4).

3.2.2 Cast Insertion. Holes in types require us to defer certain structural checks to run time. To see why this is necessary, consider the following example: $(\lambda x{:}⦇⦈.x(c))(c)$. Viewed as an external expression, this example synthesizes type $⦇⦈$, since the hole type annotation on variable x permits applying x as a function of type $⦇⦈ \to ⦇⦈$, and base constant c may be checked against type $⦇⦈$, by

[4] We use a hole context, rather than recording the typing context and type directly on each hole closure, to ensure that all closures for a hole name have the same typing context and type.

subsumption. However, viewed as an internal expression, this example is not well-typed—the type assignment system defined in Fig. 8 lacks subsumption. Indeed, it would violate type safety if we could assign a type to this example in the internal language, because beta reduction of this example viewed as an internal expression would result in $c(c)$, which is clearly not well-typed. The difficulty arises because leaving the argument type unknown also leaves unknown how the argument is being used (in this case, as a function).[5] By our interpretation of hole types as unknown types from gradual type theory, we can address the problem by performing cast insertion.

The cast form in Hazelnut Live is $d\langle\tau_1 \Rightarrow \tau_2\rangle$. This form serves to "box" an expression of type τ_1 for treatment as an expression of a consistent type τ_2 (Rule TACast in Fig. 8).[6]

Expansion inserts casts at function applications and ascriptions. The latter is more straightforward: Rule ESAsc in Fig. 7 inserts a cast from the assigned type to the ascribed type. Theorem 3.1 inductively ensures that the two types are consistent. We include ascription for exposition purposes—this form is derivable by using application together with the half-annotated identity, $e : \tau = (\lambda x{:}\tau.x)(e)$; as such, application expansion, discussed below, is more general.

Rule ESAp expands function applications. To understand the rule, consider the expansion of external expression $(\lambda x{:}\llparenthesis\rrparenthesis.x(c))(c)$, the example discussed above:

$$(\lambda x{:}\llparenthesis\rrparenthesis.\underbrace{x\langle\llparenthesis\rrparenthesis \Rightarrow \llparenthesis\rrparenthesis \to \llparenthesis\rrparenthesis\rangle(c\langle b \Rightarrow \llparenthesis\rrparenthesis\rangle)}_{\text{expansion of function body } x(c)})\langle\llparenthesis\rrparenthesis \to \llparenthesis\rrparenthesis \Rightarrow \llparenthesis\rrparenthesis \to \llparenthesis\rrparenthesis\rangle(c\langle b \Rightarrow \llparenthesis\rrparenthesis\rangle)$$

Consider the (indicated) function body, where expansion inserts a cast on both the function expression x and its argument c. Together, these casts for x and c permit assigning a type to the function body according to the rules in Fig. 8, where we could not do so under the same context without casts. We separately consider the expansions of x and of c.

First, consider the function position of this application, here variable x. Without any cast, the type of variable x is the hole type $\llparenthesis\rrparenthesis$; however, the inserted cast on x permits treating it as though it has arrow type $\llparenthesis\rrparenthesis \to \llparenthesis\rrparenthesis$. The first three premises of Rule ESAp accomplish this by first synthesizing a type for the function expression, here $\llparenthesis\rrparenthesis$, then by determining the matched arrow type $\llparenthesis\rrparenthesis \to \llparenthesis\rrparenthesis$, and finally, by performing analytic expansion on the function expression with this matched arrow type. The resulting expansion has some type τ_1' consistent with the matched arrow type. In this case, because the subexpression x is a variable, analytic expansion goes through subsumption so that type τ_1' is simply $\llparenthesis\rrparenthesis$. The conclusion of the rule inserts the corresponding cast. We go through type synthesis, *then* analytic expansion, so that the hole context records the matched arrow type for holes in function position, rather than the type $\llparenthesis\rrparenthesis$ for all such holes, as would be the case in a variant of this rule using synthetic expansion for the function expression.

Next, consider the application's argument, here constant c. The conclusion of Rule ESAp inserts the cast on the argument's expansion, from the type it is assigned by the final premise of the rule (type b here), to the argument type of the matched arrow type of the function expression (type $\llparenthesis\rrparenthesis$ here).

The example's second, outermost application goes through the same application expansion rule. In this case, the cast on the function is the identity cast for $\llparenthesis\rrparenthesis \to \llparenthesis\rrparenthesis$. For simplicity, we do not attempt to avoid the insertion of identity casts in the core calculus; these will simply never fail during evaluation. However, it is safe in practice to eliminate such identity casts during expansion, and some formal accounts of gradual typing do so by defining three application expansion rules, including the original account of Siek and Taha [2006].

[5]In a system where type reconstruction is first used to try to fill in type holes, we could express a similar example by using x at two or more different types, thereby causing type reconstruction to fail.

[6] In the earliest work on gradual type theory, the cast form only gave the target type τ_2 [Siek and Taha 2006], but it simplifies the dynamic semantics substantially to include the assigned type τ_1 in the syntax [Siek et al. 2015].

3.3 Dynamic Semantics

To recap, the result of expansion is a well-typed internal expression with appropriately initialized hole closures and casts. This section specifies the dynamic semantics of Hazelnut Live as a "small-step" transition system over internal expressions equipped with a meaningful notion of type safety even for incomplete programs, i.e. expressions typed under a non-empty hole context, Δ. We establish that evaluation does not stop immediately when it encounters a hole, nor when a cast fails, by precisely characterizing when evaluation *does* stop. We also establish that an essentially standard notion of type safety holds when running complete programs.

It is perhaps worth stating at the outset that a dynamic semantics equipped with these properties does not simply "fall out" from the observations made above that (1) empty hole closures correspond to metavariable closures from CMTT [Nanevski et al. 2008] and (2) casts also arise in gradual type theory [Siek et al. 2015]. We say more in Sec. 5.

Figures 9-12 define the dynamic semantics. Most of the cast-related machinery closely follows the cast calculus from the "refined" account of the gradually typed lambda calculus by Siek et al. [2015], which is known to be theoretically well-behaved. In particular, Fig. 9 defines the judgement τ ground, which distinguishes the base type b and the least specific arrow type $⦇⦈ \to ⦇⦈$ as *ground types*; this judgement helps simplify the treatment of function casts, discussed below.

Fig. 10 defines the judgement d final, which distinguishes the final, i.e. irreducible, forms of the transition system. The two rules distinguish two classes of final forms: (possibly-)boxed values and indeterminate forms.[7] The judgement d boxedval defines (possibly-)boxed values as either ordinary values (Rule BVVal), or one of two cast forms: casts between disequal function types and casts from a ground type to the hole type. In each case, the cast must appear inductively on a boxed value. These forms are irreducible because they represent values that have been boxed but have never flowed into a corresponding "unboxing" cast, discussed below. The judgement d indet defines *indeterminate* forms, so named because they are rooted at expression holes and failed casts, and so, conceptually, their ultimate value awaits programmer action (see Sec. 4). The first two rules specify that empty hole closures are always indeterminate, and that non-empty hole closures are indeterminate when they consist of a final inner expression. Below, we describe failed casts and the remaining indeterminate forms simultaneously with the corresponding transition rules.

Figures 11-12 define the transition rules. Top-level transitions are *steps*, $d \mapsto d'$, governed by Rule Step in Fig. 12, which (1) decomposes d into an evaluation context, $\mathcal{E}$, and a selected sub-term, d_0; (2) takes an *instruction transition*, $d_0 \longrightarrow d_0'$, as specified in Fig. 11; and (3) places d_0' back at the selected position, indicated in the evaluation context by the *mark*, $\circ$, to obtain d'.[8] This approach was originally developed in the reduction semantics of Felleisen and Hieb [1992] and is the predominant style of operational semantics in the literature on gradual typing. Because we distinguish final forms judgementally, rather than syntactically, we use a judgemental formulation of this approach called a *contextual dynamics* by Harper [2016]. It would be straightforward to construct an equivalent structural operational semantics [Plotkin 2004] by using search rules instead of evaluation contexts (Harper [2016] relates the two approaches).

The rules maintain the property that final expressions truly cannot take a step.

Theorem 3.7 (Finality). *There does not exist d such that both d* final *and $d \mapsto d'$ for some d'.*

3.3.1 Application and Substitution. Rule ITBeta in Fig. 11 defines the standard beta reduction transition. The bracketed premises of the form [d final] in Fig. 11-12 may be *included* to specify

[7] Most accounts of the cast calculus distinguish ground types and values with separate grammars together with an implicit identification convention. Our judgemental formulation is more faithful to the mechanization and cleaner for our purposes, because we are distinguishing several classes of final forms.

[8] We say "mark", rather than the more conventional "hole", to avoid confusion with the (orthogonal) holes of Hazelnut Live.

$\boxed{\tau \text{ ground}}$ τ is a ground type

$$\text{GBase}\quad \frac{}{b \text{ ground}} \qquad \text{GHole}\quad \frac{}{\llparenthesis\rrparenthesis \to \llparenthesis\rrparenthesis \text{ ground}}$$

$\boxed{\tau \blacktriangleright_{\text{ground}} \tau'}$ τ has matched ground type τ'

$$\text{MGArr}\quad \frac{\tau_1 \to \tau_2 \neq \llparenthesis\rrparenthesis \to \llparenthesis\rrparenthesis}{\tau_1 \to \tau_2 \blacktriangleright_{\text{ground}} \llparenthesis\rrparenthesis \to \llparenthesis\rrparenthesis}$$

Fig. 9. Ground Types

$\boxed{d \text{ final}}$ d is final

$$\text{FBoxedVal}\quad \frac{d \text{ boxedval}}{d \text{ final}} \qquad \text{FIndet}\quad \frac{d \text{ indet}}{d \text{ final}}$$

$\boxed{d \text{ val}}$ d is a value

$$\text{VConst}\quad \frac{}{c \text{ val}} \qquad \text{VLam}\quad \frac{}{\lambda x{:}\tau.d \text{ val}}$$

$\boxed{d \text{ boxedval}}$ d is a boxed value

$$\text{BVVal}\quad \frac{d \text{ val}}{d \text{ boxedval}} \qquad \text{BVArrCast}\quad \frac{\tau_1 \to \tau_2 \neq \tau_3 \to \tau_4 \qquad d \text{ boxedval}}{d\langle \tau_1 \to \tau_2 \Rightarrow \tau_3 \to \tau_4\rangle \text{ boxedval}} \qquad \text{BVHoleCast}\quad \frac{d \text{ boxedval} \qquad \tau \text{ ground}}{d\langle \tau \Rightarrow \llparenthesis\rrparenthesis\rangle \text{ boxedval}}$$

$\boxed{d \text{ indet}}$ d is indeterminate

$$\text{IEHole}\quad \frac{}{\llparenthesis\rrparenthesis^u_\sigma \text{ indet}} \qquad \text{INEHole}\quad \frac{d \text{ final}}{\llparenthesis d\rrparenthesis^u_\sigma \text{ indet}} \qquad \text{IAp}\quad \frac{d_1 \neq d_1'\langle \tau_1 \to \tau_2 \Rightarrow \tau_3 \to \tau_4\rangle \qquad d_1 \text{ indet} \qquad d_2 \text{ final}}{d_1(d_2) \text{ indet}}$$

$$\text{ICastGroundHole}\quad \frac{d \text{ indet} \qquad \tau \text{ ground}}{d\langle \tau \Rightarrow \llparenthesis\rrparenthesis\rangle \text{ indet}} \qquad \text{ICastHoleGround}\quad \frac{d \neq d'\langle \tau' \Rightarrow \llparenthesis\rrparenthesis\rangle \qquad d \text{ indet} \qquad \tau \text{ ground}}{d\langle \llparenthesis\rrparenthesis \Rightarrow \tau\rangle \text{ indet}}$$

$$\text{ICastArr}\quad \frac{\tau_1 \to \tau_2 \neq \tau_3 \to \tau_4 \qquad d \text{ indet}}{d\langle \tau_1 \to \tau_2 \Rightarrow \tau_3 \to \tau_4\rangle \text{ indet}} \qquad \text{IFailedCast}\quad \frac{d \text{ final} \qquad \tau_1 \text{ ground} \qquad \tau_2 \text{ ground} \qquad \tau_1 \neq \tau_2}{d\langle \tau_1 \Rightarrow \llparenthesis\rrparenthesis \not\Rightarrow \tau_2\rangle \text{ indet}}$$

Fig. 10. Final Forms

an eager, left-to-right evaluation strategy, or *excluded* to leave the evaluation strategy and order unspecified. In our metatheory, we exclude these premises, both for the sake of generality, and to support the specification of the fill-and-resume operation (see Sec. 4).

