THE COOPER UNION FOR THE ADVANCEMENT OF SCIENCE AND ART

ALBERT NERKEN SCHOOL OF ENGINEERING

Optimization by memoization of evaluation tasks in the Hazel structured programming environment

by Jonathan Lam

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Engineering

Professor Fred L. Fontaine, Advisor

Professor Robert Marano, Co-advisor

Performed in collaboration with the

Future of Programming Lab at the University of Michigan

THE COOPER UNION FOR THE ADVANCEMENT OF SCIENCE AND ART

ALBERT NERKEN SCHOOL OF ENGINEERING

This thesis was prepared under the direction of the Candidate's Thesis Advisor and has received approval. It was submitted to the Dean of the School of Engineering and the full Faculty, and was approved as partial fulfillment of the requirements for the degree of Master of Engineering.

Barry L. Shoop, Ph.D., P.E. Date

Fred L. Fontaine, Ph.D. Date

ACKNOWLEDGEMENTS

TODO

ABSTRACT

TODO

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

LIST OF THEOREMS

TABLE OF NOMENCLATURE

τ	Hazel type
Γ	Typing context
Δ	Hole context

Table 1: Hazel expression and hole typing

d	Internal expression
$\lambda x.d$	Lambda abstraction
$\mathrm{fix}\, f.d$	Fixpoint function
σ	Environment
x, f	Variable name
u	Hole number
i	Hole instance or closure number
$\llparenthesis\rrparenthesis^{u:i}_{\sigma}$	Empty hole expression
$\llparenthesis d \rrparenthesis^{u:i}_{\sigma}$	Non-empty hole expression

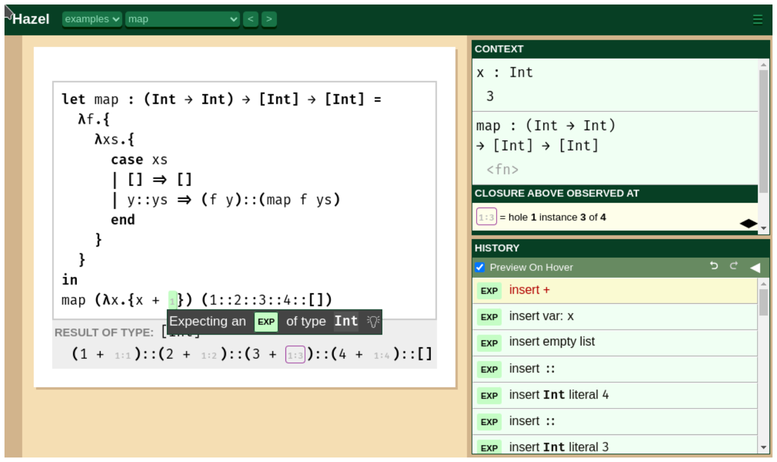
Table 2: Hazel internal language

$d\ \mathsf{value}$	Value
$d\ \mathsf{final}$	Final
$\sigma \vdash d \Downarrow d'$	Evaluation
$d \Uparrow d'$	Postprocessing
$d \Uparrow_{[]} d'$	Postprocessing (λ-conversion)
$d \Uparrow_{i} (H, d')$	Postprocessing (hole closure numbering)

Table 3: Hazel evaluation and postprocessing judgments

H	Hole instance/closure information
hid	Hole instance/closure id generation function
p	Hole instance path

Table 4: Hazel postprocessing



```
let map : (Int → Int) → [Int] → [Int] =
  λf.{
    λxs.{
      case xs
      | [] ⇒ []
      | y::ys ⇒ (f y)::(map f ys)
      end
    }
  }
in
map (λx.{x + 1}) (1::2::3::4::[])
```

Expecting an EXP of type Int

```
(1 + 1:1)::(2 + 1:2)::(3 + 1:3)::(4 + 1:4)::[]
```

```
x : Int
  3

map : (Int → Int)
→ [Int] → [Int]
  <fn>
```

1:3 = hole **1** instance **3** of **4**

EXP insert +
EXP insert var: x
EXP insert empty list
EXP insert ::
EXP insert **Int** literal 4
EXP insert ::
EXP insert **Int** literal 3

Figure 1: A screenshot of the Hazel live programming environment. Screenshot taken of the dev branch demo[1] on 02/06/2022.

1 Introduction

1.1 Problem statement

Unstructured plaintext editing has remained the dominant mode of programming for decades, but makes it more difficult to implement editor services to aid the process. Several structural editors, which only allow valid edit states, have been proposed to improve the programming experience and editor services, such as the elimination of syntax errors or graphical editing.

Hazel is an experimental structural language definition and implementation that aims to solve the "gap problem": spatial and temporal holes that temporarily prevent code from being able to be compiled or evaluated. The structural editor is defined by a bidirectional edit calculus Hazelnut [2], which governs the structural editor and the static semantics (typing rules) of the language. The dynamic semantics (evaluation semantics) are described in [1].

Hazel is a relatively new research effort, with little effort placed on performance enhancements. This work attempts to achieve several modifications that will benefit the performance of evaluation and related tasks. Part of the work will be the standard conversion from evaluation using the substitution model (simpler to reason about) to the environment model (more performant), with

[1] `https://hazel.org/build/dev/`

emphasis on evaluation of holes and postprocessing of the evaluation result to match the result from evaluation with substitution. The latter parts of this work will use the environment model of evaluation to improve the memoization of certain tasks related to Hazel's structure (such as hole closure numbering), and also implement the fill-and-resume performance enhancement described in [1]. The novelty of this work lies in the novelty and optimization opportunity of Hazel's hole-based static and dynamic semantics.

1.2 The contribution of this work

This thesis presents several algorithms designed for Hazel's evaluation:

- The evaluation semantics of the Hazel language using the environment model (which replaces the substitution model as implemented on the trunk branch and described in [1]). While most of this is standard, we aim to keep the implementation pure (which is less trivial in the case of recursion), introduce uniquely-numbered environments (for later use in memoization), and describe the evaluation of holes (which are unique to Hazel).

- Postprocessing, which is memoized by environments and has the dual functions of converting the result to the equivalent result from evaluation with substitution, and performing hole closure numbering. Converting the result to the substitution model, hole closure numbering, and memoization of environments are all described separately.

- Fill-and-resume, as originally proposed in [1]. This algorithm is described at a high level in the original description and not yet implemented until this thesis work. We provide the implementation and a lower-level description of said implementation.

The first two algorithms will be provided as a series of (big-step) inference rules, in the same style as the existing literature. Fill-and-resume will be presented at a higher level, being more of a composition of existing functions of the Hazel architecture.

In addition to the algorithms above, several core concepts or data structures are introduced to Hazel, such as unique hole closures (as opposed to hole instances) and generalized closures. While the first one is specific to holes and thus specific to Hazel(nut), the latter is a concept that may be transferred to any live environment that may perform a similar conversion between evaluation with environments (for evaluation performance) to a result using substitution (for display and debugging purposes).

The performance of this work is measured primarily in terms of empirical performance gains (via evaluation-step counting and benchmarking), and discussed with respect to the theoretical performance. This proof of correctness of the algorithms was not mechanized in the Agda proof assistant as was much of the core of Hazelnut and Hazelnut Live (and is deferred to future work); instead, correctness of implementation is validated by standard software testing procedures with manual test cases.

1.3 Structural overview

Section 2 provides a background on necessary topics in programming language (PL) theory and programming language implementations, in order to frame understanding for the Hazel live programming environment. Section 3 provides an overview of Hazel, in order to frame the work completed for this thesis project. Sections 4 to 6 describe the primary work completed for this project, as described in Section 1.2. Section 7 comprises an assessment of the work completed in terms of correctness and the theoretical performance. Section 8 is a discussion of future research directions that may be spawned off from this work. Section 9 concludes with a summary of findings and future work. The appendices contain additional information about the Hazel project not directly related to the primary contribution of this project, as well as selected source code snippets.

2 Background

This chapter is intended to provide a practical introduction to the theory of functional programming and programming languages, as relevant to this work on Hazel. It is in particular geared towards engineers with a general programming background, and especially a background programming in imperative languages.

2.1 Functional programming

Functional programming is a programming paradigm tat is highly involved with function application, function composition, and first-class functions. It is generally a subtype of, and often associated with, the declarative programming paradigm, which is concerned with expression-based computation, often without mutable state or side-effects. Declarative programming is often considered the complement of imperative programming, which may be characterized as programming with mutable state, side effects, or statements. Purely functional programming is a subset of functional programming that deals solely with pure functions; non-pure languages may allow varying degrees of mutable state but typically encourage the use of pure functions.

