THE COOPER UNION FOR THE ADVANCEMENT OF SCIENCE AND ART
ALBERT NERKEN SCHOOL OF ENGINEERING

Implementation of performance optimizations to the Hazel live 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

ACKNOWLEDGMENTS

I thank my advisor Professor Fred Fontaine, for keeping me on track and supporting me in my efforts to do research even as I diverge from EE into theoretical CS. Professor Fontaine has also been extremely generous in assisting me as my academic advisor of four years and in inviting me as an instructor for the ECE-210 course, all while serving as the Chair of Electrical Engineering and encouraging all my peers in the EE major to excel. I thank my co-advisor Professor Robert Marano, for sharing the passion in software engineering research and helping guide me towards research in programming education. I thank Professor Cyrus Omar at the University of Michigan, as well as the other members of the FPLab, for providing the technical mentorship for this project and allowing me access to your time and advice, despite the unconventional collaboration. The openness for an outsider such as myself was pivotal in allowing me to learn and accomplish as much as I was able to for this project.

I thank my peers in electrical engineering, for supporting me all this way. Among many peers, I thank the members of the 10B organization, for being my closest friends since the beginning of our shared Cooper experience and for the continual and occasionally flamboyantly-terrorizing level of support. I also thank Victor Zhang and Derek Lee for sharing the experience of various independent studies, senior projects, programming competitions, job applications, and the dual-degree program, all over the course of the past year.

Lastly, and most importantly, I thank my family, who have allowed me to spend all my efforts in school unhindered for the greater part of the past eight years, even through the panic of a pandemic. I thank my parents for the immense support in this process, and my sisters Jessica, Juliet, Josie, Sharon, and Jane for their emotional support. I thank the numerous family members that have allowed me to reside with them during my stay in the city: my grandma, Uncle Frank, Ben, Uncle John, and Aunt Amy. I hope I can repay your generosity someday.

ABSTRACT

Hazel is a live programming environment with typed holes that serves as a reference implementation of the Hazelnut Live dynamic semantics [1] and the Hazelnut static semantics [2], both of which tackle the "gap problem." This work attempts to further develop the Hazel evaluation model by implementing the environment model of evaluation (as opposed to the current substitution model) and memoizing several evaluation-related operations to improve performance. Additionally, we provide an implementation-level description and a reference implementation of the fill-and-resume (FAR) performance optimization proposed in Hazelnut Live. We produce a metatheory and reference implementation of the proposed changes. Our implementation is benchmarked against the existing Hazel implementation to show that the results match the expectation, although there is room for future improvement with the development with memoization. Finally, we discuss some useful theoretical generalizations that result from this work.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

LIST OF THEOREMS

LIST OF NOTATIONS

e	(External) expression
τ	Type
Γ	Typing context
Δ	Hole context
$\Gamma \vdash e : \tau$	Type judgment
$\Gamma \vdash e \Rightarrow \tau$	Synthetic type judgment
$\Gamma \vdash e \Leftarrow \tau$	Analytic type judgment
$[e'/x]e$	Substitution of e' for x in e

Table 1: Common notation for the λ-calculus

Λ	Untyped λ-calculus
$\Lambda_{\to}$	Simply-typed λ-calculus
$\Lambda^{?}_{\to}$	Gradually-typed λ-calculus
$\Lambda^{⦶}_{\to}$	λ-calculus with typed holes

Table 2: The λ-calculi

$*$	Unknown type
$\tau \sim \tau'$	Type consistency
$\tau \blacktriangleright_{\to} \tau_1 \to \tau_2$	Matched arrow judgment
$\Gamma \vdash e \leadsto d : \tau$	Elaboration judgment
$d\langle \tau \Rightarrow \tau' \rangle$	Cast expression

Table 3: The gradually-typed λ-calculus

d	Internal expression
$\lambda x : \tau.d$	λ-abstraction
$\mathrm{fix}\, f : \tau.d$	Fixpoint form
σ	Environment
x, f	Variable name
u	Hole number
i	Hole instance or closure number
$⦇⦈^{u:i}$	Empty hole expression
$⦇d⦈^{u:i}$	Non-empty hole expression
$[\sigma]d$	Generalized closure

Table 4: Hazel internal language

d value	Value
d final	Final
$\sigma \vdash d \Downarrow d'$	Evaluation

Table 5: Hazel evaluation judgments

H	Hole instance/closure information
hid	Hole instance/closure id generation function
p	Hole closure parent
$d \Uparrow d'$	Postprocessing
$d \Uparrow_{[]} d'$	Postprocessing (λ-conversion)
$H, p \vdash d \Uparrow_i d' \dashv H'$	Postprocessing (hole closure numbering)

Table 6: Hazel postprocessing judgments

$d_1 \trianglerighteq d_2$	No diff (empty diff) between d_1 and d_2
$d_1 \triangleright d_2$	Non-fill diff from d_1 to d_2
$d_1 \blacktriangleright_d^u d_2$	Fill diff from d_1 to d_2 with fill parameters u and d
$d_1 \not\trianglerighteq_d^u d_2$	Some (non-empty) diff from d_1 to d_2
$d_1 \triangleright_d^u d_2$	Any (possibly empty) diff from d_1 to d_2
$d_1 \sim d_2$	Expression form equality

Table 7: Fill-and-resume structural diff algorithm

$[\![u/d]\!]d' = d''$	Fill-and-resume operation on an expression
$[\![u/d]\!]\sigma = \sigma'$	Fill-and-resume operation on an environment
$[\![\sigma]\!]d$	Closure with re-eval flag set
$(\!\vert d \vert\!)_i$	Expression d filled in hole with hole closure number i
ρ	Fill memoization context
$\sigma, \rho \vdash d \Downarrow d' \dashv \rho'$	Fill-memoized evaluation

Table 8: Fill-and-resume pre-processing and evaluation

Chapter 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. Structural editors, on the other hand, only allow valid edit states. Several structural editors, such as Scratch [4], Lamdu [5], and mbeddr [6], have been proposed to improve the programming experience and introduce editor services, such as the elimination of syntax errors or graphical editing.

Hazel [7] 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 [1], which governs the structural editor and the static semantics (typing rules) of the language. The dynamic semantics (evaluation semantics) are described in Hazelnut Live [2].