Substitution, written $[d/x]d'$, operates in the standard capture-avoiding manner [Harper 2016] (see Appendix A). The only cases of special interest arise when substitution reaches a hole closure:

$$\begin{aligned} [d/x]\llparenthesis\rrparenthesis^u_\sigma &= \llparenthesis\rrparenthesis^u_{[d/x]\sigma} \\ [d/x]\llparenthesis d'\rrparenthesis^u_\sigma &= \llparenthesis [d/x]d'\rrparenthesis^u_{[d/x]\sigma} \end{aligned}$$

In both cases, we write $[d/x]\sigma$ to perform substitution on each expression in the hole environment σ, i.e. the environment "records" the substitution. For example, $(\lambda x{:}b.\lambda y{:}b.\llparenthesis\rrparenthesis^u_{[x/x,y/y]})(c) \mapsto \lambda y{:}b.\llparenthesis\rrparenthesis^u_{[c/x,y/y]}$. Beta reduction can duplicate hole closures. Consequently, the environments of different closures with the same hole name may differ, e.g., when an reduction applies a function with a hole closure body multiple times as in Fig. 1. Hole closures may also appear within the environments of other hole closures, giving rise to the closure paths described in Sec. 2.2.

$\boxed{d \longrightarrow d'}$ d takes an instruction transition to d'

$$\texttt{ITBeta}\quad \frac{[d_2 \text{ final}]}{(\lambda x{:}\tau.d_1)(d_2) \longrightarrow [d_2/x]d_1} \qquad \texttt{ITApCast}\quad \frac{[d_1 \text{ final}] \qquad [d_2 \text{ final}] \qquad \tau_1 \to \tau_2 \neq \tau_1' \to \tau_2'}{d_1\langle \tau_1 \to \tau_2 \Rightarrow \tau_1' \to \tau_2'\rangle(d_2) \longrightarrow (d_1(d_2\langle \tau_1' \Rightarrow \tau_1\rangle))\langle \tau_2 \Rightarrow \tau_2'\rangle}$$

$$\texttt{ITCastId}\quad \frac{[d \text{ final}]}{d\langle \tau \Rightarrow \tau\rangle \longrightarrow d} \qquad \texttt{ITCastSucceed}\quad \frac{[d \text{ final}] \qquad \tau \text{ ground}}{d\langle \tau \Rightarrow \llparenthesis\rrparenthesis \Rightarrow \tau\rangle \longrightarrow d} \qquad \texttt{ITCastFail}\quad \frac{[d \text{ final}] \qquad \tau_1 \neq \tau_2 \qquad \tau_1 \text{ ground} \qquad \tau_2 \text{ ground}}{d\langle \tau_1 \Rightarrow \llparenthesis\rrparenthesis \Rightarrow \tau_2\rangle \longrightarrow d\langle \tau_1 \Rightarrow \llparenthesis\rrparenthesis \not\Rightarrow \tau_2\rangle}$$

$$\texttt{ITGround}\quad \frac{[d \text{ final}] \qquad \tau \blacktriangleright_{\text{ground}} \tau'}{d\langle \tau \Rightarrow \llparenthesis\rrparenthesis\rangle \longrightarrow d\langle \tau \Rightarrow \tau' \Rightarrow \llparenthesis\rrparenthesis\rangle} \qquad \texttt{ITExpand}\quad \frac{[d \text{ final}] \qquad \tau \blacktriangleright_{\text{ground}} \tau'}{d\langle \llparenthesis\rrparenthesis \Rightarrow \tau\rangle \longrightarrow d\langle \llparenthesis\rrparenthesis \Rightarrow \tau' \Rightarrow \tau\rangle}$$

Fig. 11. Instruction Transitions

$$\text{EvalCtx} \quad \mathcal{E} \;::=\; \circ \mid \mathcal{E}(d) \mid d(\mathcal{E}) \mid \llparenthesis\mathcal{E}\rrparenthesis^u_\sigma \mid \mathcal{E}\langle \tau \Rightarrow \tau\rangle \mid \mathcal{E}\langle \tau \Rightarrow \llparenthesis\rrparenthesis \not\Rightarrow \tau\rangle$$

$\boxed{d = \mathcal{E}\{d'\}}$ d is obtained by placing d' at the mark in $\mathcal{E}$

$$\texttt{FHOuter}\quad \frac{}{d = \circ\{d\}} \qquad \texttt{FHAp1}\quad \frac{d_1 = \mathcal{E}\{d_1'\}}{d_1(d_2) = \mathcal{E}(d_2)\{d_1'\}} \qquad \texttt{FHAp2}\quad \frac{[d_1 \text{ final}] \qquad d_2 = \mathcal{E}\{d_2'\}}{d_1(d_2) = d_1(\mathcal{E})\{d_2'\}} \qquad \texttt{FHNEHoleInside}\quad \frac{d = \mathcal{E}\{d'\}}{\llparenthesis d\rrparenthesis^u_\sigma = \llparenthesis\mathcal{E}\rrparenthesis^u_\sigma\{d'\}}$$

$$\texttt{FHCastInside}\quad \frac{d = \mathcal{E}\{d'\}}{d\langle \tau_1 \Rightarrow \tau_2\rangle = \mathcal{E}\langle \tau_1 \Rightarrow \tau_2\rangle\{d'\}} \qquad \texttt{FHFailedCast}\quad \frac{d = \mathcal{E}\{d'\}}{d\langle \tau_1 \Rightarrow \llparenthesis\rrparenthesis \not\Rightarrow \tau_2\rangle = \mathcal{E}\langle \tau_1 \Rightarrow \llparenthesis\rrparenthesis \not\Rightarrow \tau_2\rangle\{d'\}}$$

$\boxed{d \mapsto d'}$ d steps to d'

$$\texttt{Step}\quad \frac{d = \mathcal{E}\{d_0\} \qquad d_0 \longrightarrow d_0' \qquad d' = \mathcal{E}\{d_0'\}}{d \mapsto d'}$$

Fig. 12. Evaluation Contexts and Steps

The ITBeta rule is not the only rule we need to handle function application, because lambdas are not the only final form of arrow type. Two other situations may also arise.

First, the expression in function position might be a cast between arrow types, in which case we apply the arrow cast conversion rule, Rule `ITApCast`, to rewrite the application form, obtaining an equivalent application where the expression d_1 under the function cast is exposed. We know from inverting the typing rules that d_1 has type $\tau_1 \to \tau_2$, and that d_2 has type τ_1', where $\tau_1 \sim \tau_1'$. Consequently, we maintain type safety by placing a cast on d_2 from τ_1' to τ_1. The result of this application has type τ_2, but the original cast promised that the result would have consistent type τ_2', so we also need a cast on the result from τ_2 to τ_2'.

Second, the expression in function position may be indeterminate, where arrow cast conversion is not applicable, e.g. $(\llparenthesis\rrparenthesis^u_\sigma)(c)$. In this case, the application is indeterminate (Rule `IAp` in Fig. 10), and the application reduces no further.

3.3.2 Casts. Rule ITCastId strips identity casts. The remaining instruction transition rules assign meaning to non-identity casts. As discussed in Sec. 3.2.2, the structure of a term cast *to* hole type is statically obscure, so we must await a *use* of the term at some other type, via a cast *away* from hole type, to detect the type error dynamically. Rules ITCastSucceed and ITCastFail handle this situation when the two types involved are ground types (Fig. 9). If the two ground types are equal, then the cast succeeds and the cast may be dropped. If they are not equal, then the cast fails and the failed cast form, $d\langle\tau_1 \Rightarrow (\!|\,|\!) \nRightarrow \tau_2\rangle$, arises. Rule TAFailedCast specifies that a failed cast is well-typed exactly when d has ground type τ_1 and τ_2 is a ground type disequal to τ_1. Rule IFailedCast specifies that a failed cast operates as an indeterminate form (once d is final), i.e. evaluation does not stop. For simplicity, we do not include blame labels as found in some accounts of gradual typing [Siek et al. 2015], but it would be straightforward to do so by recording the blame labels from the two constituent casts on the two arrows of the failed cast form.

Rules ITCastSucceed and ITCastFail rules just described only operate at ground type. The two remaining instruction transition rules, Rule ITGround and ITExpand, insert intermediate casts from non-ground type to a consistent ground type, and *vice versa*. These rules serve as technical devices, permitting us to restrict our interest exclusively to casts involving ground types and type holes elsewhere. Here, the only non-ground types are the arrow types, so the grounding judgement $\tau_1 \blacktriangleright_{\mathsf{ground}} \tau_2$ (Fig. 9), produces the ground arrow type $(\!|\,|\!) \rightarrow (\!|\,|\!)$. More generally, the following invariant governs this judgement.

Lemma 3.8 (Grounding). *If* $\tau_1 \blacktriangleright_{\mathsf{ground}} \tau_2$ *then* τ_2 ground *and* $\tau_1 \sim \tau_2$ *and* $\tau_1 \neq \tau_2$.

In all other cases, casts evaluate either to boxed values or to indeterminate forms according to the remaining rules in Fig. 10. Of note, Rule ICastHoleGround handles casts from hole to ground type that are not of the form $d\langle\tau_1 \Rightarrow (\!|\,|\!) \Rightarrow \tau_2\rangle$. The need for this case is perhaps not obvious because in the cast calculus, the only irreducible term of type $(\!|\,|\!)$ is $d\langle\tau_1 \Rightarrow (\!|\,|\!)\rangle$.

3.3.3 Type Safety. The purpose of establishing type safety is to ensure that the static and dynamic semantics of a language cohere. We follow the approach developed by Wright and Felleisen [1994], now standard [Harper 2016], which distinguishes two type safety properties, preservation and progress. To permit the evaluation of incomplete programs, we establish these properties for terms typed under arbitrary hole context Δ. We assume an empty typing context, Γ; to run open programs, the system may treat free variables as empty holes with a corresponding name.