Functional languages are based on Church's λ calculus as its core evaluation and typing semantics, which provides a minimal foundation for computation. Functional programming's syntax is based off the λ calculus. This, along with the lack of mutable state and side effects, allows functional programming to be easily mathematically modeled and reasoned about, making it particularly amenable to proofs about programming languages and programming language research. This is as opposed to in imperative programming, in which the mutable "memory cell" interpretation of variables and side-effects complicates formalizations.

Hazel is one such (purely) functional programming languages. Other languages that are classified as functional include the ML family of languages, Haskell, Elm, and the LISP family of languages. Examples of imperative programming languages include C, C++, FORTRAN, Java, and Golang. A number of languages incorporate both functional and imperative styles, such as Javascript, Python, Scala, and Rust.

2.1.1 The λ-calculus

In any computer program, complexity is built through the use of algorithmic and data abstractions. The type of data abstractions in the ML family is described in Section 2.1.3.

Algorithmic abstraction and Turing-completeness in the λ calculus is requires function recursion. Recursion closely corresponds to mathematical induction, which allows us to elegantly model algorithms using the judgment-rule style notation described in Section 2.3.1.

2.1.2 Purity and statefulness

2.1.3 The ML family, Elm, and Hazel

2.2 Implementations for programming languages

In order for a programming language to be practical, it must not only be defined as a set of syntax and semantics, but also have an *implementation* to run programs in the language. Hazel is implemented as an interpreted language, whose runtime is transpiled to Javascript so that it may be run as a client-side web application in the browser.

It is important to note that the definition of a language (its syntax and semantics) are largely orthogonal to its implementation. In other words, a programming language does not dictate whether it requires a compiler or interpreter implementation, and languages sometimes have multiple implementations.

2.2.1 Compiler vs. interpreter implementations

There are two general classes of programming language implementations: *interpreters* and *compilers*. Both types of implementations share the function of taking a program as input, and should be able to produce the same result (assuming an equal and determinstic machine state, equal inputs, correct implementations, and no exceptional behavior due to differences in resource usage).

A compiler is a programming language implementation that converts the program to some low-level representation that is natively executable on the hardware architecture (e.g., x86-64 assembly for most modern personal computers, or the virtualized JVM architecture) before evaluation. This process typically comprises *lexing* (breaking down into atomic tokens) the program text, *parsing* the lexed tokens into a suitable *intermediate representation* (IR) such as LLVM, performing optimization passes on the intermediate representation, and then generating the target bytecode (such as x86-64 assembly). The bytecode outputted from the compilation process is used for evaluation. Compiled implementations tend to produce better runtime efficiency, since the compilation steps are performed separate of the evaluation, and because there is little to no runtime overhead.

An interpreter is a programming language implementation that does not compile down to native bytecode, and thus requires an interpreter or *runtime*, which performs the evaluation. Interpreters still require lexing and parsing, and may have any number of optimization stages, but do not generate bytecode for the native machine, instead evaluating the program directly.

In certain contexts (especially in the ML spheres), the term *elaboration* is used to the process of transforming the *external language* (a well-formed, textual program) into the *internal language* (IR). The interior language may include additional information not present in the external language, such as types generated by type inference (e.g., in SML/NJ) or bidirectional typing (e.g., in Hazel).

The distinction between compiled and interpreted languages is not a very clear line: some implementations feature just-in-time (JIT) compilation that allow "on-the-fly" compilation (e.g., the JVM or CLR), and some implementations may perform the lexing and parsing separately to generate a non-native bytecode representation to be later evaluated by a runtime. A general characterization of compiled vs. interpreted languages is the amount of runtime overhead required by the implementation.

Hazel is a purely interpreted language implementation, as optimizations for speed are not among its main concerns. However, performance is clearly one of the main concerns of this thesis project, but the gains will be algorithmic and use the nature of Hazel's structural editing and hole calculus to benefit performance, rather than changing the fundamental implementation. There is, however, a separate endeavor to write a compiled interpretation of Hazel[2], which is outside the scope of this project.

[2]`https://github.com/hazelgrove/hazel/tree/hazelc`

2.2.2 The substitution and environment models of evaluation

Evaluation in Hazel was performed using originally using a *substitution model of evaluation*, which is a theoretically simpler model. In this model, variables that are bound by some construct are substituted into the construct's body. For example, the variable(s) bound using a `let`-expression pattern are substituted in the `let`-expression's body, and the variable(s) bound during a function application are substituted into the function's body, and then the body is evaluated.

In this formulation, variables are "given meaning" via substitution; once evaluation reaches an expression, all variables in scope (in the typing context) will have been replaced by their value by some containing binding expression. In other words, variables are never evaluated directly; they are substituted by their values when bound, and their values are evaluated. The substitution model is useful for teaching purposes because it is simple and close to its mathematical definition: a variable can be thought of as an equivalent stand-in for its value.

However, for the purpose of computational efficiency, a model in which values are lazily expanded ("looked-up") only when needed is more efficient. This is called the *environment model of evaluation*, and generally is more efficient because the runtime does not need to perform an extra substitution pass over subexpressions; untraversed (unevaluated) branches do not require substituting; and the runtime does not need to carry an expression-level IR of the language. The last point is due to the fact that the substitution model manipulates expressions, while evaluation does not; this means that the latter is more amenable for compilation, and is how compiled languages tend to be implemented: each frame of the theoretical stack frame is a de facto environment frame. While switching from the substitution to environment model is not an improvement in asymptotic efficency, these effects are useful especially for high-performance and compiled languages.

Note that the substitution model does not imply a lazy (i.e., normal-order, call-by-name, call-by-need) evaluation as in languages such as Haskell or Miranda, in which bound variables are (by default) not evaluated until their value is required. Laziness is conceptually tied to substitution, but the substitution model does not require laziness. Like most programming languages, Hazel only has strict (i.e., applicative-order, call-by-value) evaluation: the expressions bound to variables are evaluated at the time of binding.

The implementation of evaluation with environments differs from that of evaluation with substitution primarily in that: an evaluation environment is required to look up bound variables as evaluation reaches them; binding constructs extend the evaluation environment rather than performing substitution; and λ abstractions are bound with their evaluation environment at runtime to form (lexical) closures.

2.3 Programming language semantics

2.3.1 Notation

2.3.2 Static and dynamic semantics

2.3.3 Gradual typing

3 An overview of the Hazel programming environment

Hazel is the experimental language that implements the Hazelnut bidirectionally-typed edit static semantics with holes and the Hazelnut Live dynamic semantics, and it is also the name of the reference implementation. It is intended to serve as a proof-of-concept of the semantics with holes that attempt to mitigate the gap problem; however, the implementation is becoming increasingly practical with additional research effort. The reference implementation is an interpreter written in OCaml and transpiled to Javascript using the `js_of_ocaml` (JSOO) library [3] so that it may be run client-side in the browser. A screenshot of the reference implementation[3] is shown in Figure 1. The source code may be found on GitHub[4].

Hazel's syntax and semantics resembles languages in the ML (Meta Language) family of languages such as OCaml, although Hazel does not support polymorphism at this time. Hazel can be characterized as a purely functional, statically-typed, bidirectionally-typed, strict-order evaluation, structured editor language. Hazel semantically differs most significantly from other ML languages in the last respect due to its theoretic foundations in solving the gap problem.

[3]`https://hazel.org/build/dev/`
[4]`https://github.com/hazelgrove/hazel`

3.1 Hazelnut static semantics

3.1.1 Expression and type holes

3.1.2 Bidirectional typing

3.1.3 Example of bidirectional type derivation

3.2 Hazelnut Live dynamic semantics

3.2.1 Example of elaboration

3.2.2 Example of evaluation

3.2.3 Example of hole instance numbering

3.3 Hazel programming environment

3.3.1 Explanation of interface

3.3.2 Implications of Hazel

4 Implementing the environment model of evaluation

4.1 Hazel-specific implementation

In the case of Hazel (which does not prioritize speed of evaluation in its implementation, and is not a compiled language), evaluation with (reified) environments offers an additional (performance) benefit over the substitution model: the ability to easily identify (and thus memoize) operations over environments. This is useful for the optimizations described later in this paper.

The implementation of evaluation in Hazel differs from a typical interpreter implementation of evaluation with environments in three regards: we need to account for hole environments; environments are uniquely identified by an identifier for memoization (in turn for optimization); and any closures in the evaluation result should be converted back into plain λ abstractions.