Hazel is a relatively new research effort by the University of Michigan's Future of Programming Lab (FPLab), with little effort placed on performance optimizations. This work attempts to achieve several enhancements that will benefit the performance of evaluation and related tasks. Part of the work will be focused on transitioning the evaluation model from

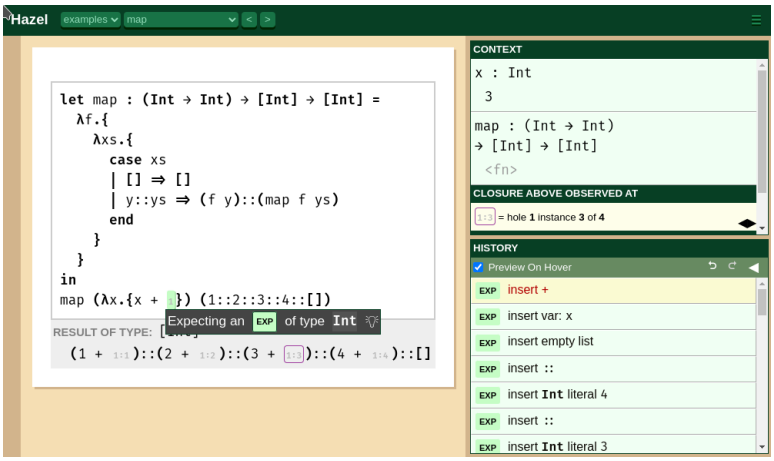


Figure 1.1: A screenshot of the Hazel live programming environment. Screenshot taken of the dev branch demo on 02/06/2022 [3].

using substitution for variable bindings to using environments, with 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 specific to Hazel (such as hole closure numbering), and also implement the fill-and-resume performance enhancement described in [2]. The novelty of this work lies in the optimization capacity in the unique design of the Hazel language as a live programming editor with expression holes.

1.2 The contribution of this work

This thesis presents several algorithms designed for Hazel's evaluation. These algorithms will be provided using the big-step inference rule notation introduced in Section 2.1.2.

- The evaluation semantics of the Hazel language using the environment model. We aim to keep the implementation pure, introduce uniquely-numbered environments (for later use in memoization), and describe the evaluation of holes (which are unique to Hazel). We introduce the concepts of generalized closures, and the evaluation boundary.

- Postprocessing, which is mostly memoized by environments and has the two major

functions. The first is to convert the result to the equivalent result if substitution was used. The second is to number hole closure instances.

- Fill-and-resume, as originally proposed in [2]. This algorithm is described at a high level in the original description, but an implementation was not provided. We provide a possible implementation, including an algorithm to detect a valid fill operation and advice on memoizing the resumption operation.

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 [1] and Hazelnut Live [2]. Instead, we provide a series of core metatheorems describing invariants describing Hazel evaluation, and argue the correctness of these metatheorems by informal reasoning on the provided inference rules. A mechanized proof is deferred for future work.

1.3 Structural overview

Chapter 2 provides a background on necessary topics in programming language (PL) theory and programming language implementations, in order to frame the understanding for the Hazel live programming environment. Chapter 3 provides an overview of Hazel, in order to frame the work completed for this thesis project. Chapters 4 to 6 describe the primary work completed for this project, as described in Section 1.2. Chapter 7 comprises an emprical performance assessment of the work. Chapter 8 is a discussion of theoretical results. Chapter 9 describes unfinished work and future research directions. Chapter 10 concludes with a summary of findings and future work. The Appendices contain extra inference rules and selected source code snippets.

Chapter 2

Programming language principles

This chapter is intended to provide a primer to the theory of functional programming and programming languages, as relevant to this work on Hazel. The work performed for this thesis is concerned with the dynamic semantics of Hazel.

Section 2.1 is concerned with explaining the notation used throughout this paper to describe formal systems. Section 2.2 is concerned with incrementally building up the conceptual foundation for the gradually-typed λ-calculus, from which Hazel heavily borrows. In this section, the syntax and static semantics of the λ-calculus are explored; even though not directly tied to this work concerned with dynamics, these are critical to understanding the Hazel system. Much of the material presented in this section is standard material in a introductory text in programming language theory such as [8]. Section 2.3 provides some detail on different types of programming language implementations, which is standard material in an introductory compilers text such as [9]. In particular, this section sheds some light on the rationale behind switching from an evaluation model based on substitution to an evaluation model based on environments, which forms the basis for a large part of this thesis. Section 2.4 provides an overview of relevant topics in programming language interfaces.

2.1 Specifications of programming languages

To be able to rigorously work with programming languages, as with any mathematical activity, we need to precisely define the behavior of programming languages that serve as our interface to computation. The definition (or specification) of a programming language is typically given as the combination of its *syntax* and *semantics*, which will be discussed below.

Note that the specification of a programming language is orthogonal to its *implementation(s)*; a programming language may have several implementations, which may have differing support for language features and different performance characteristics. Common classifications of programming language implementations are discussed in Section 2.3.1.

2.1.1 Syntax

The syntax of a programming language is defined by a grammar. The grammar described in the original paper on Hazelnut [1] is reproduced below as an example.

$$\tau ::= \tau \to \tau \mid \texttt{num} \mid ⦇⦈$$
$$e ::= x \mid \lambda x.e \mid e\ e \mid e + e \mid e : \tau \mid ⦇⦈ \mid ⦇e⦈$$

In this simple grammar, we have two productions: types and expressions. A type may have one of three forms: the `num` type, an arrow (function) type, or the hole type (similar to the $*$ type from the GTLC described in Section 2.2.4). An expression may be a variable, a λ-abstraction[1], a primitive binary operation, a type ascription, an empty hole, or a non-empty hole. Hole expressions and the hole type are specific to Hazel. Parentheses are not shown in this grammar; they are optional except to specify order of operations.

Parts of this grammar will be revisited when discussing the λ-calculus described in Sec-

[1]This may have several names depending on the context and programming language, such as: λ-function, function literal, arrow function, anonymous function, or simply function. λ-abstractions are unary by definition – higher-arity functions may be constructed via *function currying*.

tion 2.2.2, and when discussing Hazel's grammar described in Chapter 3. In particular, this Hazelnut grammar is a superset of the grammar of the GTLC described in Section 2.2.4, and a subset of the grammar in the implementation of Hazel, which includes additional forms such as `let`, `case`, and pair expressions. Some of these forms will be important cases for our study of evaluation.