The preservation theorem establishes that transitions preserve type assignment, i.e. that the type of an expression accurately predicts the type of the result of reducing that expression.

Theorem 3.9 (Preservation). *If* $\Delta; \emptyset \vdash d : \tau$ *and* $d \mapsto d'$ *then* $\Delta; \emptyset \vdash d' : \tau$.

The proof relies on an analogous preservation lemma for instruction transitions and a standard substitution lemma stated in Appendix A. Hole closures can disappear during evaluation, so we rely on simple weakening of Δ.

The progress theorem establishes that the dynamic semantics accounts for every well-typed term, i.e. that we have not forgotten some necessary rules or premises.

Theorem 3.10 (Progress). *If* $\Delta; \emptyset \vdash d : \tau$ *then either (a)* $d \mapsto d'$ *or (b)* d boxedval *or (c)* d indet.

The key to establishing the progress theorem under a non-empty hole context is to explicitly account for indeterminate forms, i.e. those rooted at either a hole closure or a failed cast. The proof relies on canonical forms lemmas stated in Appendix A.

3.3.4 Complete Programs. Although this paper focuses on running *incomplete* programs, it helps to know that the necessary machinery does not interfere with running *complete* programs, i.e. those with no type or expression holes. Appendix A defines the predicates e complete, τ complete,

Γ complete and d complete. Of note, failed casts cannot appear in complete internal expressions. The following theorem establishes that expansion preserves program completeness.

Theorem 3.11 (Complete Expansion).

(1) If Γ complete *and* e complete *and* $\Gamma \vdash e \Rightarrow \tau \leadsto d \dashv \Delta$ *then* τ complete *and* d complete.

(2) If Γ complete *and* e complete *and* τ complete *and* $\Gamma \vdash e \Leftarrow \tau \leadsto d : \tau' \dashv \Delta$ *then* d complete.

The following preservation theorem establishes that stepping preserves program completeness.

Theorem 3.12 (Complete Preservation). *If* $\Delta; \emptyset \vdash d : \tau$ *and* d complete *and* $d \mapsto d'$ *then* $\Delta; \emptyset \vdash d' : \tau$ *and* d' complete.

The following progress theorem establishes that evaluating a complete program always results in classic values, not boxed values nor indeterminate forms.

Theorem 3.13 (Complete Progress). *If* $\Delta; \emptyset \vdash d : \tau$ *and* d complete *then either* $d \mapsto d'$ *or* d val.

3.4 Agda Mechanization

The supplemental material includes our Agda mechanization [Aydemir et al. 2005; Norell 2007, 2009] of the semantics and metatheory of Hazelnut Live, including all of the theorems stated above and necessary lemmas. We take as assumptions only a few standard properties of general hypothetical judgements and finite maps, and, to avoid excessive machinery related to hole name tracking in the bidirectional typing rules, we axiomatize some hole name uniqueness assumptions made implicitly in this paper. Our approach is otherwise standard: we model judgements as inductive datatypes, and rules as dependently typed constructors of these judgements. We adopt Barendregt's convention for bound variables [Barendregt 1984; Urban et al. 2007] and hole names and encode typing contexts, hole contexts and finite substitutions using metafunctions. To support this encoding choice, we postulate function extensionality (which is independent of Agda's axioms) [Awodey et al. 2012]. The documentation provided with the mechanization has more technical details.

3.5 Implementation

The supplemental material includes a browser-based implementation of Hazelnut Live. All of the live programming features from Sec. 2 are available in the implementation essentially as shown, but for this more austere language (with some minor conveniences, notably let binding). Appendix C provides screenshots of the full user interface.

The implementation is written with the Reason toolchain for OCaml [Leroy 2003; Reason Team 2017] together with the OCaml `React` library [Bünzli 2018] and the `js_of_ocaml` compiler and its associated libraries [Vouillon and Balat 2014]. This follows the implementation of Hazel, which, although tracking toward an Elm-like semantics, is implemented in OCaml because the Elm compiler is not yet self-hosted. The implementation of the dynamic semantics consists of a simple evaluator that closely follows the rules specified in this section.

The editor component of the implementation is derived from the structure editor calculus of Hazelnut, but with support for more natural cursor-based movement and infix operator sequences (the details are beyond the scope of this paper). It exposes a language of structured edit actions (summarized in the sidebar on the left of the screenshots in Appendix C) that automatically inserts holes as necessary to guarantee that every edit state has some (possibly incomplete) type . This corresponds to the Sensibility invariant established by Omar et al. [2017a]. By composing this metatheoretic property with the Expandability, Typed Expansion and Progress properties from this paper, we establish a uniquely powerful invariant: *every* edit state has a *static meaning* (a la Hazelnut) and a *dynamic meaning* (a la Hazelnut Live). In other words, live feedback never "flickers in and out" nor reflects "stale" edit states. The implementation is intended as a proof-of-concept of this invariant, and as a companion to this paper.

$\boxed{\llbracket d/u \rrbracket d' = d''}$ d'' is obtained by filling hole u with d in d'

$$\begin{array}{lcll}
\llbracket d/u \rrbracket c & = & c & \\
\llbracket d/u \rrbracket x & = & x & \\
\llbracket d/u \rrbracket \lambda x{:}\tau.d' & = & \lambda x{:}\tau.\llbracket d/u \rrbracket d' & \\
\llbracket d/u \rrbracket d_1(d_2) & = & (\llbracket d/u \rrbracket d_1)(\llbracket d/u \rrbracket d_2) & \\
\llbracket d/u \rrbracket \llparenthesis\rrparenthesis^{u}_{\sigma} & = & [\llbracket d/u \rrbracket \sigma]d & \\
\llbracket d/u \rrbracket \llparenthesis\rrparenthesis^{v}_{\sigma} & = & \llparenthesis\rrparenthesis^{v}_{\llbracket d/u \rrbracket \sigma} & \text{when } u \neq v \\
\llbracket d/u \rrbracket \llparenthesis d' \rrparenthesis^{u}_{\sigma} & = & [\llbracket d/u \rrbracket \sigma]d & \\
\llbracket d/u \rrbracket \llparenthesis d' \rrparenthesis^{v}_{\sigma} & = & \llparenthesis \llbracket d/u \rrbracket d' \rrparenthesis^{v}_{\llbracket d/u \rrbracket \sigma} & \text{when } u \neq v \\
\llbracket d/u \rrbracket d'\langle \tau \Rightarrow \tau' \rangle & = & (\llbracket d/u \rrbracket d')\langle \tau \Rightarrow \tau' \rangle & \\
\llbracket d/u \rrbracket d'\langle \tau_1 \Rightarrow \llparenthesis\rrparenthesis \not\Rightarrow \tau_2 \rangle & = & (\llbracket d/u \rrbracket d')\langle \tau_1 \Rightarrow \llparenthesis\rrparenthesis \not\Rightarrow \tau_2 \rangle &
\end{array}$$

Fig. 13. Hole Filling

4 A Contextual Modal Interpretation of Fill-and-Resume

When the programmer performs one or more edit actions to fill in a hole in the program, a new result must be computed, ideally quickly [Tanimoto 1990, 2013]. Naïvely, the system would need to compute the result "from scratch" on each such edit. For small exploratory programming tasks, recomputation is acceptable, but in cases where a large amount of computation might occur, e.g. in data science tasks, a more efficient approach is to resume evaluation from where it left off after an edit that amounts to hole filling. This section briefly discusses the foundations of this feature, which we call *fill-and-resume*. This approach is complementary to incremental computing techniques [Hammer et al. 2014].

Formally, the key idea is to interpret hole environments as *delayed substitutions*. This is the same interpretation suggested for closures in contextual modal type theory (CMTT) by Nanevski et al. [2008]. Fig. 13 defines the hole filling operation $\llbracket d/u \rrbracket d'$ based on the contextual substitution operation of CMTT. Unlike usual notions of capture-avoiding substitution, hole filling imposes no condition on the binder when passing into the body of a lambda expression—the expression that fills a hole can, of course, refer to variables in scope where the hole appears. When hole filling encounters an empty closure for the hole being instantiated, $\llbracket d/u \rrbracket \llparenthesis\rrparenthesis^{u}_{\sigma}$, the result is $[\llbracket d/u \rrbracket \sigma]d$. That is, we apply the delayed substitution to the fill expression d after first recursively filling any instances of hole u in σ. Hole filling for non-empty closures is analogous, where it discards the previously-enveloped expression. This case shows why we cannot interpret a non-empty hole as an empty hole of arrow type applied to the enveloped expression—the hole filling operation would not operate as expected under this interpretation.

The following theorem characterizes the static behavior of hole filling.

Theorem 4.1 (Filling). *If* $\Delta, u :: \tau'[\Gamma']; \Gamma \vdash d : \tau$ *and* $\Delta; \Gamma' \vdash d' : \tau'$ *then* $\Delta; \Gamma \vdash \llbracket d'/u \rrbracket d : \tau$.

Dynamically, the correctness of fill-and-resume depends on the following *commutativity* property: if there is some sequence of steps that go from d_1 to d_2, then one can fill a hole in these terms at *either* the beginning or at the end of that step sequence. We write $d_1 \mapsto^* d_2$ for the reflexive, transitive closure of stepping (see Fig. 15 in Appendix A).

Theorem 4.2 (Commutativity). *If* $\Delta, u :: \tau'[\Gamma']; \emptyset \vdash d_1 : \tau$ *and* $\Delta; \Gamma' \vdash d' : \tau'$ *and* $d_1 \mapsto^* d_2$ *then* $\llbracket d'/u \rrbracket d_1 \mapsto^* \llbracket d'/u \rrbracket d_2$.

Critically, this property relies on the more version of the semantics from Sec. 3 where evaluation order is unspecified (i.e. formally, we omit the bracketed premises). In general, resuming from $\llbracket d'/u \rrbracket d_2$ will not reduce sub-expressions in the same order as a "fresh" left-to-right reduction sequence starting from $\llbracket d'/u \rrbracket d_1$. In other words, this notion of fill-and-resume only works for languages where evaluation order “does not matter”. (There are various standard ways to formalize this intuition. For the sake of space, we review these in Appendix A.6.)

We describe the proof, which involves a number of lemmas and definitions, in Appendix A.5. In particular, care is needed to handle the situation where a now-filled non-empty hole had taken a step in the original evaluation.

We do not separately define hole filling in the external language (i.e. we consider a change to an external expression to be a hole filling if the new expansion differs from the previous expansion up to hole filling). In practice, it may be useful to cache more than one recent edit state to take full advantage of hole filling. As an example, consider two edits, the first filling a hole u with the number 2, and the next applying operator +, resulting in $2 + (\!|\,|\!)^{v}_{\sigma}$. This second edit is not a hole filling edit with respect to the immediately preceding edit state, 2, but it can be understood as filling hole u from two states back with $2 + (\!|\,|\!)^{v}_{\sigma}$.

Hole filling also allows us to give a contextual modal interpretation to lab notebook cells like those of Jupyter/IPython [Pérez and Granger 2007] (and read-eval-print loops as a restricted case where edits to previous cells are impossible). Each cell can be understood as a series of **let** bindings ending implicitly in a hole, which is filled by the next cell. The live environment in the subsequent cell is exactly the hole environment of this implicit trailing hole. Hole filling when a subsequent cell changes avoids recomputing the environment from preceding cells, without relying on mutable state. Commutativity provides a reproducibility guarantee missing from Jupyter/IPython, where editing and executing previous cells can cause the state to differ substantially from the state that would result when attempting to run the notebook from the top.