4.1.1 Evaluation rules

Omar et al. [1] describes evaluation with the substitution model using a little-step semantics with an evaluation context $\mathcal{E}$. The Hazel implementation follows a big-step model for evaluation, which is simpler, more performant, and does not require the evaluation context. Thus it is more convenient to follow a big-step semantics as shown in Figure 2.

The evaluation model threads a run-time environment σ[5] throughout the evaluation process. An environment is conceptually a mapping $\sigma : x \mapsto d$, although it will later be augmented to be more amenable to memoization.

Evaluation judgments are shown for a subset of the Hazel language, similar to the internal language described in [1]. The expressions considered include a single base type b, variables x, λ abstractions, function application, and hole expressions. Casts and type ascriptions, which are part of the internal language follow the same rules as described in the Hazelnut paper, and thus are omitted here. Additionally, a rule is included for `let` bindings, even if not strictly necessary. There are additional forms in the Hazel external and internal languages that are omitted for brevity and whose rules are trivial: these include binary sum injections and tuples, for which evaluation recurses

[5]The symbol σ was chosen to represent the environment as it was used to represent hole environments in [1]. The relationship between these two environments will be discussed in Section 4.1.2.

$\boxed{\sigma \vdash d \Downarrow d'}$ Internal expression d evaluates to d' given environment σ

$$\frac{\sigma \vdash d \ \mathsf{final}}{\sigma \vdash d \Downarrow d}\ \text{EvalB-Final} \qquad \frac{}{\sigma \vdash (\lambda x : \tau . d) \Downarrow [\sigma](\lambda x : \tau . d')}\ \text{EvalB-Lam}$$

$$\frac{d \neq \text{fix}\, f . d'}{\sigma, x \leftarrow d \vdash x \Downarrow d}\ \text{EvalB-Var} \qquad \frac{\sigma \vdash \text{fix}\, f . d \Downarrow d'}{\sigma, x \leftarrow \text{fix}\, f . d \vdash x \Downarrow d'}\ \text{EvalB-Unwind}$$

$$\frac{\sigma \vdash d \Downarrow d' \qquad \sigma, f \leftarrow \text{fix}\, f . d' \vdash d \Downarrow d''}{\sigma \vdash \text{fix}\, f . d \Downarrow d''}\ \text{EvalB-Fix}$$

$$\frac{\sigma \vdash d_1 \Downarrow d_1' \qquad d_1' \neq ([\sigma']\lambda x . d) \qquad \sigma \vdash d_2 \Downarrow d_2'}{\sigma \vdash d_1(d_2) \Downarrow d_1'(d_2'))}\ \text{EvalB-Ap}_1$$

$$\frac{\sigma \vdash d_1 \Downarrow ([\sigma']\lambda x . d_1') \qquad \sigma \vdash d_2 \Downarrow d_2' \qquad \sigma, x \leftarrow d_2' \vdash d_1' \Downarrow d}{\sigma \vdash d_1(d_2) \Downarrow d}\ \text{EvalB-Ap}_2$$

$$\frac{\sigma \vdash d_2 \Downarrow d_2' \qquad \sigma, x \leftarrow d_2' \vdash d_1 \Downarrow d}{\sigma \vdash \texttt{let}\ x = d_2\ \texttt{in}\ d_1 \Downarrow d}\ \text{EvalB-Let}$$

$$\frac{}{\sigma \vdash \llparenthesis\rrparenthesis^u_\varnothing \Downarrow \llparenthesis\rrparenthesis^u_\sigma}\ \text{EvalB-EHole} \qquad \frac{\sigma \vdash d \Downarrow d'}{\sigma \vdash \llparenthesis d \rrparenthesis^u_\varnothing \Downarrow \llparenthesis d' \rrparenthesis^u_\sigma}\ \text{EvalB-NEHole}$$

Figure 2: Big-step semantics for the environment model of evaluation

through subexpressions. `case` expressions are also omitted: it acts like a sequence of `let` bindings. This select subset of the Hazel language will be reused throughout this paper for judgment rules; the goal is to provide a practical intuition of the evaluation semantics of Hazel that is close to the implementation, and not to provide a minimal theoretic foundation or the complete set of rules for all Hazel expressions. The latter is deferred to the source code in the reference implementation. Patterns and pattern holes will also be omitted from the rules, as they are not the focus of this work.

As always, elements of the base type are values and do not further evaluate. Bound variables evaluate to their value in the environment. (Unbound variables are marked as free during elaboration and do not further evaluate.)

λ-abstractions $\lambda x.d$ are no longer final values; they evaluate further to the function closure $[\sigma]\lambda x.d$, which captures the lexical environment of the λ expression[6].

A description of recursive λ-abstractions (the fixpoint form) is described in Section 4.1.3.

Function application is broken into two cases: if the expression in function position evaluates to a closure and the argument matches the argument pattern, then the evaluated expression in argument position extends the closure's environment, and that extended environment is used as the lexical environment in which to evaluate the λ expression body. Otherwise, the expression in function position must evaluate to an indeterminate (failed cast) form, in which case evaluation cannot proceed further. The case of failed pattern matches is described in Section 4.2.1.

`let` bindings extend the current lexical environment with the bound variable. As with λ-abstractions, the case in which the pattern match fails is described in Section 4.2.1.

4.1.2 Evaluation of holes

Hole expressions are separated into the empty and non-empty cases due to the lack of empty exprsessions, as in the original Hazelnut and Hazelnut Live descriptions. When evaluation reaches a hole, the hole environment is simply set to be equal to the lexical environment. In this interpretation, free variables do not exist in the hole environment.

[6]This step is conceptually similar to the first step of closure conversion, in which λ-abstractions are converted to functions that take two parameters: the argument and the environment.

Note that the initial hole environment is different than in the substitution model. When evaluating using the substitution model, the initial hole environment generated by elaboration is the identity substitution id(Γ), and variable bindings are recursively substituted into the environment's bindings. This is not necessary anymore with the environment model, and the initial environment created by elaboration is not as important. In this interpretation, free variables exist in the hole environment as the identity substitution.

It is convenient to replace the identity substitution with a distinguished empty environment (represented by $\varnothing$) that indicates that evaluation has not yet reached a hole. This will also be useful for detecting errors with the evaluation boundary discussed in Section 4.2.

4.1.3 Evaluation of recursive functions

When evaluating with substitution, recursion needs to be explicitly handled using a fixpoint form that allows for self-recursion, otherwise inifinitely recursive substitution will occur.

Recursion with the environment model also requires self-reference, but this can be achieved in two ways: by accounting for the fixpoint form, or by using self-referential data structures. In OCaml, self-referential (mutually recursive) data can be achieved using the `let rec` keyword or by using `ref`s (mutable data cells); however, the latter will affect the purity of the implementation, as discussed in Section 2.1.2.

Both pure methods were implemented; their tradeoffs are described below. The final implementation (and the rules shown in Figure 2) uses the implementation with the fixpoint form, although the choice is somewhat arbitrary.

Performing evaluation with the fixpoint form follows very similar rules to the substitution model. Recursive λ functions in the external language elaborate to a λ function wrapped in a `FixF` variant during elaboration in the internal language[7]. The evaluation of the `FixF` form introduces the self-reference to the current environment. To do this, the body expression is first evaluated without the self-reference; that evaluated expression is added to the environment; and then the body

[7]The current implementation only allows recursion for type-ascribed `let` expressions with a single λ abstraction on the RHS. Mutual recursion is currently not supported, but is being worked on in the mutual-rec branch. The described implementation should extend straightforwardly to an implementation of mutual recursion involving self-reference of a tuple and projection out of the tuple.

$$\frac{\sigma' = \sigma, f \leftarrow d_1' \qquad d_1' = [\sigma']\lambda x.d_1 \qquad \sigma' \vdash d_2 \Downarrow d}{\sigma \vdash \texttt{let}\ f = \lambda x.d_1\ \texttt{in}\ d_2 \Downarrow d}\ \text{EvalB-LetSimpleRec}$$

Figure 3: Evaluation rule for simple recursion using self-recursive data structures

expression is evaluated again, with the self-reference[8]. The unwrapping of the recursive function occurs when the recursive form is looked up in its environment, which is indicated by the special variable evaluation rule EvalB-Unwind.

We may avoid the fixpoint form by using mutually-recursive data structures, so that a closure may contain an environment which contains itself as a binding. This is easy to implement in a language with pointers or mutable references, and how recursion is generally implemented. Mutually-recursive data in OCaml is somewhat tricky in the general case, as it requires statically-constructive forms[9]. In the more general case of mutual recursion, this would likely make implementation very tricky, and it would be more practical to use impure `ref`s to achieve self-reference. However, for the simple case of a simply recursive function, we may recognize `let`-bindings which introduce a function, and statically construct the mutual recursion using the rule shown in Figure 3. This is very similar to the way that `FixF` expressions are inserted automatically during elaboration; the need for that elaboration step is eliminated, since the `FixF` form doesn't exist during evaluation.