Due to Hazelnut being a structural edit calculus (as described in Section 2.4.1), there is no need to worry about syntax errors. The syntax describes the external language of Hazel, which will be translated into the internal language via the elaboration algorithm prior to evaluation.

2.1.2 Notation for semantics

In formal logic, a standard notation for *rules of inference* is shown below.

$$\frac{p_1 \qquad p_2 \qquad \cdots \qquad p_n}{q}\ \text{SampleRule}$$

$p_1, p_2, \ldots, p_n$ are the *antecedents* (alternatively, *premises*) and q is the single *consequent* (alternatively, *conclusion*). Each of $p_1, p_2, \ldots, p_n, q$ is a *judgment* (alternatively, *proposition* or *statement*). We may pronounce this rule as "if all of $p_1, p_2, \ldots, p_n$ are true, then q must be true." Note that the antecedent of the rule is the logical conjunction of the antecedants $\bigwedge_{i=1}^{n} p_i$. The logical disjunction of antecedants $\bigvee_{i=1}^{n} p_i$ is expressed by writing separate rules with the same consequent. A rule with zero premises is an *axiom*, i.e., the conclusion is vacuously true. We may build up a formal logic system using such rules. Note that the set of judgments that form the consequents of a rule, as well as the set of rules in a formal system, are both unordered; however, any computer program that carries out these judgments must choose some order in which to evaluate the set of antecedents or the order in which to evaluate a set of equally-viable rules.

Each new judgment form will be introduced annotated with the modes of each term.

For example, the typing judgment $\Gamma^- \vdash e^- : \tau^+$ indicates that Γ and e are inputs to the judgment and τ is an output of the judgment. If there are no terms with output mode in a judgment, then the ability to logically construct the judgment is its sole (boolean) output.

A derivation (proof) of a judgment is shown by chaining together inference rules, such that the final consequent is the statement to be proved. We may visualize a derivation as a tree rooted at the judgment to be proved, and whose children are (recursively) the antecedent judgments; the leaf nodes of this tree must be axioms.

To ensure that the system of inference rules covers the entire semantics of a language, to ensure that rules to not conflict, and to ensure that rules give the language the desired behavior, we may establish *metatheorems*. Metatheorems are intuitive, high-level invariants or properties that describe the behavior of the overall system. We prove the correctness of an implementation by proving that the metatheorems are upheld by the inference rules. In the foundational papers for Hazel's core semantics [1, 2], metatheorems are amply used to justify and verify the correctness of the rules. Agda [10], an interactive proof checker and dependently-typed programming language, is used for these proofs [11, 12].

2.1.3 Static semantics

The *static semantics* of a programming language describes properties of a program that can be checked prior to program evaluation. Static semantics typically refers primarily to *type checking*. In Hazelnut Live, we have the process of *elaboration* that transforms the *external language* (a program expressed in the syntax of Hazel) to the *internal language* (an intermediate representation more amenable to evaluation), which occurs before evaluation and incorporates the type checking rules. Elaboration and the internal language will be discussed further in Section 2.3.1. The type checking and elaboration algorithms form the static semantics of Hazel.

It is formative to provide an overview of type checking. While the static semantics is not very important to the core work in this thesis, a fundamental understanding is key to

understanding the motivation and bidirectionally-typed action calculus behind Hazel, as well as understanding the formulation of gradual typing described in Section 2.2.4.

The *typing judgment* $\Gamma^- \vdash e^- : \tau^+$ states that, with respect to the typing context Γ, the expression e is well-typed with type τ. The typing context is a set of variable typing judgments $\{x : \tau\}$. A few sample typing judgments are shown below.

$$\frac{}{\Gamma \vdash \underline{n} : \texttt{num}}\ \text{TNum} \qquad \frac{}{\Gamma, x : \tau \vdash x : \tau}\ \text{TVar} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x : \tau_1 . e) : \tau_1 \to \tau_2}\ \text{TAnnArr}$$

$$\frac{\Gamma \vdash e_2 : \tau_1 \qquad \Gamma \vdash e_1 : \tau_1 \to \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_2}\ \text{TAp}$$

There are a few noteworthy items here. The syntax $\Gamma, x : \tau$ indicates the typing context Γ extended with the binding $x : \tau$. Thus, when it is part of the consequent, it means that we are stating the typing judgment with respect to a different typing context $\Gamma' = \Gamma, x : \tau$. The type of a number is always `num`. The type of a variable may only be determined if its type exists in the typing context (which, according to this limited set of rules, may only be extended during a function application). Lambda expressions can only be typed if they are fully-annotated: i.e., if the argument's type is annotated and the body is also assigned a type. This example typing system is very minimal and not practical for larger systems: every λ-abstraction would have to be typed for the entire expression to be well-typed. Consider even the simple example $(\lambda x.x)\ 2$, which cannot be typed according to the simple system above due to the unannotated λ-abstraction.

A type system that allows for fewer type annotations, while remaining reasonably simple to formulate and implement, is *bidirectional typing* [13, 14, 15], or *local type inference*. Bidirectional typing involves two typing judgments: the *synthetic type judgment* $\Gamma^- \vdash e^- \Rightarrow \tau^+$ (pronounced "given typing context Γ, expression e synthesizes type τ), and the *analytic type judgment* $\Gamma^- \vdash e^- \Leftarrow \tau^-$ (pronounced "given typing context Γ, expression e analyzes against type τ). The synthetic type judgment outputs a type (the exact or "narrowest" type of the

expression), whereas the analytic type judgment takes a type as an input and "checks" the expression against that ("wider") type. With these two judgments, we loosen the antecedant judgments when synthesizing a type. We re-express the above type assignment system into a similar bidirectional type system.

$$\frac{}{\Gamma \vdash \underline{n} \Rightarrow \texttt{num}}\ \text{TSynNum} \qquad \frac{}{\Gamma, x : \tau \vdash x \Rightarrow \tau}\ \text{TSynVar}$$

$$\frac{\Gamma, x : \tau_1 \vdash e \Rightarrow \tau_2}{\Gamma \vdash (\lambda x : \tau_1 . e) \Rightarrow \tau_1 \to \tau_2}\ \text{TSynAnnArr} \qquad \frac{\Gamma \vdash e_2 \Rightarrow \tau_1 \qquad \Gamma \vdash e_1 \Leftarrow \tau_1 \to \tau_2}{\Gamma \vdash e_1\ e_2 \Rightarrow \tau_2}\ \text{TSynAp}$$

$$\frac{\Gamma, x : \tau_1 \vdash e \Leftarrow \tau_2}{\Gamma \vdash \lambda x . e \Leftarrow \tau_1 \to \tau_2}\ \text{TAnaArr} \qquad \frac{\Gamma \vdash e \Rightarrow \tau}{\Gamma \vdash e \Leftarrow \tau}\ \text{TAnaSubsume}$$

Now, we may synthesize the type of $(\lambda x.x)\ 2$; the derivation uses all of the rules above. Note the presence of the last rule; *subsumption* states that an expression analyzes against its synthesized type, which should fit the earlier intuition of type synthesis producing the "narrowest" type and type analysis checking against a "wider" type. Subsumption allows us to avoid manually writing type analysis rules for most types.