5 Related and Future Work

This paper defined a dynamic semantics for incomplete functional programs based directly on the static semantics developed by Omar et al. [2017a]. A subsequent “vision paper” introducing Hazel suggested the need for a corresponding dynamic semantics for the purposes of live programming [Omar et al. 2017b]. This paper contributes the type-theoretic first steps necessary to achieve this long-term vision.

The semantics borrows substantially from the theory of type consistency and casts developed in gradual type theory [Siek and Taha 2006; Siek et al. 2015], as discussed at length in Sec. 3. The main innovation relative to this prior work is the treatment of cast failures as holes, rather than errors. Many of the methods developed to make gradual typing more expressive and practical are directly relevant to future directions for Hazel and other implementations of the ideas herein. For example, there has been substantial work on the problem of implementing casts efficiently, e.g. by Herman et al. [2010], and on integrating gradual typing with polymorphism, e.g. by [Garcia and Cimini 2015b]. Another interesting direction that we leave to future work is the integration of these ideas into dependently typed proof assistants, where evaluation is at the service of equational reasoning.

Another important future direction is to move beyond pure functional programming a la Elm and carefully integrate imperative features, e.g. ML-style references. These require particular care, as recognized by [Siek and Taha 2006], though we see no reason why the type safety properties established in this paper would not be conserved in an effectful setting. Going beyond references to incorporate general effects, e.g. network and IO effects, raises some important practical concerns, however—we do not want to continue past a hole or error and in so doing haphazardly trigger an unintended effect. Developing language features and user interface features that give the programmer fine-grained control over evaluation when it encounters a hole or cast failure are likely to be particularly helpful in these settings.

The other major pillar of related work is the work on contextual modal type theory (CMTT) [Nanevski et al. 2008], which we also discussed at length throughout the paper. To reiterate, there is a close relationship between holes in this paper and metavariables in CMTT. Hole contexts correspond to modal contexts. Empty hole closures relate to the concept of a *metavariable closure* in CMTT, which consists of a metavariable paired with a substitution for all of the variables in the typing context associated with that metavariable. Hole filling relates to contextual substitution.

Although these connections are encouraging, our contributions do not neatly fall out from the prior work. The problem is first that Nanevski et al. [2008] defined only the logical reductions for CMTT, viewing it as a proof system for intuitionistic contextual modal logic via the propositions-as-types (Curry-Howard) principle. The paper therefore proved only a subject reduction property (which is closely related to type preservation). This is not a full dynamic semantics, and in particular, there is no notion of *progress*, i.e. that well-typed terms cannot get "stuck" in an undefined state [Wright and Felleisen 1994]. In any case, a conventional dynamic semantics for CMTT would not be immediately relevant to our goal of evaluating incomplete programs because, by our interpretation of hole closures, we would need a dynamic semantics for terms with free metavariables. Nanevski et al. [2008] sketched an interpretation of CMTT into the simply-typed lambda calculus with sums under permutation conversion[9], which has been studied by de Groote [2002], but under this interpretation an analogous problem arises—metavariables become variables of a function type, so again we cannot rely on the standard notion of progress on closed terms.

It is also worth emphasizing that we use the machinery borrowed from CMTT extralinguistically. However, a key feature CMTT that we have not yet touched on is the *internalization* of metavariable binding and contextual substitution via the contextual modal types, $[\Gamma]\tau$, which are introduced by the operation $\mathsf{box}(\Gamma.d)$ and eliminated by the operation $\mathsf{letbox}(d_1, u.d_2)$. A hole filling can be interpreted as an expression of contextual modal type, and the act of hole filling followed by evaluation to the next possibly-indeterminate edit state as evaluation under the binder of a suitable letbox construct, which is enabled by the dynamic semantics in Sec. 3. This interpretation allows us to *compute* hole fillings, rather than simply stating them, by specifying non-trivial expressions of modal type. This could serve as the basis for a *live* computational hole refinement system, extending the capabilities of purely static hole refinement systems like those available in some proof assistants, e.g. the elaborator reflection system of Idris [Brady 2013; Christiansen and Brady 2016] and the refinement system of Beluga [Pientka 2010; Pientka and Cave 2015]. Each applied hole filling serves as a boundary between dynamic *edit stages*. This contextual modal interpretation of live staged hole refinement nicely mirrors the modal interpretation of staged argument evaluation [Davies and Pfenning 2001].

Indeed, the connection to CMTT neatly explains the difference between our proposed dynamics and existing work on staging and partial evaluation—that existing work is rooted in modal type theory [Davies and Pfenning 2001] rather than the derivative contextual modal type theory (which, to date, has been studied far less extensively.) In other words, existing systems are focused on partial evaluation with respect to an input that sits outside of a function, whereas holes can appear in context, directly within the body of a function.

There are various other systems similar in various ways to CMTT in that they consider the problem of reasoning about metavariables. For example, McBride's OLEG is another system for reasoning contextually about metavariables [McBride 2000], and it is the basis of certain analogous features of Idris [Brady 2013]. Geuvers and Jojgov [2002] discuss similar ideas. Work on explicit substitutions also confronts many related technical subtleties [Aba 1991; Abadi et al. 1990; Lévy

[9]Permutation conversions are necessary to encode the commuting reductions of CMTT, which in turn are necessary to prove a strong normalization property. These issues are not relevant in Hazelnut Live because, as in the gradually typed lambda calculus, type holes admit non-termination: we can express the Y combinator as $(\lambda x{:}(\!|\,|\!).x(x))(\lambda x{:}(\!|\,|\!).x(x))$.

and Maranget 1999]. CMTT is unique relative to prior approaches in that it was designed with commutativity properties in mind (though the theory was not fully developed).

Note that the commutativity property we establish will not hold for a language that supports non-commutative effects. We leave to future work the task of defining more restricted special cases of commutativity in effectful settings.

Holes play a prominent role in structure editors, and indeed the prior work on Hazelnut was primarily motivated by this application. Most work on structure editors has focused on the user interfaces that they present. This is important work—presenting a fluid user interface involving exclusively structural edit actions is a non-trivial problem that has not yet been fully resolved, though recent studies have started to show productivity gains for keyboard-driven structure editors [Asenov and Müller 2014; Voelter et al. 2014]. This work, together with the end-to-end well-definedness guarantee described in Sec. 3.5, suggests that structure editing together with live programming might be the foundation for a "next-generation" programming experience. We emphasize, however, that our proposed contributions will be relevant no matter how holes come to be inserted into the program.

Understanding and debugging static type errors is notoriously difficult, particularly for novices. A variety of approaches have been proposed [Chen and Erwig 2014; Lerner et al. 2006; Pavlinovic et al. 2015; Zhang et al. 2017] to better localize and explain type errors. One of these approaches [Seidel et al. 2016] proposes to use program synthesis techniques to generate a dynamic witness that demonstrates a run-time failure, and then display a compressed execution trace to the user as a graph. Hazelnut Live has similar motivations in that it can run programs with type errors (i.e., programs with non-empty holes) as far as possible, in particular, until they would go wrong, and provide meaningful feedback. However, no attempt is made to generate examples that do not already appear in the program. Combining the strengths of these approaches may be fruitful in the future.

More generally, expression holes also often appear in the context of program synthesis, serving as placeholders in *templates* [Srivastava et al. 2013] or *sketches* [Solar-Lezama 2009] to be filled in by an expression synthesis engine. We leave to future work the possibility of combining these approaches, i.e. using the information generated by running an incomplete program to inform the generation of program and edit action suggestions.

This approach is reminiscent of the workflow that debuggers make available using breakpoints [Fitzgerald et al. 2008; Tolmach and Appel 1995] and various logging and tracing capabilities. The difference here is twofold. First, evaluation does not stop at each breakpoint and so it is straightforward to explore the *space* of values that a variable takes. Second, breakpoints make available the values of a variable at a position in the evaluation trace. Hole closures, on the other hand, convey information from a syntactic position in the result of evaluation. The result is, by nature, a simpler object than the full evaluation trace that produced the result. These approaches are fundamentally complementary, rather than opposed.

Hole-like constructs also appear in work on program slicing [Perera et al. 2012], where holes are used as a technical device to determine which parts of the program that do not impact a selected part of the result.

6 Conclusion

To conclude, we quote Weinberg from The Psychology of Computer Programming (1998): "how truly sad it is that just at the very moment when the computer has something important to tell us, it starts speaking gibberish."

REFERENCES

1991. Explicit Substitutions. *Journal of Functional Programming* 1, 4 (1991), 375–416.

M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Levy. 1990. Explicit substitutions. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 31–46.

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)*.

Dimitar Asenov and Peter Müller. 2014. 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*.

Steve Awodey, Nicola Gambino, and Kristina Sojakova. 2012. Inductive types in homotopy type theory. In *Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science*. IEEE Computer Society, 95–104.

Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn Stephanie Weirich, and Steve Zdancewic. 2005. Mechanized Metatheory for the Masses: The POPLmark Challenge. (2005).

H.P. Barendregt. 1984. *The Lambda Calculus*. Vol. 103.

Tomasz Blanc, Jean-Jacques Lévy, and Luc Maranget. 2005. Sharing in the Weak Lambda-Calculus. In *Processes, Terms and Cycles: Steps on the Road to Infinity, Essays Dedicated to Jan Willem Klop, on the Occasion of His 60th Birthday (Lecture Notes in Computer Science)*, Aart Middeldorp, Vincent van Oostrom, Femke van Raamsdonk, and Roel C. de Vrijer (Eds.), Vol. 3838. Springer, 70–87. https://doi.org/10.1007/11601548_7

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

Daniel Bünzli. 2018. React / Erratique. (2018). http://erratique.ch/software/react/. Retrieved Apr. 7, 2018.

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)*.

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*.

Naim Çagman and J. Roger Hindley. 1998. Combinatory Weak Reduction in Lambda Calculus. *Theor. Comput. Sci.* 198, 1-2 (1998), 239–247. https://doi.org/10.1016/S0304-3975(97)00250-8

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.

Sheng Chen and Martin Erwig. 2014. Counter-Factual Typing for Debugging Type Errors. In *Symposium on Principles of Programming Languages (POPL)*.

Adam Chlipala, Leaf Petersen, and Robert Harper. 2005. Strict bidirectional type checking. In *TLDI '05: Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*. 71–78.

David Raymond Christiansen. 2013. Bidirectional Typing Rules: A Tutorial. http://davidchristiansen.dk/tutorials/bidirectional.pdf. (2013).

David R. Christiansen and Edwin Brady. 2016. Elaborator reflection: extending Idris in Idris. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 284–297. https://doi.org/10.1145/2951913.2951932

Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and Direct Manipulation, Together at Last. In *Conference on Programming Language Design and Implementation (PLDI)*.

Alonzo Church and J Barkley Rosser. 1936. Some properties of conversion. *Trans. Amer. Math. Soc.* 39, 3 (1936), 472–482.

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

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

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

Luis Damas and Robin Milner. 1982. Principal type-schemes for functional programs. In *POPL*.

Rowan Davies and Frank Pfenning. 2001. A modal analysis of staged computation. *J. ACM* 48, 3 (2001), 555–604.