Using the recursive environment in closures helps improve performance, due to the elimination of special processing (unwinding) for recursive function definitions and invocations. However,

[8]Evaluating the body expression twice may seem expensive, except that the body is (in the current implementation) always a lambda function, which trivially evaluates to a closure by binding its environment. As a result, we can simplify the evaluation of a `FixF` to one of the following forms. The first occurs when the recursive function is defined, and the second occurs when the recursive form is looked up in its environment (and unwrapped).

$$\frac{}{\sigma \vdash \text{fix}\, f.\lambda x.d \Downarrow [\sigma, f \leftarrow \text{fix}\, f.[\sigma]\lambda x.d]\lambda x.d}\ \text{EvalB-FixF}_1$$

$$\frac{}{\sigma \vdash \text{fix}\, f.[\sigma']\lambda x.d \Downarrow [\sigma', f \leftarrow \text{fix}\, f.[\sigma']\lambda x.d]\lambda x.d}\ \text{EvalB-FixF}_2$$

Mutual recursion can be implemented as a self-reference applied to a tuple of λ functions, which requires the more general form presented in Figure 2. It also does not take many evaluation steps and is thus not an expensive operation.

[9]§10.1: Recursive definitions of values of the OCaml reference describes this in greater detail. Simply put, this prevents recursive variables from being defined as arguments to functions, instead only allowing recursive forms to be arguments to data constructors.

it complicates the display of recursive functions in the context inspector and structural equality checking, due to infinite recursion. The first problem is solved by re-introducing the FixF form during postprocessing (Section 4.2) by detecting recursive environments and converting them to FixF expressions; however, there is a nuance that may cause the postprocessed result to be slightly different[10]. The second problem is solved by the fast equality checker for memoized environments described in Section 5.5.3, which is useful even for non-recursive environments. We may also say that using recursive data structures without mutable `ref`s is limited by the language limitations, necessitating workarounds even for the simply-recursive case, and potentially much more complicated workarounds for the mutual recursion case.

The performance improvement is described in Section 7. The complexities of postprocessing outweigh the small performance benefit, so it was chosen for the final implementation. However, both are viable for a practical implementation of recursion using only pure constructs in OCaml.

4.2 The evaluation boundary and general closures

Evaluation with the environment model is "lazy" in that evaluation steps that require the environment (e.g., evaluation of holes, and evaluation of variables) are only performed when evaluation reaches the expression of interest. Evaluation with the substitution model is "eager" because variable variable values propagate through all subexpressions (even unevaluated ones) upon binding. While lazy evaluation is better for performance, in the Hazel environment we expect to see fully-substituted values in the context inspector for hole contexts environments. This means that we

[10]To illustrate this, consider the simple Hazel program:

```
let f = λ x . { ⦇⦈^1 } in
f f
```

The result will be a closure of hole 1 with the identifiers `x` and `f` in scope. When evaluating using the FixF form, the binding for `f` will be the expression $(\text{fix}\, f.[\varnothing]\lambda x.⦇⦈^1)$, and the binding for `x` is $([f \leftarrow \text{fix}\, f.[\varnothing]\lambda x.⦇⦈^1]\lambda x.⦇⦈^1)$. `f` is bound to the closure in the EvalB-Fix rule, and `x` is bound during EvalB-Ap to the evaluated value of `f`.

However, when evaluating with a recursive data structure, both `x` and `f` refer to the same value $d = ([f \leftarrow d]\lambda x.⦇⦈^1)$. It is impossible to discern the two and decide where to begin the "start of the recursion", i.e., to determine that `f` should be a FixF expression and `x` should be a Lam expression, at least without significant additional extra effort. Thus to remove the recursion, we may arbitrarily decide that the outermost recursive form should be a Lam expression and set the recursive binding in its environment to be a FixF form, which will successfully remove the recursion but mistakenly change some expressions that would be FixF forms to Lam expressions. Whether this distinction is very important is another story, but it may at least confuse the user.

require a postprocessing step to perform substitution of bound variables in environments to achieve the same result as if we had evaluated by means of substitution.

In other words, any unevaluated expression must be "caught up" to the substituted equivalent after evaluation. This requires that the environment be stored alongside the unevaluated expression, and that a postprocessing step should be taken to perform the substitution and discard the stored environment. Note that this is essentially performing substitution pass after evaluation, but is preferred over substitution during evaluation because it is only performed on the result (rather than all the intermediate expressions during evaluation).

We define the **evaluation boundary** to be the conceptual distinction between expressions for which evaluation has reached ("inside" the boundary), and for those that remain unevaluated ("outside" the boundary). This definition will be useful for describing the postprocessing algorithm.

4.2.1 Evaluation of failed pattern matching using generalized closures

There are two cases where an expression in the evaluation result may lie outside the evaluation boundary. The first is in the body of a λ expression. A λ expression evaluates to a closure, and thus captures an environment with it. The second case is that of an unmatched `let` or `case` expression (in which the scrutinee matches none of the rules), for which the body expression(s) will remain unevaluated in the result without an associated environment[11]. This is not captured in the original description of Hazelnut Live [1] or in this paper because pattern-matching is not a primary concern of either of these works. However, it is a practical concern that arises from the introduction of evaluation with environments.

We solve this by introducing (lexical) **generalized closures**, the product of an arbitrary expression and its lexical environment. Traditionally, the term "closure" refers to **function closures**, which are the product of a λ abstraction with its lexical environment. Hazelnut Live [1] introduces **hole closures**, which are the product of empty and non-empty holes with their lexical environments, and are fundamental to the Hazel live environment: they allow a user to inspect a hole's

[11]There is a third place where pattern-matching may fail: the pattern of an applied λ abstraction may not match its argument. However, this is not an issue since there exists a function closure containing the unevaluated expression's environment.

```
type t =
  (* Hole types *)
  | EmptyHole(u, i, σ)
  | NonEmptyHole(u, i, σ, d)
  | Keyword(u, i, σ, ...)
  | InvalidText(u, i, σ, ...)
  | FreeVar(u, i, σ, ...)
  | InconsistentBranches(u, i, σ, ...)
  (* Lambda expressions and λ closures *)
  | Lam(x, τ, d)
  | FnClosure(σ, x, τ, d)
  (* ... *) ;
```

(a) Non-generalized closures

```
type t =
  (* Hole types *)
  | EmptyHole(u, i)
  | NonEmptyHole(u, i, d)
  | Keyword(u, i, ...)
  | InvalidText(u, i, ...)
  | FreeVar(u, i, ...)
  | InconsistentBranches(u, i, ...)
  (* Lambda expressions and closures *)
  | Lam(x, τ, d)
  (* Generalized closure *)
  | Closure(σ, d)
  (* ... *) ;
```

(b) Generalized closures

Figure 4: Comparison of internal expression datatype definitions (in module `DHExp`) for non-generalized and generalized closures.

environment in the context inspector, and enable the fill-and-resume optimization. We propose generalizing the term "closures" to the definition stated above. Conceptually, all generalized closures represent a partial or stopped evaluation (using the environment model), as well as the state (the environment) that may be used to resume the evaluation. Similar to the evaluation of function closures, closures are final (boxed) values and evaluate to themselves.

The application of generalized closures to the problem of unevaluated `let` or `case` bodies is straightforward: if there is a failed pattern match, wrap the entire expression in a (generalized) closure with the current lexical environment. Then, the postprocessing can successfully perform the substitution.

4.2.2 Generalization of existing hole types

Consider the abbreviated definition of the internal expression variant type in Figure 4. In Figure 4a the previous implementation is shown (when evaluating using the substitution model), augmented with a type for function closures. There are ordinary `Let` and `Case` variants, which do not contain an environment. In this version, each expression variant that requires an environment has the environment hardcoded into the variant. In Figure 4b the proposed version with generalized closures is shown. The `Lam`, `Let`, and `Case` variants are unchanged. Importantly, the environments are removed from the hole types and a new generalized `Closure` is introduced. In this model, a hole, λ abstraction, unmatched `let`, or unmatched `case` expression is wrapped in the `Closure` variant when evaluated.