Algorithmically, bidirectional typing begins by synthesizing the type of the top-level expression; if it successfully synthesizes, then the expression is well-typed. A more complete discussion of bidirectional typing is left to Dunfield [13], who provides an overview of bidirectional typing, or to the formulation of Hazel's bidirectional typing [1]. Hazelnut is at its core a bidirectionally-typed "edit calculus" [1], citing the balance of usability and simplicity of implementation.

The elaboration algorithm is bidirectionally-typed and fairly specific to Hazel and described in Section 3.3.4. It is based off of the cast calculus from the GTLC.

More advanced type inference algorithms such as type unification are used in the highly advanced type systems of languages such as Haskell [16], and are out of scope for this work.

2.1.4 Dynamic semantics

The *dynamic semantics* (alternatively, *evaluation semantics*) of a programming language describes the evaluation process. Evaluation is the algorithmic reduction of an expression to a *value*, an irreducible expression.

The style of rules that are used to define the dynamic semantics of a programming language are called *operational semantics*, because they model the operation of a computer when compiling or evaluating a programming language. There are two major styles of operational semantics.

The first of these styles is *structural operational semantics* as introduced by Plotkin [17] (alternatively, *small-step semantics*). In the small-step semantics, the *evaluation judgment* is $e_1^- \to e_2^+$, where e_1 and e_2 are expressions in the language.

For example, let us describe the dynamic semantics of an addition operation using a small-step semantics. This is described using the following three rules:

$$\frac{e_1 \to e_1'}{e_1 + e_2 \to e_1' + e_2}\ \text{EPlus}_1\text{-Small} \qquad \frac{e_2 \to e_2'}{\underline{n_1} + e_2 \to \underline{n_1} + e_2'}\ \text{EPlus}_2\text{-Small}$$

$$\frac{}{\underline{n_1} + \underline{n_2} \to \underline{n_1 + n_2}}\ \text{EPlus}_3\text{-Small}$$

The algorithm carries itself out as follows: while e_1 is reducible, reduce it using some applicable evaluation rule. Once e_1 becomes a value, the first rule is no longer applicable (as e_1 cannot further reduce) and e_2 reduces until it too is a value. Finally, the third rule is applicable, and reduces the expression down to a single number literal. Note that if either e_1 or e_2 do not reduce down to a number literal, then the expression will not evaluate fully; this kind of failure cannot happen in a strongly-typed language due to typing rules.

The second of these styles is *natural operational semantics* as introduced by Kahn [18] (alternatively, *big-step semantics*). In the big-step semantics, the evaluation judgment is $e^- \Downarrow v^+$, where e is an expression in the language, and v value.

To express the evaluation of addition in the big-step semantics, we need only a single rule. In this case, the antecedants indicate that the subexpressions must be recursively evaluated, but (as noted earlier) this notably doesn't specify the order of evaluation of the antecedants, unlike the small-step notation.

$$\frac{e_1 \Downarrow \underline{n_1} \qquad e_2 \Downarrow \underline{n_2}}{e_1 + e_2 \Downarrow \underline{n_1 + n_2}} \text{ EPlus-B}$$

In the big-step notation, values are distinguished by the judgment $v \Downarrow v$; i.e., values evaluate to themselves. In the small-step notation, there will be no applicable rule to further reduce a value. Following the notation from [1, 2], we can alternatively write this using the equivalent judgment v^- value. In the stereotypical untyped λ-calculus, the only values are λ-abstractions. We can denote this using the axiom:

$$\frac{}{\lambda x.e \ \mathsf{value}} \text{ VLam}$$

In Hazel, we have other base types such as integers, floats, and booleans, which also have axiomatic value judgments. For composite data such as pairs or injections (binary sum type constructors), the expression is a value iff its subexpression(s) are values.

The implementation of an evaluator with a program stepper capability (as is commonly found in debugger tools) is more amenable to implementation using a small-step operational semantics, since it precisely details the sub-reductions when evaluating an expression. The evaluation semantics of Hazelnut Live are originally described using a small-step semantics in [2]. To simplify the rules, the concept of an *evaluation context* $\mathcal{E}$ is used to recurse through subexpressions.

The big-step semantics is often simpler because it involves fewer rules, and is more efficient to implement. As a result, the implementation of evaluation in Hazel more closely follows the big-step semantics, and it is the notation used predominantly throughout this work.

2.2 Introduction to functional programming and the λ-calculus

To understand this work, one must have a satisfactory understanding of Hazel. Understanding Hazel requires some understanding of the *functional programming paradigm*, as Hazel is a stereotypical functional language. One must also have some knowledge of the *gradually-typed λ-calculus* (GTLC) introduced by Siek [19, 20]. This itself is an extension of the simply-typed λ-calculus (STLC), which is an extension of the untyped λ-calculus (ULC), the simplest implementation of Church's λ-calculus. The STLC, the untyped λ-calculus, and Church's λ-calculus are standard textbook material in programming language theory [8], but a brief overview will be provided here.

2.2.1 Introduction to functional programming

Hazel is a pure functional language. Functional programming [21] is a programming paradigm that is highly involved with function application, function composition, and first-class functions. It is a subclass of the declarative programming paradigm, which is concerned with pure[2] expression-based computation. 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 Alonzo Church's λ calculus [8] as its core evaluation and typing semantics, which provides a minimal foundation for computation. The syntax of functional programming languages 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

[2] "Pure" in the sense of a pure function, i.e., without mutable state or side-effects.

languages. This is as opposed to in imperative programming, in which the mutable "memory cell" interpretation of variables and side-effects complicates formalizations. A number of programming languages incorporate both functional and imperative language features, such as Hazel's implementation language OCaml [22]. These languages are classified as multi-paradigm programming languages.