Philippe de Groote. 2002. On the Strong Normalisation of Intuitionistic Natural Deduction with Permutation-Conversions. *Inf. Comput.* 178, 2 (2002), 441–464. https://doi.org/10.1006/inco.2002.3147

Joshua Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. 429–442. https://doi.org/10.1145/2500365.2500582

Matthias Felleisen and Robert Hieb. 1992. The Revised Report on the Syntactic Theories of Sequential Control and State. *Theor. Comput. Sci.* 103, 2 (1992), 235–271. https://doi.org/10.1016/0304-3975(92)90014-7

Francisco Ferreira and Brigitte Pientka. 2014. Bidirectional Elaboration of Dependently Typed Programs. In *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming, Kent, Canterbury, United Kingdom, September 8-10, 2014*, Olaf Chitil, Andy King, and Olivier Danvy (Eds.). ACM, 161–174. https://doi.org/10.1145/2643135.2643153

Sue Fitzgerald, Gary Lewandowski, Renee McCauley, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education* 18, 2 (2008), 93–116.

Ronald Garcia and Matteo Cimini. 2015a. Principal Type Schemes for Gradual Programs. In *POPL*.

Ronald Garcia and Matteo Cimini. 2015b. Principal type schemes for gradual programs. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 303–315.

Herman Geuvers and Gueorgui I. Jojgov. 2002. Open Proofs and Open Terms: A Basis for Interactive Logic. In *Computer Science Logic, 16th International Workshop, CSL 2002, 11th Annual Conference of the EACSL, Edinburgh, Scotland, UK, September 22-25, 2002, Proceedings (Lecture Notes in Computer Science)*, Julian C. Bradfield (Ed.), Vol. 2471. Springer, 537–552. https://doi.org/10.1007/3-540-45793-3_36

Adele Goldberg and David Robson. 1983. *Smalltalk-80: the language and its implementation.*

Susan L. Graham, Charles B. Haley, and William N. Joy. 1979. Practical LR Error Recovery. In *SIGPLAN Symposium on Compiler Construction (CC).*

Matthew A. Hammer, Yit Phang Khoo, Michael Hicks, and Jeffrey S. Foster. 2014. Adapton: composable, demand-driven incremental computation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*

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

Robert Harper and Christopher Stone. 2000. A Type-Theoretic Interpretation of Standard ML. In *Proof, Language and Interaction: Essays in Honour of Robin Milner.* MIT Press.

Brian Hempel and Ravi Chugh. 2016. Semi-Automated SVG Programming via Direct Manipulation. In *Symposium on User Interface Software and Technology (UIST).*

David Herman, Aaron Tomb, and Cormac Flanagan. 2010. Space-efficient gradual typing. *Higher-Order and Symbolic Computation* 23, 2 (2010), 167.

Gérard Huet. 1980. Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems: Abstract Properties and Applications to Term Rewriting Systems. *J. ACM* 27, 4 (1980), 797–821.

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).*

Lennart C. L. Kats and Eelco Visser. 2010. The spoofax language workbench: rules for declarative specification of languages and IDEs. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).*

Benjamin Lerner, Dan Grossman, and Craig Chambers. 2006. Seminal: Searching for ML Type-error Messages. In *Workshop on ML.*

Xavier Leroy. 2003. *The Objective Caml Documentation and User's Manual.*

Jean-Jacques Lévy and Luc Maranget. 1999. Explicit substitutions and programming languages. In *International Conference on Foundations of Software Technology and Theoretical Computer Science.* Springer, 181–200.

Conor McBride. 2000. *Dependently typed functional programs and their proofs.* Ph.D. Dissertation. University of Edinburgh, UK. http://hdl.handle.net/1842/374

Renee McCauley, Sue Fitzgerald, Gary Lewandowski, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: a review of the literature from an educational perspective. *Computer Science Education* 18, 2 (2008), 67–92.

Sean McDirmid. 2007. Living It Up with a Live Programming Language. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).*

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.

Martin Odersky, Christoph Zenger, and Matthias Zenger. 2001. Colored local type inference. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* 41–53.

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

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)*.

Zvonimir Pavlinovic, Tim King, and Thomas Wies. 2015. Practical SMT-Based Type Error Localization. In *International Conference on Functional Programming (ICFP)*.

Roly Perera, Umut A. Acar, James Cheney, and Paul Blain Levy. 2012. Functional programs that explain their work. In *ACM SIGPLAN International Conference on Functional Programming (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

Fernando Perez and Brian E. Granger. 2007. IPython: A System for Interactive Scientific Computing. *Computing in Science and Engg.* 9, 3 (May 2007), 21–29. https://doi.org/10.1109/MCSE.2007.53

Brigitte Pientka. 2010. Beluga: Programming with Dependent Types, Contextual Data, and Contexts. In *International Symposium on Functional and Logic Programming (FLOPS)*.

Brigitte Pientka and Andrew Cave. 2015. Inductive Beluga: Programming Proofs. In *International Conference on Automated Deduction*. Springer, 272–281.

Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press.

Benjamin C. Pierce and David N. Turner. 2000. Local type inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (2000), 1–44.

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

Reason Team. 2017. Reason Guide: What and Why? (2017). https://reasonml.github.io/guide/what-and-why/. Retrieved Nov. 14, 2017.

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 and Walid Taha. 2007. Gradual Typing for Objects. In *ECOOP 2007*, Vol. 4609. 2–27.

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

Armando Solar-Lezama. 2009. The Sketching Approach to Program Synthesis.. In *APLAS*, Vol. 5904. Springer, 4–13.

Saurabh Srivastava, Sumit Gulwani, and Jeffrey S Foster. 2013. Template-based program verification and program synthesis. *International Journal on Software Tools for Technology Transfer* 15, 5-6 (2013), 497–518.

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

Steven L. Tanimoto. 2013. A perspective on the evolution of live programming. In *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. https://doi.org/10.1017/S0956796800001313

Christian Urban, Stefan Berghofer, and Michael Norrish. 2007. Barendregt's Variable Convention in Rule Inductions. In *Conference on Automated Deduction (CADE)*. 16.

B Victor. 2012. Inventing on principle, Invited talk at the Canadian University Software Engineering Conference (CUSEC). (2012).

Markus Voelter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. 2014. Towards User-Friendly Projectional Editors. In *International Conference on Software Language Engineering (SLE)*.

Jérôme Vouillon and Vincent Balat. 2014. From bytecode to JavaScript: the Js_of_ocaml compiler. *Software: Practice and Experience* 44, 8 (2014), 951–972.

David Wakeling. 2007. Spreadsheet functional programming. *J. Funct. Program.* 17, 1 (2007), 131–143. https://doi.org/10.1017/S0956796806006186

Andrew K. Wright and Matthias Felleisen. 1994. A syntactic approach to type soundness. *Information and Computation* 115, 1 (1994), 38–94.

Y. S. Yoon and B. A. Myers. 2014. A longitudinal study of programmers' backtracking. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*.

Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2017. SHErrLoc: A Static Holistic Error Locator. *Transactions on Programming Languages and Systems (TOPLAS)* (2017).

A Additional Definitions for Hazelnut Live

A.1 Substitution

$\boxed{[d/x]d' = d''}$ d'' is obtained by substituting d for x in d'

$\boxed{[d/x]\sigma = \sigma'}$ σ' is obtained by substituting d for x in σ

$$\begin{array}{lcll}
[d/x]c & = & c & \\
[d/x]x & = & d & \\
[d/x]y & = & y & \text{when } x \neq y \\
[d/x]\lambda y{:}\tau.d' & = & \lambda y{:}\tau.[d/x]d' & \text{when } x \neq y \text{ and } y \notin \mathsf{FV}(d) \\
[d/x]d_1(d_2) & = & ([d/x]d_1)([d/x]d_2) & \\
[d/x]\llparenthesis\rrparenthesis^u_\sigma & = & \llparenthesis\rrparenthesis^u_{[d/x]\sigma} & \\
[d/x]\llparenthesis d'\rrparenthesis^u_\sigma & = & \llparenthesis [d/x]d'\rrparenthesis^u_{[d/x]\sigma} & \\
[d/x]d'\langle\tau_1 \Rightarrow \tau_2\rangle & = & ([d/x]d')\langle\tau_1 \Rightarrow \tau_2\rangle & \\
[d/x]d'\langle\tau_1 \Rightarrow \llparenthesis\rrparenthesis \not\Rightarrow \tau_2\rangle & = & ([d/x]d')\langle\tau_1 \Rightarrow \llparenthesis\rrparenthesis \not\Rightarrow \tau_2\rangle & \\
[d/x]\cdot & = & \cdot & \\
[d/x]\sigma, d/y & = & [d/x]\sigma, [d/x]d/y &
\end{array}$$

Lemma A.1 (Substitution).
(1) If $\Delta; \Gamma, x : \tau' \vdash d : \tau$ and $\Delta; \Gamma \vdash d' : \tau'$ then $\Delta; \Gamma \vdash [d'/x]d : \tau$.
(2) If $\Delta; \Gamma, x : \tau' \vdash \sigma : \Gamma'$ and $\Delta; \Gamma \vdash d' : \tau'$ then $\Delta; \Gamma \vdash [d'/x]\sigma : \Gamma'$.

Proof. By rule induction on the first assumption in each case. The conclusion follows from the definition of substitution in each case. □

A.2 Canonical Forms

Lemma A.2 (Canonical Value Forms). *If $\Delta; \emptyset \vdash d : \tau$ and d* val *then $\tau \neq \llparenthesis\rrparenthesis$ and*
(1) If $\tau = b$ then $d = c$.
(2) If $\tau = \tau_1 \to \tau_2$ then $d = \lambda x{:}\tau_1.d'$ where $\Delta; x : \tau_1 \vdash d' : \tau_2$.