The notation used to express a function closure may be extended to all generalized closure types. In particular, the environment for a hole changes from the initial notation used in [1]:

$$[\sigma]\lambda x.d \qquad \text{(function closure)}$$

$$[\sigma](\!|d|\!)^u \qquad \text{(hole closure)}$$

$$[\sigma](\texttt{let}\ x = d_1\ \texttt{in}\ d_2) \qquad \text{(closure around \texttt{let})}$$

$$[\sigma](\texttt{case}\ x\ \texttt{of}\ \text{rules}) \qquad \text{(closure around \texttt{case})}$$

This implementation of closures is an improvement in two ways. Firstly, it simplifies the variant types by factoring out the environment, separating the "core" expression from the environment coupled with it. Secondly, it allows for a more intuitive understanding of holes in the environment model of evaluation. This solves the question of what environment to initialize a hole with when it is created during the elaboration phase: a hole is simply initialized without a hole environment, much as a function closure is initially without an environment (a plain syntactical λ abstraction). It also removes the ambiguity of the notation $(\!|d|\!)_\varnothing$, which could intuitively mean either a hole that has not been evaluated (if initialized during elaboration with a special empty environment) or a hole that has been evaluated in the empty environment.

Note that while the generalized closures for the body expressions of λ abstractions, unmatched `let` expressions, and unmatched `case` expressions represent expressions outside of the evaluation boundary, the expressions within non-empty holes (which also are bound to a hole closure) lie within the evaluation boundary. This shows the two goal that generalized closures achieve; to encapsulate a stopped expression (which is used during postprocessing to perform substitution), and to encapsulate an expression to be fill-and-resumed.

4.2.3 Alternative strategies for evaluating past the evaluation boundary

Without generalized closures, unevaluated expressions (body expressions of λ abstractions, unmatched `let` expressions, and unmatched `case` expressions) may be filled by a modified form of evaluation, which is only different in that a failed lookup (due to unmatched variables) will leave the variable unchanged[12]. However, this is essentially the same as substitution, and is expensive to do during evaluation. Also, while this speculative execution would be reasonable for `let` expressions, it would be highly undesirable for `case` expression, where it is easy to imagine an example where speculative execution leads to infinite recursion.

Another way to eliminate the case of unmatched expressions is to introduce an exhaustiveness checker to Hazel; then, we can guarantee (at run-time) that a pattern will never fail to match. This would also require changing the semantics of pattern holes, which always fail to match; the behavior may be changed so that pattern holes always match, but do not introduce new bindings. Since the focus of this work is not on patterns, these ideas were not explored and are left for future work in the Hazel project.

4.3 The postprocessing substitution algorithm ($\Uparrow_{[]}$)

The postprocessing process aims to perform substitution on expressions that lie outside the evaluation boundary in the evaluation result (an internal expression). The algorithm works in two stages: first inside the evaluation boundary, and then proceeding outside when necessary in closures.

The symbol chosen to denote postprocessing is $\Uparrow_{[]}$. The choice of symbol is somewhat arbitrary, but we may read it as "reverting" some expressions generated by and useful for evaluation (i.e., closures) to a more context-inspector-friendly form, which is in some sense the opposite of evaluation ($\Downarrow$). The bracket subscript indicates that this post-processing step is intended to remove closure expressions. The two stages of this algorithm will be denoted $\Uparrow_{[],1}$ and $\Uparrow_{[],2}$, respectively.

[12]Ordinarily, a lookup on a `BoundVar` (a variable which is in scope) should never fail during evaluation, and thus throws an exception during evaluation.

4.3.1 Substitution within the evaluation boundary ($\Uparrow_{[],1}$)

When inside the evaluation boundary, all (bound) variables have been looked up and all hole environments assigned, so there is no need for a stored environment (as there is in a closure). The main point of this step is to recurse through the expression until a closure is found, at which point we enter the second stage.

For primary expressions (expressions without subexpressions), the expression is returned unchanged; there is nothing to do. For other non-closure expression types, $\Uparrow_{[],1}$ recurses through any subexpressions.

For closure types, we first need to recusively apply $\Uparrow_{[],1}$ to all bindings in the closure environment. For (non-empty) holes, the body is inside the evaluation boundary and thus $\Uparrow_{[],1}$ is applied. For other expressions, the body expression is outside the evaluation boundary, and thus $\Uparrow_{[],2}$ is applied to the body expression, using the closure environment. The closure is then removed.

A λ abstraction, `let` expression, `case` expression, or hole outside of a closure, or a bound variable that has not been looked up, will never exist outside of a closure within the evaluation boundary, so these cases need not be handled.

Note that in the implementation with recursive data structures used to represent environments as described in Section 4.1.3, an additional step must be taken before recursing into function closures. Recursive function bindings must be detected and converted to `FixF` expressions to prevent infinite recursion.

4.3.2 Substitution outside the evaluation boundary ($\Uparrow_{[],2}$)

When outside the evaluation boundary (and inside a closure), we need to substitute bound variables[13] and assign an environment to holes.

Bound variables are looked up in the environment; this lookup may fail if the variable does not exist in the environment, in which case the variable is left unchanged. For other primary expressions, the expression is left unchanged. When a hole is encountered, its environment is the

[13]The wording is a little tricky here, since there are the `BoundVar` and `FreeVar` internal expression variants, which refer to variables which are in scope or not in scope. However, we may only substitute variables which are in-scope (`BoundVar`) and bound; some instances may not yet be bound.

closure environment[14]. A closure will never exist outside the evaluation boundary in the evaluation result.

Note that the $\Uparrow_{[],1}$ algorithm only takes an internal expression d as its input, whereas the $\Uparrow_{[],2}$ algorithm takes an internal expression d and a (closure) environment σ as inputs.

4.4 Post-processing memoization

We may wonder if there is repeated processing if the same closure environment is encountered multiple times in the evaluation result. If we can identify and look up environments, then we can memoize their postprocessing.

4.4.1 Modifications to the environment datatype

Memoization of environments requires a unique key for each environment. The existing environment type `Environment.t` is a map $\sigma = x \mapsto d$. We introduce a new environment type `EvalEnv.t`[15] that is the product of an identifier and the variable map $\sigma = (\text{id}_\sigma, x \mapsto d)$, in which id_σ indicates a unique environment identifier.

To ensure that there is a bijection between environment identifiers and environments, a new unique identifier must be generated each time an environment is extended. An instance of `EvalEnvIdGen.t` is used to generate a new unique identifier, and is required as an additional argument to functions in the `EvalEnv` module that modify the environment[16].

Note that while physical identity may be used to distinguish between different environments, it is difficult to use for efficient lookups due to the abstraction of pointers in a high-level language like OCaml or Javascript. We may think of numeric identifiers (in general) as high-level pointers. We may state this property of environment identifiers as a metatheorem, which allows us to use environment identifiers as a key for environments.

[14]There is nothing to do at this point for hole closures. The hole closure numbering step will assign a closure identifier to the hole as described in the second postprocessing algorithm in Section 5.4.

[15]This is the name in the current implementation (due to this environment type being specialized for evaluation), but perhaps a better name is `MemoEnv.t`.

[16]In the same manner as `MetaVarGen.t`, `EvalEnvId.t` is implemented as type `int` and `EvalEnvIdGen.t` is implemented as a simple counter. To keep the implementation pure, the instance of `EvalEnvIdGen.t` needs to be threaded through all calls of `Evaluator.evaluate` to avoid a global mutable state, and is discussed in Section 2.1.2.

TODO: this needs to be updated/corrected

$\boxed{\sigma \vdash d \Uparrow_{\llparenthesis\rrparenthesis} d'}$ d postprocess-evaluates (λ-conversion) to d' outside the evaluation boundary

$$\frac{d\ \mathsf{value} \qquad d \neq \lambda x.d}{d \Uparrow_{\llparenthesis\rrparenthesis} d}\ \text{PPO}_{\llparenthesis\rrparenthesis}\text{-Value} \qquad \frac{}{\sigma, x \leftarrow d \vdash x \Uparrow_{\llparenthesis\rrparenthesis} d}\ \text{PPO}_{\llparenthesis\rrparenthesis}\text{-Var}$$

$$\frac{\sigma \vdash d \Uparrow_{\llparenthesis\rrparenthesis} d'}{\sigma \vdash \text{fix}\, f.d \Uparrow_{\llparenthesis\rrparenthesis} \text{fix}\, f.d'}\ \text{PPO}_{\llparenthesis\rrparenthesis}\text{-Fix} \qquad \frac{\sigma \vdash d \Uparrow_{\llparenthesis\rrparenthesis} d'}{\sigma \vdash \lambda x.d \Uparrow_{\llparenthesis\rrparenthesis} \lambda x.d'}\ \text{PPO}_{\llparenthesis\rrparenthesis}\text{-Lam}$$