2.2.2 The untyped λ-calculus

Church introduced the *untyped λ-calculus* Λ as an example of a simple universal language of computable functions, and it forms the foundation for the syntax and evaluation semantics of functional programming languages.

The grammar of Λ is very simple, only comprising three forms (excluding parentheses[3]), shown below.

$$
\begin{aligned}
e &::= x && \text{(variable)} \\
&\mid \lambda x.e && (\lambda\text{-abstraction}) \\
&\mid e\ e && \text{(function application)}
\end{aligned}
$$

The static semantics of this syntax are very simple: every expression in Λ is well-fromed if all variables are bound by some binder[4].

The dynamic semantics are similarly simple, shown below using a big-step semantics. λ-abstractions are values (expressions that evaluate to themselves), and application is applied by substituting variables[5].

[3]The imperative programmer with a background in a C-family language be warned: parentheses are not required for function application. Rather, space ($\_$) is an infix operator that represents function application in Λ and many functional languages. It traditionally is left-associative and has the highest precedence of any infix operator. Parentheses around function arguments are only required when it affects the order of operations. An exception to this rule in functional programming is in the LISP family of languages, in which parentheses specify function application rather than operator precedence, and space separates operators and operands, but that is not the interpretation here.

[4]There are no typing rules in the static semantics, because there is only a single type: the recursive arrow type $\tau ::= \tau \to \tau$. Thus, it may be more correct to say that Λ is "uni-typed" as opposed to "untyped," as noted in [8]. Thus no type errors will occur when evaluating a (well-formed) expression in Λ.

[5]The substitution of the function variable during function application is known as β-reduction. Renaming

$$\frac{}{\lambda x.e \Downarrow \lambda x.e}\ \Lambda\text{-ELam} \qquad \frac{e_1 \Downarrow \lambda x.e_1' \qquad [e_2/x]e_1' \Downarrow e}{e_1\ e_2 \Downarrow e}\ \Lambda\text{-EAp}$$

Λ is an example of a Turing-complete language. One of the key characteristics to such a language is the ability to implement recursive algorithms. To implement recursion, a function must be able to refer to itself. Since there is no construct to bind an expression to a variable other than λ-abstractions (i.e., there is no construct such as OCaml's `let rec` expressions or other standalone variable declarations), one must pass a self-reference of a function to itself. For example, let us consider the example of a factorial function in Λ[6].

$$\text{fact'} \equiv \lambda f.\lambda x.\text{if } x = 0 \text{ then } 1 \text{ else } x * f(x-1)$$

To facilitate the recursion, we need the help of an auxiliary operator which converts a recursive function formulated with a self-reference parameter as shown above. The Y-combinator is such an operator. The operation of this operator is made clear by working through the β-reduction of the `fact` function.

$$Y \equiv \lambda f.(\lambda x.f(x\ x))\ (\lambda x.f(x\ x))$$
$$\text{fact} \equiv Y\ \text{fact'}$$

A more thorough discussion of Λ and the Y-combinator is left to standard material on programming language theory, such as [8].

of bound variables (a process known as α-conversion) is used to avoid substituting variables of the same name bound by a different binder.

[6]For sake of illustration, the language is extended with a conditional statement, integers, and simple integer operators.

2.2.3 The simply-typed λ-calculus

While the λ-calculus is Turing complete and sufficient to represent any computation, it is not practical in terms of efficiency or usability if all data is represented with functions[7].

The *simply-typed λ-calculus* (STLC) $\Lambda_{\rightarrow}$ extends Λ with one or more base types b_i, such as integers, booleans, or floating-point numbers. Consider the case of a single base type b. The extended grammar is shown below.

$$
\begin{array}{rr}
\tau ::= \tau \rightarrow \tau & \text{(function type)} \\
\quad \mid b & \text{(base type)} \\
e ::= c & \text{(constant)} \\
\quad \mid x & \text{(variable)} \\
\quad \mid \lambda x : \tau . e & \text{(type-annotated function)} \\
\quad \mid e\ e & \text{(function application)} \\
\quad \mid \text{fix}\, f : \tau . e & \text{(fixpoint)}
\end{array}
$$

The grammar is extended to include constants of the base type. The type of functions parameters must be annotated[8].

We now define what it means for a program in $\Lambda_{\rightarrow}$ to be well-typed. The following typing judgments assign a type to a $\Lambda_{\rightarrow}$ program.

$$
\frac{}{\Gamma \vdash c : b}\ \Lambda_{\rightarrow}\text{-TConst} \qquad \frac{}{\Gamma, x : \tau \vdash x : \tau}\ \Lambda_{\rightarrow}\text{-TVar} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x : \tau_1 . e) : \tau_2}\ \Lambda_{\rightarrow}\text{-TLam}
$$

$$
\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau}\ \Lambda_{\rightarrow}\text{-TAp} \qquad \frac{}{\Gamma \vdash (\text{fix}\, f : \tau . e) : \tau}\ \Lambda_{\rightarrow}\text{-TFix}
$$

The dynamic semantics are not much different than Λ. Additional evaluation rules are defined for constants and fixpoints; evaluation of λ-abstractions and function application

[7] All data may be represented in terms of functions with a notation called the Church encoding. For example, there are standard Church encodings for natural numbers, for boolean values and conditionals, and for pairs (`cons`), which can be used to construct structured data.

[8] This is in the simplest case of type-assignment. With a type inference system such as bidirectional typing as described in Section 2.1.3, some type annotations may be optional.

remains the same.

$$\frac{}{c \Downarrow c}\ \Lambda_{\to}\text{-EConst} \qquad \frac{}{\lambda x : \tau.e \Downarrow \lambda x : \tau.e}\ \Lambda_{\to}\text{-ELam}$$

$$\frac{e_1 \Downarrow \lambda x : \tau.e_1' \qquad [e_2/x]e_1' \Downarrow e}{e_1\ e_2 \Downarrow e}\ \Lambda_{\to}\text{-EAp} \qquad \frac{[\text{fix}\, f : \tau.e/f]e \Downarrow e'}{\text{fix}\, f : \tau.e \Downarrow e'}\ \Lambda_{\to}\text{-EFix}$$

We may characterize type systems by establishing certain desirable properties. One such property is *soundness*. Soundness means that if a program in $\Lambda_{\to}$ type-checks, then it will not fail with a type error at run-time. This property is not necessary to prove for Λ because there is only one type in Λ, the recursive type $\tau ::= \tau \to \tau$.