Lemma A.3 (Canonical Boxed Forms). *If $\Delta; \emptyset \vdash d : \tau$ and d* boxedval *then*
(1) If $\tau = b$ then $d = c$.
(2) If $\tau = \tau_1 \to \tau_2$ then either
i. $d = \lambda x{:}\tau_1.d'$ where $\Delta; x : \tau_1 \vdash d' : \tau_2$, or
ii. $d = d'\langle\tau_1' \to \tau_2' \Rightarrow \tau_1 \to \tau_2\rangle$ where $\tau_1' \to \tau_2' \neq \tau_1 \to \tau_2$ and $\Delta; \emptyset \vdash d' : \tau_1' \to \tau_2'$.
(3) If $\tau = \llparenthesis\rrparenthesis$ then $d = d'\langle\tau' \Rightarrow \llparenthesis\rrparenthesis\rangle$ where τ' ground *and $\Delta; \emptyset \vdash d' : \tau'$.*

Lemma A.4 (Canonical Indeterminate Forms). *If $\Delta; \emptyset \vdash d : \tau$ and d* indet *then either*
(1) $d = \llparenthesis\rrparenthesis^u_\sigma$ and $u :: \tau[\Gamma'] \in \Delta$, or
(2) $d = \llparenthesis d'\rrparenthesis^u_\sigma$ and d' final *and $\Delta; \emptyset \vdash d' : \tau'$ and $u :: \tau[\Gamma'] \in \Delta$, or*
(3) $d = d_1(d_2)$ and $\Delta; \emptyset \vdash d_1 : \tau_2 \to \tau$ and $\Delta; \emptyset \vdash d_2 : \tau_2$ and d_1 indet *and d_2* final *and $d_1 \neq d_1\langle\tau_3 \to \tau_4 \Rightarrow \tau_3' \to \tau_4'\rangle$, or*
(4) $\tau = b$ and $d = d'\langle\llparenthesis\rrparenthesis \Rightarrow b\rangle$ and d' indet *and $d' \neq d''\langle\tau' \Rightarrow \llparenthesis\rrparenthesis\rangle$, or*
(5) $\tau = b$ and $d = d'\langle\tau' \Rightarrow \llparenthesis\rrparenthesis \not\Rightarrow b\rangle$ and τ' ground *and $\tau' \neq b$ and $\Delta; \emptyset \vdash d' : \tau'$, or*
(6) $\tau = \tau_{11} \to \tau_{12}$ and $d = d'\langle\tau_1 \to \tau_2 \Rightarrow \tau_{11} \to \tau_{12}\rangle$ and d' indet *and $\tau_1 \to \tau_2 \neq \tau_{11} \to \tau_{12}$, or*
(7) $\tau = \llparenthesis\rrparenthesis \to \llparenthesis\rrparenthesis$ and $d = d'\langle\llparenthesis\rrparenthesis \Rightarrow \llparenthesis\rrparenthesis \Rightarrow \llparenthesis\rrparenthesis\rangle$ and d' indet *and $d' \neq d''\langle\tau' \Rightarrow \llparenthesis\rrparenthesis\rangle$, or*
(8) $\tau = \llparenthesis\rrparenthesis \to \llparenthesis\rrparenthesis$ and $d = d'\langle\tau' \Rightarrow \llparenthesis\rrparenthesis \not\Rightarrow \llparenthesis\rrparenthesis \to \llparenthesis\rrparenthesis\rangle$ and $\tau' \neq \tau$ and τ' ground *and d'* indet *and $\Delta; \emptyset \vdash d' : \tau'$, or*
(9) $\tau = \llparenthesis\rrparenthesis$ and $d = d'\langle\tau' \Rightarrow \llparenthesis\rrparenthesis\rangle$ and τ' ground *and d'* indet.

The proofs for all three of these theorems follow by straightforward rule induction.

A.3 Complete Programs

$\boxed{\tau \text{ complete}}$ τ is complete

$$\texttt{TCBase}\ \frac{}{b \text{ complete}} \qquad \texttt{TCArr}\ \frac{\tau_1 \text{ complete} \qquad \tau_2 \text{ complete}}{\tau_1 \to \tau_2 \text{ complete}}$$

$\boxed{e \text{ complete}}$ e is complete

$$\texttt{ECVar}\ \frac{}{x \text{ complete}} \qquad \texttt{ECConst}\ \frac{}{c \text{ complete}} \qquad \texttt{ECLam1}\ \frac{\tau \text{ complete} \qquad e \text{ complete}}{\lambda x{:}\tau.e \text{ complete}} \qquad \texttt{ECLam2}\ \frac{e \text{ complete}}{\lambda x.e \text{ complete}}$$

$$\texttt{ECAp}\ \frac{e_1 \text{ complete} \qquad e_2 \text{ complete}}{e_1(e_2) \text{ complete}} \qquad \texttt{ECAsc}\ \frac{e \text{ complete} \qquad \tau \text{ complete}}{e : \tau \text{ complete}}$$

$\boxed{d \text{ complete}}$ d is complete

$$\texttt{DCVar}\ \frac{}{x \text{ complete}} \qquad \texttt{DCConst}\ \frac{}{c \text{ complete}} \qquad \texttt{DCLam}\ \frac{\tau \text{ complete} \qquad d \text{ complete}}{\lambda x{:}\tau.d \text{ complete}} \qquad \texttt{DCAp}\ \frac{d_1 \text{ complete} \qquad d_2 \text{ complete}}{d_1(d_2) \text{ complete}}$$

$$\texttt{DCCast}\ \frac{d \text{ complete} \qquad \tau_1 \text{ complete} \qquad \tau_2 \text{ complete}}{d\langle \tau_1 \Rightarrow \tau_2 \rangle \text{ complete}}$$

Fig. 14. Complete types, external expressions, and internal expressions

When two types are complete and consistent, they are equal.

Lemma A.5 (Complete Consistency). *If* $\tau_1 \sim \tau_2$ *an* τ_1 complete *and* τ_2 complete *then* $\tau_1 = \tau_2$.

Proof. By straightforward rule induction. □

This implies that in a well-typed internal expression, every cast is an identity cast.

We define Γ complete as follows.

Definition A.6 (Typing Context Completeness). Γ complete *iff for each* $x : \tau \in \Gamma$, *we have* τ complete.

A.4 Multiple Steps

$\boxed{d \mapsto^* d'}$ d multi-steps to d'

$$\texttt{MultiStepRefl}\ \frac{}{d \mapsto^* d} \qquad \texttt{MultiStepSteps}\ \frac{d \mapsto d' \qquad d' \mapsto^* d''}{d \mapsto^* d''}$$

Fig. 15. Multi-Step Transitions

A.5 Hole Filling

Lemma A.7 (Filling).
(1) If $\Delta, u :: \tau'[\Gamma']; \Gamma \vdash d : \tau$ *and* $\Delta; \Gamma' \vdash d' : \tau'$ *then* $\Delta; \Gamma \vdash \llbracket d'/u \rrbracket d : \tau$.
(2) If $\Delta, u :: \tau'[\Gamma']; \Gamma \vdash \sigma : \Gamma''$ *and* $\Delta; \Gamma' \vdash d' : \tau'$ *then* $\Delta; \Gamma \vdash \llbracket d'/u \rrbracket \sigma : \Gamma''$.

Proof. In each case, we proceed by rule induction on the first assumption, appealing to the Substitution Lemma as necessary. □

We need the following auxiliary definitions, which lift hole filling to evaluation contexts taking care to consider the special situation where the mark is inside the hole that is being filled, to prove the Commutativity theorem.

$\boxed{\mathsf{inhole}(u; \mathcal{E})}$ The mark in $\mathcal{E}$ is inside non-empty hole closure u

$$\texttt{InHoleNEHole}\quad \frac{}{\mathsf{inhole}(u; \llparenthesis \mathcal{E} \rrparenthesis^{u}_{\sigma})} \qquad \texttt{InHoleAp1}\quad \frac{\mathsf{inhole}(u; \mathcal{E})}{\mathsf{inhole}(u; \mathcal{E}(d))} \qquad \texttt{InHoleAp2}\quad \frac{\mathsf{inhole}(u; \mathcal{E})}{\mathsf{inhole}(u; \mathcal{E}(d))}$$

$$\texttt{InHoleCast}\quad \frac{\mathsf{inhole}(u; \mathcal{E})}{\mathsf{inhole}(u; \mathcal{E}\langle \tau_1 \Rightarrow \tau_2 \rangle)} \qquad \texttt{InHoleFailedCast}\quad \frac{\mathsf{inhole}(u; \mathcal{E})}{\mathsf{inhole}(u; \mathcal{E}\langle \tau_1 \Rightarrow \llparenthesis\rrparenthesis \nRightarrow \tau_2 \rangle)}$$

$\boxed{\llbracket d/u \rrbracket \mathcal{E} = \mathcal{E}'}$ $\mathcal{E}'$ is obtained by filling hole u in $\mathcal{E}$ with d

$$\texttt{EFillMark}\quad \frac{}{\llbracket d/u \rrbracket \circ = \circ} \qquad \texttt{EFillAp1}\quad \frac{\llbracket d/u \rrbracket \mathcal{E} = \mathcal{E}'}{\llbracket d/u \rrbracket \mathcal{E}(d_2) = \mathcal{E}'(\llbracket d/u \rrbracket d_2)} \qquad \texttt{EFillAp2}\quad \frac{\llbracket d/u \rrbracket \mathcal{E} = \mathcal{E}'}{\llbracket d/u \rrbracket d_1(\mathcal{E}) = (\llbracket d/u \rrbracket d_1)(\mathcal{E}')}$$

$$\texttt{EFillNEHole}\quad \frac{u \neq v \qquad \llbracket d/u \rrbracket \mathcal{E} = \mathcal{E}'}{\llbracket d/u \rrbracket \llparenthesis \mathcal{E} \rrparenthesis^{v}_{\sigma} = \llparenthesis \mathcal{E}' \rrparenthesis^{v}_{\llbracket d/u \rrbracket \sigma}} \qquad \texttt{EFillCast}\quad \frac{\llbracket d/u \rrbracket \mathcal{E} = \mathcal{E}'}{\llbracket d/u \rrbracket \mathcal{E}\langle \tau_1 \Rightarrow \tau_2 \rangle = \mathcal{E}'\langle \tau_1 \Rightarrow \tau_2 \rangle}$$

$$\texttt{EFillFailedCast}\quad \frac{\llbracket d/u \rrbracket \mathcal{E} = \mathcal{E}'}{\llbracket d/u \rrbracket \mathcal{E}\langle \tau_1 \Rightarrow \llparenthesis\rrparenthesis \nRightarrow \tau_2 \rangle = \mathcal{E}'\langle \tau_1 \Rightarrow \llparenthesis\rrparenthesis \nRightarrow \tau_2 \rangle}$$

Fig. 16. Evaluation Context Filling

Lemma A.8 (Substitution Commutativity). *If*
(1) $\Delta, u :: \tau'[\Gamma']; x : \tau_2 \vdash d_1 : \tau$ *and*
(2) $\Delta, u :: \tau'[\Gamma']; \emptyset \vdash d_2 : \tau_2$ *and*
(3) $\Delta; \Gamma' \vdash d' : \tau'$
then $\llbracket d'/u \rrbracket [d_2/x] d_1 = [\llbracket d'/u \rrbracket d_2/x] \llbracket d'/u \rrbracket d_1$.

Proof. We proceed by structural induction on d_1 and rule induction on the typing premises, which serve to ensure that the free variables in d' are accounted for by every closure for u. □

Lemma A.9 (Instruction Commutativity). *If*
(1) $\Delta, u :: \tau'[\Gamma']; \emptyset \vdash d_1 : \tau$ *and*
(2) $\Delta; \Gamma' \vdash d' : \tau'$ *and*

(3) $d_1 \longrightarrow d_2$
then $[\![d'/u]\!]d_1 \longrightarrow [\![d'/u]\!]d_2$.

Proof. We proceed by cases on the instruction transition assumption (no induction is needed). For Rule `ITBeta`, we defer to the Substitution Commutativity lemma above. For the remaining cases, the conclusion follows from the definition of hole filling. □

Lemma A.10 (Filling Totality). *Either* $\mathsf{inhole}(u; \mathcal{E})$ *or* $[\![d/u]\!]\mathcal{E} = \mathcal{E}'$ *for some* $\mathcal{E}'$.

Proof. We proceed by structural induction on $\mathcal{E}$. Every case is handled by one of the two judgements. □

Lemma A.11 (Discarding). *If*
(1) $d_1 = \mathcal{E}\{d'_1\}$ *and*
(2) $d_2 = \mathcal{E}\{d'_2\}$ *and*
(3) $\mathsf{inhole}(u; \mathcal{E})$
then $[\![d/u]\!]d_1 = [\![d/u]\!]d_2$.