$$\frac{\sigma \vdash d_1 \Uparrow_{\llparenthesis\rrparenthesis} d_1' \qquad \sigma \vdash d_2 \Uparrow_{\llparenthesis\rrparenthesis} d_2'}{\sigma \vdash d_1(d_2) \Uparrow_{\llparenthesis\rrparenthesis} d_1'(d_2')}\ \text{PPO}_{\llparenthesis\rrparenthesis}\text{-Ap} \qquad \frac{\sigma \vdash d_1 \Uparrow_{\llparenthesis\rrparenthesis} d_1' \qquad \sigma \vdash d_2 \Uparrow_{\llparenthesis\rrparenthesis} d_2'}{\sigma \vdash d_1 + d_2 \Uparrow_{\llparenthesis\rrparenthesis} d_1' + d_2'}\ \text{PPO}_{\llparenthesis\rrparenthesis}\text{-Op}$$

$$\frac{}{\sigma \vdash \llparenthesis\rrparenthesis^u_\varnothing \Uparrow_{\llparenthesis\rrparenthesis} \llparenthesis\rrparenthesis^u_\sigma}\ \text{PPO}_{\llparenthesis\rrparenthesis}\text{-EHole} \qquad \frac{\sigma \vdash d \Uparrow_{\llparenthesis\rrparenthesis} d'}{\sigma \vdash \llparenthesis d \rrparenthesis^u_\varnothing \Uparrow_{\llparenthesis\rrparenthesis} \llparenthesis d' \rrparenthesis^u_\sigma}\ \text{PPO}_{\llparenthesis\rrparenthesis}\text{-NEHole}$$

$\boxed{d \Uparrow_{\llparenthesis\rrparenthesis} d'}$ d postprocess-evaluates (λ-conversion) to d' within the evaluation boundary

$$\frac{d\ \mathsf{value} \qquad d \neq \text{fix}\, f.d \qquad d \neq [\sigma]\lambda x.d}{d \Uparrow_{\llparenthesis\rrparenthesis} d}\ \text{PPI}_{\llparenthesis\rrparenthesis}\text{-Value}$$

$$\frac{\sigma \vdash d \Uparrow_{\llparenthesis\rrparenthesis} d' \qquad \sigma, f \leftarrow (\text{fix}\, f.\lambda x.d') \vdash d' \Uparrow_{\llparenthesis\rrparenthesis} d''}{\text{fix}\, f.([\sigma]\lambda x.d) \Uparrow_{\llparenthesis\rrparenthesis} \lambda x.d''}\ \text{PPI}_{\llparenthesis\rrparenthesis}\text{-Fix} \qquad \frac{\sigma \vdash d \Uparrow_{\llparenthesis\rrparenthesis} d'}{[\sigma]\lambda x.d \Uparrow_{\llparenthesis\rrparenthesis} \lambda x.d'}\ \text{PPI}_{\llparenthesis\rrparenthesis}\text{-Closure}$$

$$\frac{d_1 \Uparrow_{\llparenthesis\rrparenthesis} d_1' \qquad d_2 \Uparrow_{\llparenthesis\rrparenthesis} d_2'}{d_1(d_2) \Uparrow_{\llparenthesis\rrparenthesis} d_1'(d_2')}\ \text{PPI}_{\llparenthesis\rrparenthesis}\text{-Ap} \qquad \frac{d_1 \Uparrow_{\llparenthesis\rrparenthesis} d_1' \qquad d_2 \Uparrow_{\llparenthesis\rrparenthesis} d_2'}{d_1 + d_2 \Uparrow_{\llparenthesis\rrparenthesis} d_1' + d_2'}\ \text{PPI}_{\llparenthesis\rrparenthesis}\text{-Op}$$

$$\frac{\sigma' = \{(x \leftarrow d') : (x \leftarrow d) \in \sigma, d \Uparrow_{\llparenthesis\rrparenthesis} d'\}}{\llparenthesis\rrparenthesis^u_\sigma \Uparrow_{\llparenthesis\rrparenthesis} \llparenthesis\rrparenthesis^u_{\sigma'}}\ \text{PPI}_{\llparenthesis\rrparenthesis}\text{-EHole}$$

$$\frac{d \Uparrow_{\llparenthesis\rrparenthesis} d' \qquad \sigma' = \{(x \leftarrow d') : (x \leftarrow d) \in \sigma, d \Uparrow_{\llparenthesis\rrparenthesis} d'\}}{\llparenthesis d \rrparenthesis^u_\sigma \Uparrow_{\llparenthesis\rrparenthesis} \llparenthesis d' \rrparenthesis^u_{\sigma'}}\ \text{PPI}_{\llparenthesis\rrparenthesis}\text{-NEHole}$$

TODO: closure needs to go recursive

Figure 5: Big-step semantics for λ-conversion post-processing

$\boxed{\mathit{TODO}}$ TODO

$$\mathit{TODO}$$

Figure 6: Big-step semantics modifications for environment memoization

Theorem 4.1 (Use of id_σ as an identifier)**.** *The mapping $i_\sigma : \sigma \mapsto id_\sigma$ that maps an environment (identified up to physical equality) to its assigned environment identifier is a bijection.*

Proof. The proof of injectivity and surjectivity are shown by construction. The relation is surjective because a new identifier is only assigned when a new environment is created. To prove injectivity, we intuit that $\sigma_i \neq \sigma_j$ implies that there is a series of modified environments $\{\sigma_i, \sigma_{i+1}, \dots, \sigma_{j-1}, \sigma_j\}$ (without loss of generality, assume σ_i is an earlier environment than σ_j). By construction, each element of the set $\{i_\sigma(\sigma_i), i_\sigma(\sigma_{i+1}), \dots, i_\sigma(\sigma_j)\}$ is unique. Thus $i_\sigma(\sigma_1) \neq i_\sigma(\sigma_2)$. $\square$

4.4.2 Modifications to the post-processing rules

During substitution postprocessing ($\Uparrow_{[]}$), a mapping $\mathrm{id}_\sigma \mapsto \sigma$ stores the set of substituted (post-processed) environments. Upon encountering a closure in the evaluation result, it is looked up in this map. If it is found, the stored result is used. If it is not found, the environment is recursively substituted by applying $\Uparrow_{[],1}$ to each binding.

4.5 Implementation considerations

This section details various design decisions and tradeoffs of the current implementation; some parts of this may require an understanding of the hole closure numbering postprocessing step described in Section 5.

4.5.1 Purity

The purity of implementation is a recurring theme. While it should not affect the capability of the implementation, there is a strong urge to keep the implementation pure. Elegance, complexity, and runtime overhead is traded off for purity. The main decisions regarding purity are summarized here, and left for the consideration of future implementors.

One offender of performance is the use of the fixpoint form when evaluating recursive functions. This involves extra evaluation steps for unwrapping fixpoints, and can be avoided with self-referential data structures, and more easily implemented using `ref`s.

An offender of elegance is the threading of the identifier generator around for memoized environments (`EvalIdGen.t`). This can be much more easily implemented as a simple global counter; instead, it is passed to and returned from every call of the the core evaluator function (`Evaluator.evaluate`), adding much clutter. The same is true for the generator for hole identifiers (`MetaVarGen.t`).

4.5.2 Data structures

As is common in functional programming, the most common data structures used are (linked) lists and maps (binary search trees). The standard library modules `List` and `Map` are used for these. In particular, the original implementation uses linked-lists for the implementation of environments, and we have not modified this decision. In Hazel, The hole closure storage data structures `HoleClosureInfo_.t` and `HoleClosureInfo.t` use a combination of maps and lists. Hashtables were not used at all in the implementation; their effect on performance is unknown and is reserved for future work.

4.5.3 Additional constraints due to hole closure numbering

The introduction of hole closure parents in Section 5.3.1 makes closure memoization more difficult for environments in non-hole closures. In particular, adding a new parent to a hole requires that the hole postprocessing (the hole closure numbering operation) be re-run on a hole. Memoizing the hole prevents a hole closure in an environment from being assigned multiple closure parents.

In fact, the memoization operation is only implemented on a per-hole-closure basis. This is due to a number of factors: an additional data structure is required to keep track of memeoized environments, and a very similar data structure to `HoleClosureInfo.t` (`HoleInstanceInfo.t`) already existed in the codebase for the hole instance numbering operation; memoization of environments was initially intended to solve the performance issue for hole numbering postprocessing step described in Section 5.2, and memoization was bootstrapped to the substitution postprocessing step as well; and the issue with hole parents mentioned in the previous paragraph.