There is an additional expression form in $\Lambda_{\to}$. This is the *fixpoint form*, $\text{fix}\, f : \tau.e$. The fixpoint is a primitive operator with the same purpose and evaluation behavior as the Y-combinator: it allows for self-reference, and thus general recursion. The reason for the explicit fixpoint operator is that the Y-combinator is ill-typed. Self-reference is inherently poorly-typed and requires a primitive operator, since it involves a function which takes itself as a parameter (leading to an infinitely-recursive arrow type). With the fix operator, we may express the factorial function as shown below[9].

$$\text{fact} \equiv \text{fix}\, f : \text{int} \to \text{int}.\lambda x : \text{int}.\text{if}\ x = 0\ \text{then}\ 1\ \text{else}\ x * f(x - 1)$$

The fixpoint operator is introduced in Plotkin's System PCF [8], and is used to implement recursion in Hazel's evaluator, which uses a substitution-based evaluation.

$\Lambda_{\to}$ is a practical foundation for many functional languages. Standard exercises include extending $\Lambda_{\to}$ with multiple base types (such as integers and booleans), conditional expressions, `let`-expressions, and `case`-expressions. The basic type system can be extended to use type inference algorithms or support more advanced types.

[9] In this example, we assume that the base type $b \equiv \text{int}$, and that conditionals and primitive integer operations extend $\Lambda_{\to}$.

2.2.4 The gradually-typed λ-calculus

We have discussed $\Lambda_{\to}$, which involves a simple *static typing* system, as type checks are part of the static semantics. However, we may extend Λ with an additional base type but without a static semantics. In this case, a well-formed expression may fail at run-time due to type errors – thus, types are checked in the dynamic semantics and this is known as *dynamic typing*. The benefit of static typing is soundness and performance (as run-time type checks are relatively slow). The benefit of dynamic checking is to avoid annotating types[10], and thus more quickly prototype or refactor programs.

A hybrid approach is the *gradually-typed λ-calculus* $\Lambda^{?}_{\to}$, originally proposed by Siek and Taha [19, 20][11]. In $\Lambda^{?}_{\to}$, all type annotations are optional and offer a "pay-as-you-go" benefit. A completely unannotated $\Lambda^{?}_{\to}$ program acts like dynamic typing (Λ extended with base type(s) but no static semantics), with run-time casts and the ability for run-time type failures. A completely annotated $\Lambda^{?}_{\to}$ program is equivalent to a $\Lambda_{\to}$ program. The performance cost of run-time casts and the possibility of run-time type failures only occurs when evaluating expressions with unannotated terms.

The grammar of $\Lambda^{?}_{\to}$ is almost exactly the same as $\Lambda_{\to}$, except that we add a new type $*$, indicating an unspecified type. Now, λ-abstractions annotated with this type may be considered to be unannotated. We define the notation $\lambda x.e \equiv \lambda x : *.e$.

The static semantics of $\Lambda^{?}_{\to}$ is expectedly also similar to $\Lambda_{\to}$. The only rule that differs is the rule for function application. We also write a new rule for subsumption, which states that if $\Gamma \vdash e : \tau$, then e may also be assigned any consistent type.

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \tau_1 \blacktriangleright_{\to} \tau_2 \to \tau_3 \qquad \Gamma \vdash e_2 : \tau_3}{\Gamma \vdash e_1\ e_2 : \tau_3}\ \Lambda^{?}_{\to}\text{-TAp} \qquad \frac{\Gamma \vdash e : \tau \qquad \tau \sim \tau'}{\Gamma \vdash e : \tau'}\ \Lambda^{?}_{\to}\text{-TSub}$$

[10] Note that type inference systems in a statically-typed system also allow for reduced type-annotations, but may still require some annotations when not enough information is given for type inference.

[11] The material presented in this section originates from Siek et al. [19, 20], but the notation conventions follow from Hazelnut Live [2] in order to stay consistent with the rest of this paper. The symbol for the hole type $*$ originates from [20]. The cast calculus notation is an improved notation introduced in [20] and also used in [2].

Two new judgments are introduced here. The first is the *matched arrow judgment* $\tau_1^- \blacktriangleright_\to (\tau_2 \to \tau_3)^+$, which is a notational convenience which allows us to write a single rule for arrow types, which may either be a hole or an arrow type. This judgment is defined by the following rules.

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

The second new judgment is the *type consistency judgment* $\tau_1^- \sim \tau_2^-$. This judgment defines the typing relation of the unknown type to other types: every type is consistent to the hole type. Thus any type will type-check where a hole is expected, and vice versa. This relation is reflexive, symmetric, and non-transitive[12].

$$\frac{}{* \sim \tau}\ \text{TCHoleTyp} \qquad \frac{}{\tau \sim *}\ \text{TCTypHole} \qquad \frac{\tau_1 \sim \tau_1' \qquad \tau_2 \sim \tau_2'}{(\tau_1 \to \tau_2) \sim (\tau_1' \to \tau_2)}\ \text{TCArr}$$

Evaluation of a gradually-typed language introduces a cast calculus, which performs run-time type checking. To do this, we design another language, $\Lambda_\to^{\langle\tau\rangle}$, whose grammar is identical to $\Lambda_\to^?$ except for the introduction of a new expression form indicating a run-time cast, $d\langle\tau \Rightarrow \tau'\rangle$. We also define the notation $d\langle\tau \Rightarrow \tau' \Rightarrow \tau''\rangle \equiv d\langle\tau \Rightarrow \tau'\rangle\langle\tau' \Rightarrow \tau''\rangle$. We refer to $\Lambda_\to^?$ as the *external language* and $\Lambda_\to^{\langle\tau\rangle}$ as the *internal language*. We denote expressions in $\Lambda_\to^?$ with the letter e and expressions in $\Lambda_\to^{\langle\tau\rangle}$ with the letter d. We only define static semantics on $\Lambda_\to^?$ and only define dynamic semantics on $\Lambda_\to^{\langle\tau\rangle}$. The process of converting from the external language to the internal language (i.e., the process of cast-insertion) is called *elaboration*.