Proof. We proceed by structural induction on $\mathcal{E}$ and rule induction on all three assumptions. Each case follows from the definition of instruction selection and hole filling. □

Lemma A.12 (Filling Distribution). *If* $d_1 = \mathcal{E}\{d'_1\}$ *and* $[\![d/u]\!]\mathcal{E} = \mathcal{E}'$ *then* $[\![d/u]\!]d_1 = \mathcal{E}'\{[\![d/u]\!]d'_1\}$.

Proof. We proceed by rule induction on both assumptions. Each case follows from the definition of instruction selection and hole filling. □

Theorem A.13 (Commutativity). *If*
(1) $\Delta, u :: \tau'[\Gamma']; \emptyset \vdash d_1 : \tau$ *and*
(2) $\Delta; \Gamma' \vdash d' : \tau'$ *and*
(3) $d_1 \mapsto^* d_2$
then $[\![d'/u]\!]d_1 \mapsto^* [\![d'/u]\!]d_2$.

Proof. By rule induction on assumption (3). The reflexive case is immediate. In the inductive case, we proceed by rule induction on the stepping premise. There is one rule, Rule `Step`. By Filling Totality, either $\mathsf{inhole}(u; \mathcal{E})$ or $[\![d/u]\!]\mathcal{E} = \mathcal{E}'$. In the former case, by Discarding, we can conclude by `MultiStepRefl`. In the latter case, by Instruction Commutativity and Filling Distribution we can take a `Step`, and we can conclude via `MultiStepSteps` by applying Filling, Preservation and then the induction hypothesis. □

We exclude these proofs and definitions from the Agda mechanization for two reasons. First, fill-and-resume is merely an optimization, and unlike the meta theory of Sec. 3, these properties are generally not conserved by certain reasonable extensions of the core calculus (e.g., reference cells and other non-commuting effects). Second, to properly encode the hole filling operation, such a mechanization requires a significantly more complex representation of hole environments; unfortunately, Agda cannot be easily convinced that the definition of hole filling is well-founded (Nanevski et al. [2008] establish that it is in fact well-founded). By contrast, the developments in Sec. 3 do not require these more complex (and somewhat problematic) representations.

A.6 Confluence, and Friends

There are various ways to encode the intuition that "evaluation order does not matter". One way to do so is by establishing a confluence property (which is closely related to the Church-Rosser property [Church and Rosser 1936]).

The most general confluence property does not hold for the dynamic semantics in Sec. 3 for the usual reason: We do not reduce under binders (Blanc et al. [2005] discuss the standard counterexample). We could recover confluence by specifying reduction under binders, either generally or in a more restricted form where only closed sub-expressions are reduced [Blanc et al. 2005; Çagman and Hindley 1998; Lévy and Maranget 1999]. However, reduction under binders conflicts with the standard implementation approaches for most programming languages [Blanc et al. 2005]. A more satisfying approach considers confluence modulo equality [Huet 1980]. The simplest such approach restricts our interest to top-level expressions of base type that result in values, in which case the following special case of confluence does hold (trivially when the only base type has a single value, but also more generally for other base types).

Lemma A.14 (Base Confluence). *If $\Delta; \emptyset \vdash d : b$ and $d \mapsto^* d_1$ and d_1 val and $d \mapsto^* d_2$ then $d_2 \mapsto^* d_1$.*

We can then prove the following property, which establishes that fill-and-resume is sound.

Theorem A.15 (Resumption). *If $\Delta, u :: \tau'[\Gamma']; \emptyset \vdash d : b$ and $\Delta; \Gamma' \vdash d' : \tau'$ and $d \mapsto^* d_1$ and $[\![d'/u]\!]d \mapsto^* d_2$ and d_2 val then $[\![d'/u]\!]d_1 \mapsto^* d_2$.*

Proof. By Commutativity, $[\![d'/u]\!]d \mapsto^* [\![d'/u]\!]d_1$. By Base Confluence, we can conclude. □

B Extensions to Hazelnut Live

We give two extensions here, numbers and sum types, to maintain parity with Hazelnut as specified by Omar et al. [2017a].

It is worth observing that these extensions do not make explicit mention of expression holes. The "non-obvious" machinery is almost entirely related to casts. Fortunately, there has been excellent work of late on generating "gradualized" specifications from standard specifications [Cimini and Siek 2016]. The extensions below, and other extensions of interest, closely follow the output of the gradualizer: http://cimini.info/gradualizerDynamicSemantics/ (which, like our work, is based on the refined account of gradual typing by [Siek et al. 2015]). The rules for indeterminate forms are mainly where the gradualizer is not necessarily sufficient.

B.1 Numbers

We extend the syntax as follows:

$$\begin{array}{lcl} \mathsf{HTyp} & \tau & ::= \cdots \mid \mathsf{num} \\ \mathsf{HExp} & e & ::= \cdots \mid \underline{n} \mid e + e \\ \mathsf{IHExp} & d & ::= \cdots \mid \underline{n} \mid d + d \end{array}$$

$\boxed{\Gamma \vdash e \Rightarrow \tau \rightsquigarrow d \dashv \Delta}$ e synthesizes type τ and expands to d

$$\frac{}{\Gamma \vdash \underline{n} \Rightarrow \mathsf{num} \rightsquigarrow \underline{n} \dashv \cdot} \qquad \frac{\Gamma \vdash e_1 \Leftarrow \mathsf{num} \rightsquigarrow d_1 : \mathsf{num} \dashv \Delta_1 \qquad \Gamma \vdash e_2 \Leftarrow \mathsf{num} \rightsquigarrow d_2 : \mathsf{num} \dashv \Delta_1}{\Gamma \vdash e_1 + e_2 \Rightarrow \mathsf{num} \rightsquigarrow d_1 + d_2 \dashv \Delta_1 \cup \Delta_2}$$

$\boxed{\Delta ; \Gamma \vdash d : \tau}$ d is assigned type τ

$$\frac{}{\Delta ; \Gamma \vdash \underline{n} : \mathsf{num}} \qquad \frac{\Delta ; \Gamma \vdash d_1 : \mathsf{num} \qquad \Delta ; \Gamma \vdash d_2 : \mathsf{num}}{\Delta ; \Gamma \vdash d_1 + d_2 : \mathsf{num}}$$

$\boxed{d\ \mathsf{val}}$ d is a value

$$\frac{}{\underline{n}\ \mathsf{val}}$$

$\boxed{\tau\ \mathsf{ground}}$ τ is a ground type

$$\frac{}{\mathsf{num}\ \mathsf{ground}}$$

$\boxed{d\ \mathsf{indet}}$ d is indeterminate

$$\frac{d_1 \neq \underline{n} \qquad d_1\ \mathsf{indet} \qquad d_2\ \mathsf{final}}{d_1 + d_2\ \mathsf{indet}} \qquad \frac{d_2 \neq \underline{n} \qquad d_1\ \mathsf{final} \qquad d_2\ \mathsf{indet}}{d_1 + d_2\ \mathsf{indet}}$$

$$\mathsf{EvalCtx} \quad \mathcal{E} \quad ::= \quad \cdots \mid \mathcal{E} + d_2 \mid d_1 + \mathcal{E}$$

$\boxed{d = \mathcal{E}\{d'\}}$ d is obtained by placing d' at the mark in $\mathcal{E}$

$$\frac{d_1 = \mathcal{E}\{d_1'\}}{d_1 + d_2 = (\mathcal{E} + d_2)\{d_1'\}} \qquad \frac{d_2 = \mathcal{E}\{d_2'\}}{d_1 + d_2 = (d_1 + \mathcal{E})\{d_2'\}}$$

$\boxed{d_1 \mapsto d_2}$ d_1 steps to d_2

$$\frac{n_1 + n_2 = n_3}{\underline{n_1} + \underline{n_2} \mapsto \underline{n_3}}$$

B.2 Sum Types

We extend the syntax for sum types as follows:

$$\begin{array}{rcll} \mathsf{HTyp} & \tau & ::= \cdots & \mid \tau + \tau \\ \mathsf{HExp} & e & ::= \cdots & \mid \mathsf{inl}(e) \mid \mathsf{inr}(e) \mid \mathsf{case}(e, x.e, y.e) \\ \mathsf{IHExp} & d & ::= \cdots & \mid \mathsf{inl}_\tau(d) \mid \mathsf{inr}_\tau(d) \mid \mathsf{case}(d, x.d, y.d) \end{array}$$

$\boxed{\mathsf{join}(\tau_1, \tau_2) = \tau_3}$ Types τ_1 and τ_2 join (consistently), forming type τ_3

$$\begin{array}{lcl} \mathsf{join}(\tau, \tau) & = & \tau \\ \mathsf{join}(\llparenthesis\rrparenthesis, \tau) & = & \tau \\ \mathsf{join}(\tau, \llparenthesis\rrparenthesis) & = & \tau \\ \mathsf{join}(\tau_1 \to \tau_2, \tau_1 \to \tau_2) & = & \mathsf{join}(\tau_1, \tau_2) \to \mathsf{join}(\tau_1, \tau_2) \\ \mathsf{join}(\tau_1 + \tau_2, \tau_1 + \tau_2) & = & \mathsf{join}(\tau_1, \tau_2) + \mathsf{join}(\tau_1, \tau_2) \end{array}$$

Theorem B.1 (Joins). *If $\mathsf{join}(\tau_1, \tau_2) = \tau$ then types τ_1, τ_2 and τ are pair-wise consistent, i.e., $\tau_1 \sim \tau_2$, $\tau_1 \sim \tau$ and $\tau_2 \sim \tau$.*

Proof. By induction on the derivation of $\mathsf{join}(\tau_1, \tau_2) = \tau$. □

$\boxed{\tau \blacktriangleright_+ \tau_1 + \tau_2}$ Type τ matches the sum type $\tau_1 + \tau_2$

$$\frac{}{\tau_1 + \tau_2 \blacktriangleright_+ \tau_1 + \tau_2} \qquad \frac{}{\llparenthesis\rrparenthesis \blacktriangleright_+ \llparenthesis\rrparenthesis + \llparenthesis\rrparenthesis}$$