To summarize, the current state of the implementation involves environments with unique identifiers so as to be more amenable to memoization, but the memoization during postprocessing is only performed for hole closures (i.e., the postprocessing will only not be repeated if the same hole number and environment are encountered multiple times, but will be repeated if the same environment occurs in different holes or in non-hole closures). For the sake of time, fully memoizing all environments and investigating the effects is left for future work, although the marginal benefit may not be very great[17].

4.5.4 Storing evaluation results versus internal expressions

The evaluation takes as input an internal expression and returns the evaluated internal expression along with a final judgment (either `BoxedValue` or `Indet`).

The decision should be made whether to store this final judgment in the environment[18]. Storing the judgment allows us to simply use the stored value directly during evaluation, but requires much boxing and unboxing in other cases (e.g., during postprocessing). On the other hand, not storing the judgment is cleaner when used outside of evaluation, but requires recalculation of the final judgment during evaluation upon lookup[19]. The decision is somewhat arbitrary but may have

[17]We may offer the following intuition for this claim. The issue of exponential hole instance exponential blowup described in Section 5.2 is solved by memoizing hole environments, which has a clear benefit.

Let us consider the other cases of repeated environments. Note that these include non-hole closures with the same environment or in hole closures with a different hole number. Also, note that an environment may only be shared by expressions which are not separated by any binders, which can be very roughly characterized as the length of an infix expression omitting `\lambda` abstraction, `let` expression, and `case` expression bodies. Firstly, we do not expect such expressions to be very long for a user prototyping a program in Hazel (rather, long expressions are more likely to be broken down into a series of `let` expressions), whereas a long linear set of `let` expressions to store intermediate values is very common in scripting. Secondly, the number of repetitions is only linear with respect to the number of times the environment is repeated, as opposed to the issue with non-memoized hole environments, in which the issue is exponential with respect to the level of repeated bindings. This is due to the fact that repeated hole closures in the environment will still be memoized, so the amount of repeated postprocessing does not recurse into children holes and cause the exponential blowup.

[18]In other words, we need to decide whether `EvalEnv.t` should be a mapping from variables to `EvalEnv.result` (including final judgment) or from variables to `DHExp.t`.

[19]Recalculating the final judgment means re-evaluating the expression upon variable lookup, since the `Evaluator.evaluate` function currently performs the evaluation and final judgments. This should not be an expensive operation since the value should already be final and cannot make any evaluation steps, but still may require

small effects on the evaluation performance and elegance of implementation.

several calls to evaluate.

```
let a = ⦇⦈^1 in
let b = λ x . { ⦇⦈^2 } in
f 3 + f 4
```

Listing 1: Illustration of hole instances

5 Memoizing hole instance numbering using environments

TODO: this chapter is currently a quick dump of handwritten stuff to typed text; need to partition this into sections

5.1 Rationale behind hole instances and unique hole closures

Consider the program displayed in Listing 1. The evaluation result of the program is

$$[a \leftarrow [\varnothing]\llparenthesis\rrparenthesis^1, x \leftarrow 3]\llparenthesis\rrparenthesis^2 + [a \leftarrow [\varnothing]\llparenthesis\rrparenthesis^1, x \leftarrow 4]\llparenthesis\rrparenthesis^2$$

Note that the two instances of $\llparenthesis\rrparenthesis^2$ have different environments, adn we thus distinguish between the two occurrences of $\llparenthesis\rrparenthesis^2$ as separate **instances** of a hole. However, note that while there are also two instances of the hole $\llparenthesis\rrparenthesis^1$ in the result, these share the same (physically equal) environment. No matter what expression we fill hole $\llparenthesis\rrparenthesis^1$ with (for example, using the fill-and-resume operation) the hole will evaluate to the same value. This differs from the hole $\llparenthesis\rrparenthesis^2$, whose filling may cause different instances to evaluate to different values due to non-capture-avoiding substitution. For example, filling hole $\llparenthesis\rrparenthesis^2$ with the expression $x + 2$ will cause the instances to resolve to 5 and 6, respectively.

The current implementation assigns an identifier i to each instance of a hole, and the instance number is unique between all instances of a hole. While this makes perfect sense for $\llparenthesis\rrparenthesis^2$, the assignment of two separate holes to $\llparenthesis\rrparenthesis^1$ may confuse Hazel users, since these hole instances are identical and filling them with any value will result in the same value. The solution is to unify all instances of a hole which share the same (physically equal) environment, and thus identify hole instances by hole number and environment. A set of hole instances that share the same environment

```
let f = λ x . { ⦇⦈¹ } in
f 2 + f 2
```

Listing 2: Illustration of physical equality for environment memoization

will be called a **unique hole closure**, or simply **hole closure**[20].

To illustrate why physical equality is used to identify environments, consider the case shown in Listing 2. This simpler program evaluates to

$$[x \leftarrow 2]⦇⦈^1 + [x \leftarrow 2]⦇⦈^1$$

In this case, hole 1 has two instances with two environments with structurally equal bindings. If the argument to the second invocation of f is changed to 3, then the holes will have different environments and may thus fill to different values. This may be confusing to the Hazel user; what appears to be a single hole closure is actually two different hole closures which incidentally have the same values bound to its variables.

An intuitive way of understanding the use of physical equality is that separate *instantiations* of the same hole should be distinguished. This is highly related to function applications. A hole may only appear multiple times in the result in two different ways: it may exist in the body of a function that is multiple times (multiple hole instantiations), or it may appear in a hole that is referenced from other holes (shared hole instantiation). An implication of this is that the values bound to an environment do not affect whether it is distinguished from another hole closure.

5.2 Issues with the current implementation

Consider the program shown in Listing 3.

A performance issue appears with the existing evaluator with the program shown in Listing 4.

[20]"Hole closure" also is used to describe the generalized closure around hole expressions as described in Section 4. Here we are referring to the set of instances of the same hole that share the same physical environment. Hence we call this interpretation "unique hole closure" to distinguish it from the former interpretation, but the interpretation should be clear from context.

$\boxed{H, p \vdash d \Uparrow_{i,d} (H', d')}$ Hole instance numbering in expression d with hole instance info H

$$\frac{d \ \mathsf{value} \qquad d \neq \lambda x.d}{H, p \vdash d \Uparrow_{i,d} (H, d)}\ \text{PP}_{i,d}\text{-Value} \qquad\qquad \frac{}{H, p \vdash x \Uparrow_{i,d} (H, x)}\ \text{PP}_{i,d}\text{-Var}$$

$$\frac{H, p \vdash d \Uparrow_{i,d} (H', d')}{H, p \vdash \lambda x.d \Uparrow_{i,d} (H', d')}\ \text{PP}_{i,d}\text{-Lam}$$

$$\frac{Hp \vdash d_1 \Uparrow_{i,d} (H', d_1') \qquad H', p \vdash d_2 \Uparrow_{i,d} (H'', d_2')}{H, p \vdash \lambda d_1(d_2) \Uparrow_{i,d} (H'', d_1'(d_2'))}\ \text{PP}_{i,d}\text{-Ap}$$

$$\frac{H, p \vdash d_1 \Uparrow_{i,d} (H', d_1') \qquad H', p \vdash d_2 \Uparrow_{i,d} (H'', d_2')}{H, p \vdash \lambda d_1 + d_2 \Uparrow_{i,d} (H'', d_1' + d_2')}\ \text{PP}_{i,d}\text{-Op}$$

$$\frac{\text{hid}(H, u) = i \qquad H' = H, (u, i, \_, p)}{H, p \vdash \llparenthesis\rrparenthesis^{u}_{\sigma} \Uparrow_{i,d} (H', \llparenthesis\rrparenthesis^{u:i}_{\sigma})}\ \text{PP}_{i,d}\text{-EHole}$$

$$\frac{\text{hid}(H, u) = i \qquad H' = H, (u, i, \_, p) \qquad H', p \vdash d \Uparrow_{i,d} (H'', d')}{H, p \vdash \llparenthesis d \rrparenthesis^{u}_{\sigma} \Uparrow_{i,d} (H'', \llparenthesis d' \rrparenthesis^{u:i}_{\sigma})}\ \text{PP}_{i,d}\text{-NEHole}$$