Usually, elaboration includes the type-checking operation rather than being a separate

[12]It may seem unintuitive at first that type consistency is a symmetric relationship, because it may seem more like a subtyping relation. However, a major revolution in Siek and Taha's original formulation of $\Lambda_\to^?$ is that the symmetric subtyping relation is more suitable than the subtyping relations that had been explored in earlier works such as Thatte's quasi-static typing [19].

operation. Elaboration fails iff type checking fails. A theorem may be stated that the type assigned by elaboration is the same as the type assignment judgment.

The elaboration process is governed by the judgment $\Gamma^- \vdash e^- \leadsto d^+ : \tau^+$. For most expression types, the expression in the external language elaborates to itself. The only exception is function applications, in which dynamic casts are inserted.

$$\frac{}{\Gamma \vdash c \leadsto c : b}\ \Lambda^?_\to\text{-ElConst} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . e \leadsto (\lambda x : \tau_1 . e) : \tau_1 \to \tau_2}\ \Lambda^?_\to\text{-ElLam}$$

$$\frac{}{\Gamma \vdash \text{fix}\, f : \tau . e \leadsto (\text{fix}\, f : \tau . e) : \tau}\ \Lambda^?_\to\text{-ElFix}$$

$$\frac{\Gamma \vdash e_1 \leadsto d_1 : \tau_1 \qquad \tau_1 \blacktriangleright_\to \tau_3 \to \tau_4 \qquad \Gamma \vdash e_2 \leadsto d_2 : \tau_2 \qquad \tau_2 \sim \tau_3}{\Gamma \vdash e_1\ e_2 \leadsto (d_1 \langle \tau_1 \Rightarrow \tau_3 \to \tau_4 \rangle)\ (d_2 \langle \tau_2 \Rightarrow \tau_5 \rangle) : \tau_4}\ \Lambda^?_\to\text{-ElApp}$$

We may now define a dynamic semantics on $\Lambda^{\langle\tau\rangle}_\to$. The following dynamic semantics is simplified from Siek et. al's formulation for the sake of clarity[13] and is not intended to be a precise description of the evaluation semantics. As before, the dynamic semantics are described using a big-step notation, where the judgment d^- final indicates that d is a value[14].

[13] Siek [20] introduces the idea of *ground types* and the *matched-ground judgment*. Casts can only succeed or fail between ground types. Additionally, Siek et al. describes *blames* and *frames* to encapsulate errors. Ground types are carried over to Hazelnut Live's formulation [2], but blames and frames are not currently implemented in Hazel.

[14] The small-step semantics is perhaps be more clear here, as it more clearly illustrates the isolated effect of the cast operation. Both Siek et al. [19, 20] and Hazelnut Live [2] describe the dynamic semantics using a small-step semantics. Hazelnut Live adds the concept of the *final judgment* to delineate values. However, we use a big-step semantics to remain consistent with the rest of the notation in this work.

$$\frac{}{c \text{ final}}\ \Lambda^{\langle\tau\rangle}_{\to}\text{-VConst} \qquad \frac{}{\lambda x : \tau.d \text{ final}}\ \Lambda^{\langle\tau\rangle}_{\to}\text{-VLam} \qquad \frac{}{e\langle\tau \Rightarrow *\rangle \text{ final}}\ \Lambda^{\langle\tau\rangle}_{\to}\text{-VBoxedVal}$$

$$\frac{d \text{ final}}{d \Downarrow d}\ \Lambda^{\langle\tau\rangle}_{\to}\text{-EVal} \qquad \frac{[\text{fix } f : \tau.d/f]d \Downarrow d'}{\text{fix } f : \tau.d \Downarrow d'}\ \Lambda^{\langle\tau\rangle}_{\to}\text{-EFix}$$

$$\frac{d_1 \Downarrow \lambda x : \tau.d_1' \qquad [d_2/x]d_1 \Downarrow d}{d_1\ d_2 \Downarrow d}\ \Lambda^{\langle\tau\rangle}_{\to}\text{-EAp}$$

$$\frac{d_1 \Downarrow d_1'\langle\tau_1 \to \tau_2 \Rightarrow \tau_1' \to \tau_2'\rangle \qquad (d_1'\ (d_2\langle\tau_1' \Rightarrow \tau_1\rangle))\langle\tau_2 \Rightarrow \tau_2'\rangle \Downarrow d}{d_1\ d_2 \Downarrow d}\ \Lambda^{\langle\tau\rangle}_{\to}\text{-ECastAp}$$

$$\frac{d \Downarrow d'}{d\langle\tau \Rightarrow \tau\rangle \Downarrow d'}\ \Lambda^{\langle\tau\rangle}_{\to}\text{-ECastId} \qquad \frac{d \Downarrow d'}{d\langle\tau \Rightarrow * \Rightarrow \tau\rangle \Downarrow d'}\ \Lambda^{\langle\tau\rangle}_{\to}\text{-ECastSucceed}$$

There is also the possibility of a dynamic cast error.

$$\frac{}{d\langle\tau \Rightarrow * \Rightarrow \tau'\rangle \text{ castfail}}\ \Lambda^{\langle\tau\rangle}_{\to}\text{-ECastFail}$$

Hazel's core calculus heavily borrows from $\Lambda^{?}_{\to}$ and $\Lambda^{\langle\tau\rangle}_{\to}$. The rules for casts will remain unchanged for the dynamic semantics when switching to use the environment mode of evaluation.

2.3 Implementations of programming languages

In order to run programs in a programming language on a computer, we must have an *implementation* of 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.

2.3.1 Compiler vs. interpreter implementations

There are two general classes of programming language implementations: *interpreters* and *compilers* [9]. 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) [9]. 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.

The term *elaboration* [23] may be used to describe the process of transforming the *external language* (a well-formed, textual program) into the *internal language*. The interior language may include additional information not present in the external language, such as types generated by type inference or bidirectional typing.

The distinction between compiled and interpreted languages may not be very clear: some implementations feature just-in-time (JIT) compilation that allow "on-the-fly" compilation (e.g., common implementations of the JVM and CLR [24]), 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 [25], which is outside the scope of this project.

2.3.2 The substitution and environment models of evaluation

Evaluation in Hazel was originally performed 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 eagerly substituted with their value when bound. 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. Variables are lazily looked up in their environment. 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 and because untraversed (unevaluated) branches do not require substituting. Lastly, the runtime does not need to carry an expression-level IR of the language, 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 use of substitution or environments for evaluation is orthogonal to strict (applicative-order) or lazy (normal-order) evaluation [26]. The use of substitution or environments refers to the eager or lazy substitution of variables. Strict and lazy evaluation refer to when the expression bound to a variable is evaluated.