$\boxed{\Gamma \vdash e \Leftarrow \tau_1 \rightsquigarrow d : \tau_2 \dashv \Delta}$ e analyzes against type τ_1 and expands to d of consistent type τ_2

$$\frac{\tau \blacktriangleright_+ \tau_1 + \tau_2 \qquad \Gamma \vdash e \Leftarrow \tau_1 \rightsquigarrow d : \tau_1' \dashv \Delta}{\Gamma \vdash \mathsf{inl}(e) \Leftarrow \tau \rightsquigarrow \mathsf{inl}_{\tau_2}(d) : \tau_1' + \tau_2 \dashv \Delta} \qquad \frac{\tau \blacktriangleright_+ \tau_1 + \tau_2 \qquad \Gamma \vdash e \Leftarrow \tau_2 \rightsquigarrow d : \tau_2' \dashv \Delta}{\Gamma \vdash \mathsf{inr}(e) \Leftarrow \tau \rightsquigarrow \mathsf{inl}_{\tau_1}(d) : \tau_1 + \tau_2' \dashv \Delta}$$

$$\frac{\begin{array}{c} \Gamma \vdash e_1 \Rightarrow \tau_1 \rightsquigarrow d_1 \dashv \Delta_1 \qquad \tau_1 \blacktriangleright_+ \tau_{11} + \tau_{12} \\ \mathsf{join}(\tau_2, \tau_3) = \tau' \qquad \Gamma, x : \tau_{11} \vdash e_2 \Leftarrow \tau \rightsquigarrow d_2 : \tau_2 \dashv \Delta_2 \qquad \Gamma, y : \tau_{12} \vdash e_3 \Leftarrow \tau \rightsquigarrow d_3 : \tau_3 \dashv \Delta_3 \end{array}}{\Gamma \vdash \mathsf{case}(e_1, x.e_2, y.e_3) \Leftarrow \tau \rightsquigarrow \mathsf{case}(d_1\langle \tau_1 \Rightarrow \tau_{11} + \tau_{12}\rangle, x.d_2\langle \tau_2 \Rightarrow \tau'\rangle, y.d_3\langle \tau_3 \Rightarrow \tau'\rangle) : \tau' \dashv \Delta_1 \cup \Delta_2 \cup \Delta_3}$$

$\boxed{\Delta; \Gamma \vdash d : \tau}$ d is assigned type τ

$$\frac{\Delta; \Gamma \vdash d : \tau_1}{\Delta; \Gamma \vdash \mathsf{inl}_{\tau_2}(d) : \tau_1 + \tau_2} \qquad \frac{\Delta; \Gamma \vdash d : \tau_2}{\Delta; \Gamma \vdash \mathsf{inr}_{\tau_1}(d) : \tau_1 + \tau_2}$$

$$\frac{\Delta; \Gamma \vdash d_1 : \tau_1 + \tau_2 \qquad \Delta; \Gamma, x : \tau_1 \vdash d_2 : \tau \qquad \Delta; \Gamma, y : \tau_2 \vdash d_3 : \tau}{\Delta; \Gamma \vdash \mathsf{case}(d_1, x.d_2, y.d_3) : \tau}$$

$\boxed{d \ \mathsf{val}}$ d is a value

$$\frac{d \ \mathsf{val}}{\mathsf{inl}_\tau(d) \ \mathsf{val}} \qquad \frac{d \ \mathsf{val}}{\mathsf{inr}_\tau(d) \ \mathsf{val}}$$

$\boxed{\tau \ \mathsf{ground}}$ τ is a ground type

$$\frac{}{\llparenthesis\rrparenthesis + \llparenthesis\rrparenthesis \text{ ground}}$$

$\boxed{d \text{ boxedval}}$ d is a boxed value

$$\frac{d \text{ boxedval}}{\mathsf{inl}_\tau(d) \text{ boxedval}} \qquad \frac{d \text{ boxedval}}{\mathsf{inr}_\tau(d) \text{ boxedval}} \qquad \frac{\tau_1 + \tau_2 \neq \tau_1' + \tau_2' \qquad d \text{ boxedval}}{d\langle \tau_1 + \tau_2 \Rightarrow \tau_1' + \tau_2' \rangle \text{ boxedval}}$$

$\boxed{d \text{ indet}}$ d is indeterminate

$$\frac{d \text{ indet}}{\mathsf{inl}_\tau(d) \text{ indet}} \qquad \frac{d \text{ indet}}{\mathsf{inr}_\tau(d) \text{ indet}} \qquad \frac{\tau_1 + \tau_2 \neq \tau_1' + \tau_2' \qquad d \text{ indet}}{d\langle \tau_1 + \tau_2 \Rightarrow \tau_1' + \tau_2' \rangle \text{ indet}}$$

$$\frac{d_1 \neq \mathsf{inl}_\tau(d_1') \qquad d_1 \neq \mathsf{inr}_\tau(d_1') \qquad d_1 \neq d_1'\langle \tau_1 + \tau_2 \Rightarrow \tau_1' + \tau_2' \rangle \qquad d_1 \text{ indet}}{\mathsf{case}(d_1, x.d_2, y.d_3)}$$

$$\mathsf{EvalCtx} \quad \mathcal{E} \quad ::= \quad \cdots \mid \mathsf{inl}_\tau(\mathcal{E}) \mid \mathsf{inr}_\tau(\mathcal{E}) \mid \mathsf{case}(\mathcal{E}, x.d_1, y.d_2)$$

$\boxed{d = \mathcal{E}\{d'\}}$ d is obtained by placing d' at the mark in $\mathcal{E}$

$$\frac{d_1 = \mathcal{E}\{d_1'\}}{\mathsf{case}(d_1, x.d_2, y.d_3) = \mathsf{case}(\mathcal{E}, x.d_2, y.d_3)\{d_1'\}}$$

$\boxed{d_1 \mapsto d_2}$ d_1 steps to d_2

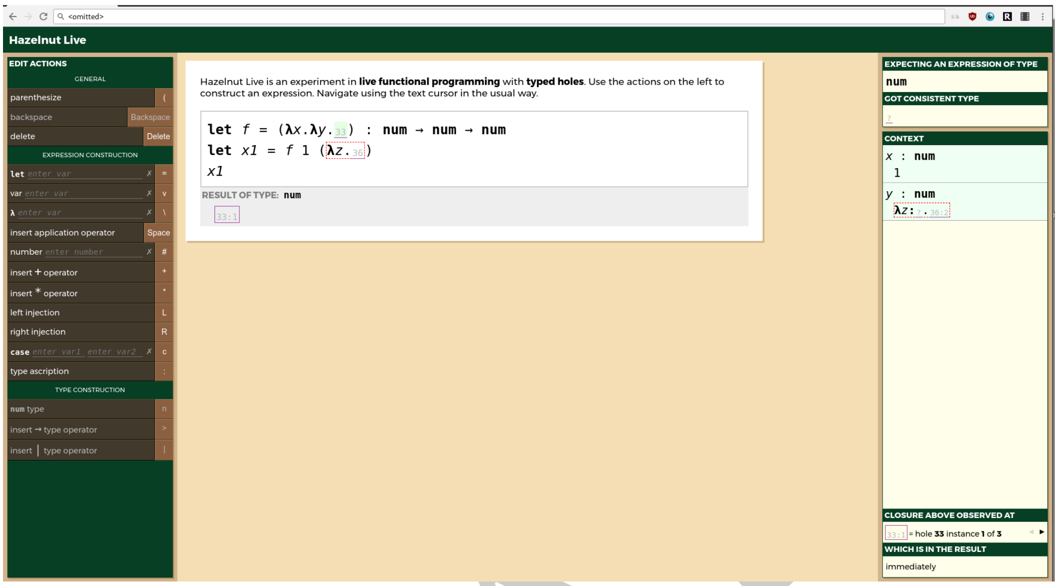
$$\frac{[d_1 \text{ final}]}{\mathsf{case}(\mathsf{inl}_\tau(d_1), x.d_2, y.d_3) \mapsto [d_1/x]d_2} \qquad \frac{[d_1 \text{ final}]}{\mathsf{case}(\mathsf{inr}_\tau(d_1), x.d_2, y.d_3) \mapsto [d_1/x]d_3}$$

$$\frac{[d_1 \text{ final}]}{\mathsf{case}(d_1\langle \tau_1 + \tau_2 \Rightarrow \tau_1' + \tau_2' \rangle, x.d_2, y.d_3) \mapsto \mathsf{case}(d_1, x.[x\langle \tau_1 \Rightarrow \tau_1' \rangle/x]d_2, y.[y\langle \tau_2 \Rightarrow \tau_2' \rangle/y]d_3)}$$

$\boxed{\tau \blacktriangleright_{\text{ground}} \tau'}$ τ has matched ground type τ'

$$\frac{\tau_1 + \tau_2 \neq \llparenthesis\rrparenthesis + \llparenthesis\rrparenthesis}{\tau_1 + \tau_2 \blacktriangleright_{\text{ground}} \llparenthesis\rrparenthesis + \llparenthesis\rrparenthesis}$$

C IMPLEMENTATION SCREENSHOTS



Hazelnut Live is an experiment in **live functional programming** with **typed holes**. Use the actions on the left to construct an expression. Navigate using the text cursor in the usual way.

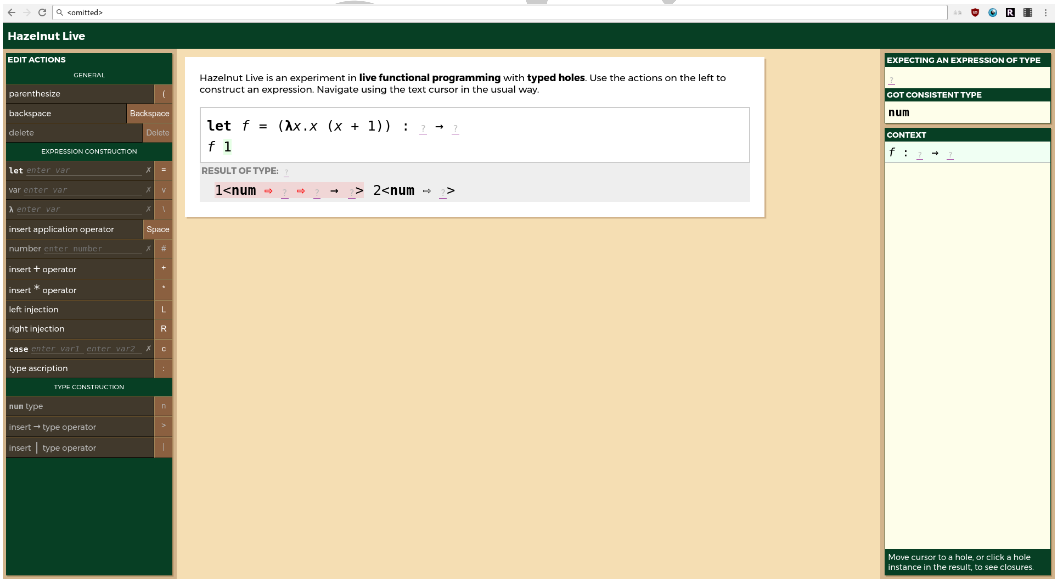
```
let f = (λx.λy.33) : num → num → num
let x1 = f 1 (λz.36)
x1
```

RESULT OF TYPE: num

```
33:1
```

EXPECTING AN EXPRESSION OF TYPE
num
GOT CONSISTENT TYPE
?
CONTEXT
x : num
1
y : num
λz:?.36:2
CLOSURE ABOVE OBSERVED AT
33:1 = hole **33** instance **1** of **3**
WHICH IS IN THE RESULT
immediately

Fig. 17. This screenshot demonstrates both empty and non-empty expression holes as well as the live context inspector.



Hazelnut Live is an experiment in **live functional programming** with **typed holes**. Use the actions on the left to construct an expression. Navigate using the text cursor in the usual way.

```
let f = (λx.x (x + 1)) : ? → ?
f 1
```

RESULT OF TYPE: ?

```
1<num ⇨ ? ⇨ ? → ?> 2<num ⇨ ?>
```

EXPECTING AN EXPRESSION OF TYPE
?
GOT CONSISTENT TYPE
num
CONTEXT
f : ? → ?

Move cursor to a hole, or click a hole instance in the result, to see closures.

Fig. 18. This screenshot demonstrates dynamic cast failure.