$\boxed{H, p \vdash d \Uparrow_{i,\sigma} (H', d')}$ Hole instance numbering in hole envs in d with hole instance info H

$$\frac{d \ \mathsf{value} \qquad d \neq \lambda x.d}{H, p \vdash d \Uparrow_{i,\sigma} (H, d)}\ \text{PP}_{i,\sigma}\text{-Value} \qquad\qquad \frac{}{H, p \vdash x \Uparrow_{i,\sigma} (H, x)}\ \text{PP}_{i,\sigma}\text{-Var}$$

$$\frac{H, p \vdash d \Uparrow_{i,\sigma} (H', d')}{H, p \vdash \lambda x.d \Uparrow_{i,\sigma} (H', d')}\ \text{PP}_{i,\sigma}\text{-Lam}$$

$$\frac{H, p \vdash d_1 \Uparrow_{i,\sigma} (H', d_1') \qquad H', p \vdash d_2 \Uparrow_{i,\sigma} (H'', d_2')}{H, p \vdash \lambda d_1(d_2) \Uparrow_{i,\sigma} (H'', d_1'(d_2'))}\ \text{PP}_{i,\sigma}\text{-Ap}$$

$$\frac{H, p \vdash d_1 \Uparrow_{i,\sigma} (H', d_1') \qquad H', p \vdash d_2 \Uparrow_{i,\sigma} (H'', d_2')}{H, p \vdash \lambda d_1 + d_2 \Uparrow_{i,\sigma} (H'', d_1' + d_2')}\ \text{PP}_{i,\sigma}\text{-Op}$$

$$\frac{\begin{array}{c} H, p, u, i \vdash \sigma \Uparrow_{i,d} (H', \sigma') \\ H', p, u, i \vdash \sigma' \Uparrow_{i,\sigma} (H'', \sigma'') \qquad H''' = H'', (u, i, \sigma'', p') \end{array}}{(H, (u, i, \_, p')), p \vdash \llparenthesis\rrparenthesis^{u:i}_{\sigma} \Uparrow_{i,\sigma} (H''', \llparenthesis\rrparenthesis^{u:i}_{\sigma''})}\ \text{PP}_{i,\sigma}\text{-EHole}$$

$$\frac{\begin{array}{c} H, p, u, i \vdash d \Uparrow_{i,\sigma} (H', d') \qquad H', p, u, i \vdash \sigma \Uparrow_{i,d} (H'', \sigma') \\ H'', p \vdash \sigma' \Uparrow_{i,\sigma} (H''', \sigma'') \qquad H'''' = H''', (u, i, \sigma'', p') \end{array}}{(H, (u, i, \_, p')), p \vdash \llparenthesis d \rrparenthesis^{u:i}_{\sigma} \Uparrow_{i,\sigma} (H'''', \llparenthesis d' \rrparenthesis^{u:i}_{\sigma''})}\ \text{PP}_{i,\sigma}\text{-NEHole}$$

$\boxed{H, p, u, i \vdash \sigma \Uparrow_{i,d} (H', \sigma')}$ Hole instance numbering in hole environment σ with HII H

$$\frac{}{H, p, u, i \vdash \varnothing \Uparrow_{i,d} (H, \varnothing)} \text{ PP}_{i,d}\text{-TrivEnv}$$

$$\frac{H, p, u, i \vdash \sigma \Uparrow_{i,d} (H', \sigma') \qquad H', (p, (x, (u, i))) \vdash d \Uparrow_{i,d} (H'', d')}{H, p, u, i \vdash \sigma, x \leftarrow d \Uparrow_{i,d} (H'', (\sigma', x \leftarrow d'))} \text{ PP}_{i,d}\text{-Env}$$

$\boxed{H, p, u, i \vdash \sigma \Uparrow_{i,\sigma} (H', \sigma')}$ Hole instance numbering in hole environment σ with HII H

$$\frac{}{H, p, u, i \vdash \varnothing \Uparrow_{i,\sigma} (H, \varnothing)} \text{ PP}_{i,\sigma}\text{-TrivEnv}$$

$$\frac{H, p, u, i \vdash \sigma \Uparrow_{i,\sigma} (H', \sigma') \qquad H', (p, (x, (u, i))) \vdash d \Uparrow_{i,\sigma} (H'', d')}{H, p, u, i \vdash \sigma, x \leftarrow d \Uparrow_{i,\sigma} (H'', (\sigma', x \leftarrow d'))} \text{ PP}_{i,\sigma}\text{-Env}$$

$\boxed{d \Uparrow_i (H', \sigma')}$ Hole instance numbering in expression d and subexpressions

$$\frac{\varnothing, \varnothing \vdash d \Uparrow_{i,d} (H, d') \qquad H, \varnothing \vdash d' \Uparrow_{i,d} (H', d'')}{d \Uparrow_i (H', d'')} \text{ PP}_i\text{-Root}$$

Figure 7: Big-step semantics for the previous hole instance numbering algorithm

```
let a = ⦇⦈¹ in
let b = λ x . { a + x + ⦇⦈² } in
let c = ⦇⦈³ in
⦇⦈⁴ + b 1 + f ⦇⦈⁵
```

Listing 3: A seemingly innocuous Hazel program

```
let a = ⦇⦈^1 in
let b = ⦇⦈^2 in
let c = ⦇⦈^3 in
let d = ⦇⦈^4 in
let e = ⦇⦈^5 in
let f = ⦇⦈^6 in
let g = ⦇⦈^7 in
...
let x = ⦇⦈^n in
⦇⦈^{n+1}
```

Listing 4: A Hazel program that generates an exponential (2^N) number of total hole instances

$\boxed{\textit{TODO}}$ TODO

TODO

Figure 8: Big-step semantics for hole closure numbering

5.3 Hole instances and closures

5.3.1 Hole instance path versus hole closure parents

5.4 Algorithmic concerns and a two-stage approach

5.5 Memoization and unification with closure post-processing

5.5.1 Modifications to the instance numbering rules

5.5.2 Unification with closure post-processing

5.5.3 Fast evaluation result structural equality checking

5.6 Differences in the hole instance numbering

$\boxed{d \Uparrow (H, d')}$ d postprocess-evaluates to d' with hole closure info H

$$\frac{d \Uparrow_{[]} d' \qquad d' \Uparrow_i (H, d'')}{d \Uparrow (H, d'')}\text{PP-Result}$$

Figure 9: Big-step semantics for post-processing

6 Implementation of fill-and-resume

6.1 CMTT interpretation of fill-and-resume

6.2 Memoization of recent actions

6.3 UI changes for notebook-like editing

```
let f : Int → Int =
  λ x . {
    case x of
    | 0 ⇒ 0
    | 1 ⇒ 1
    | n ⇒ f (n - 1) + f (n - 2)
    end
  }
in f 25
```

Listing 5: An evaluation-heavy Hazel program with no holes

7 Evaluation of methods

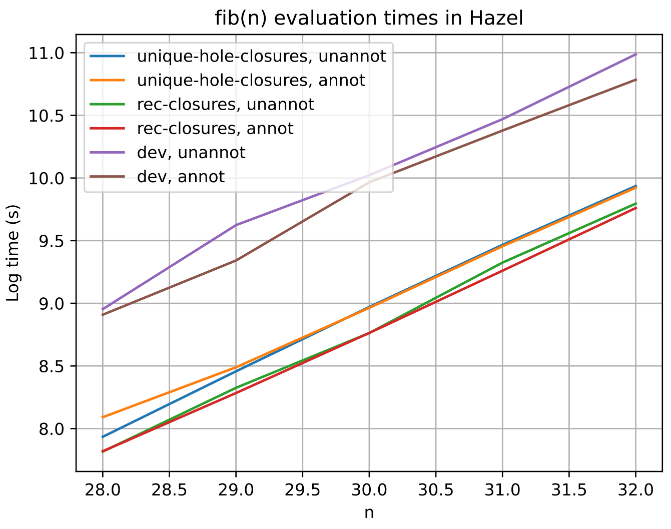


Figure 10: Performance of the different models of evaluation

8 Future work

8.1 Mechanization of metatheorems and rules

8.2 FAR for all edits

8.3 Stateless and efficient notebook environment

9 Conclusions and recommendations

10 References

[1] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. Live functional programming with typed holes. *PACMPL*, 3(POPL), 2019.

[2] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. Hazelnut: A Bidirectionally Typed Structure Editor Calculus. In *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*, 2017.

[3] Jérôme Vouillon and Vincent Balat. From bytecode to javascript: the js_of_ocaml compiler. *Software: Practice and Experience*, 44(8):951–972, 2014.

A Additional contributions to Hazel

A.1 Additional performance improvements

A.2 Documentation and learning efforts

B Code correspondence

This section aims to provide extra information about how concepts presented in this paper correspond to constructs in the source code.

C Related concurrent research directions in Hazel

This appendix lists various subdivisions of Hazel that may be affected by the changes described in this paper

C.1 Hole and hole instance numbering

C.1.1 Improved hole renumbering

C.2 Performance enhancements

C.2.1 Evaluation limits

C.2.2 Hazel compiler

C.3 Agda Formalization

D Selected code samples