2.4 Approaches to programming interfaces

The most traditional programming interface is the text source file. In this case, we describe two innovations on plaintext files that are relevant to Hazel's design and use cases.

2.4.1 Structure editors

Structure editors form a class of programming language editors that allows one to directly interface with the *abstract syntax tree* of a programming language via a restricted set of *edit actions*, rather than via the manipulation of unstructured text. The immediate benefit of this is the elimination of the entire class of errors related to syntax.

A major use case for structural editing is for the purpose of programming education.

By eliminating syntactic errors, the student may shift their attention more towards semantic issues in their code. For example, Carnegie Mellon University developed a series of structural editors (GNOME, MacGnome, and ACSE) targeted at programming education [27]. Scratch is a graphical structural editor targeted at younger students (aged 8-16) developed at MIT [4]. Programming education is one of the main proposed use cases for Hazel, such as its use in Hazel Tutor [28].

Structure editors are not limited to programming education. mebbdr is a structure editor for embedded programming [6], and Lamdu is a structure editor for a Haskell-like functional programming language [5].

One of the major drawbacks of structural editing is the decrease in usability by restricting the set of edit actions. The degree to which an editor "resists local changes" is a property known as *viscosity* [29]. Structural and visual editing is expectedly more viscous than unstructured plaintext editing. Reducing the viscosity of structural editing is not a goal of Hazel, but a related project at the University of Michigan, Tylr [30], tackles the problem of editing viscosity and may make its way into future versions of Hazel.

Omar et al. [1] describes several other structure editors and their relation to Hazel, and in particular other structure editors which also attempt to maintain well-typedness or operate on formal definitions of an underlying language.

2.4.2 Live programming environments and computational notebooks

Burckhardt et al. describes live programming environments as providing continuous feedback that narrows the "temporal and perceptive gap between program development and code execution" [31]. A common example of a live programming environment are read-evaluate-print loops (REPLs), which allow line-by-line evaluation of expressions. Computational notebooks form an example of live programming environments.

Computational notebooks, such as in IPython/Jupyter Notebook [32] or MATLAB, is

another trend in programming languages that has been popular in scientific applications. They provide much feedback about program's dynamic state, especially interactively or graphically. Notebook-style editing allows one to intersperse editing and evaluation of a program. Programs may be run in sections (potentially out of order), maintaining state between sections evaluations – this is typically for efficiency reasons. There is a large design space in current computational notebooks, with many possible variations in code evaluation, editing semantics, and displaying notebook outputs [33].

Hazel may be considered a live editor as it attempts to eliminate the feedback gap, by providing static and dynamic feedback throughout the lifetime of then program. The fill-and-resume functionality described in [2] and implemented in this work provide a novel possible implementation of notebook-like partial evaluation.

Chapter 3

An overview of the Hazel programming environment

Hazel is the reference implementation for the Hazelnut bidirectionally-typed action semantics and the Hazelnut Live dynamic semantics. It is intended to serve as a proof-of-concept of the semantics with static holes that attempt to mitigate the gap problem; however, the implementation is becoming increasingly practical with recent additions to the language. The reference implementation is an interpreter written in OCaml and transpiled to Javascript using the `js_of_ocaml` (JSOO) library [34] so that it may be run client-side in the browser. A screenshot of the reference implementation is shown in Figure 1.1 [3]. The source code may be found on GitHub [7]. Hazel may be characterized as a purely functional, statically-typed, bidirectionally-typed, strict-order evaluation, structured editor programming language.

3.1 Motivation for Hazel

3.1.1 The gap problem

Programming editor environments aim to provide feedback to a programmer in the form of editor services such as syntax highlighting or warnings using the LSP. Live programming

environments aim to provide continuous static (static type error) and dynamic (run-time type error) feedback in real-time, allowing for rapid prototyping. However, over the course of the lifetime of a program, the program may enter many edit states when it is *meaningless* (ill-formed or ill-typed).

Editor services can only assign static and dynamic meaning to programs that are statically well-typed and free of dynamic type errors. Some may deploy reduced ad hoc algorithms of meaningless edit states. This means that over the course of editing, the programmer experiences temporal gaps between moments of complete editor services. This is known as the *gap problem* [31, 2].

3.1.2 An intuitive introduction to typed expression holes

Hazelnut and Hazelnut Live address the gap problem by defining a static and dynamic semantics, respectively, for a small functional programming language extended with typed holes. It is built on top of a *structure editor*, which ensures that a program is always well-formed (syntactically correct) by disallowing invalid edit actions. The Hazelnut action semantics for typed holes ensures that a well-formed program is always well-typed. The Hazelnut Live dynamic semantics defines an encapsulated behavior for type errors, such that evaluation continues "around" and captures information about type errors in order to provide dynamic feedback to the programmer.

The Hazelnut Live paper provides the following intuitive understanding of holes.

> Empty holes stand for missing expressions or types, and non-empty holes operate as "membranes" around static type inconsistencies (i.e. they internalize the "red underline" that editors commonly display under a type inconsistency).

We have already acknowledged the existence of type holes in dynamically-typed languages and in the $\Lambda^{\langle\tau\rangle}_{\rightarrow}$, in which type holes are represented by the type $*$. This allows unannotated expressions to statically type-check, with the possibility of running into a dynamic type error

at runtime.

Some languages also have the concept of expression holes, which allow a program to be well-typed with missing expressions. In Haskell, for example, the special error value `undefined` always type-checks but will immediately crash the program if it is encountered during evaluation. Haskell also provides the syntax `_u` for expression holes [**TODO: need reference(s): cite this**], which provides static type information but will not successfully compile. The mechanism to insert expressions holes may be either automatic or manual [**TODO: need reference(s): cite this**]. However, no such example of expression holes have a well-defined dynamic semantics that allows continuation past the hole with useful feedback [**TODO: need reference(s): cite this – perhaps cite hazelnut 2019 paper**].

In summary, Hazel provides empty type and expression holes, which represent dynamic typing and missing expressions. Nonempty holes are also provided to encapsulate error conditions and provide a well-defined dynamic semantics while providing useful feedback to the user. The dynamic semantics is carefully defined to stop when such indeterminate expressions are encountered, but continue elsewhere ("around" holes or failed casts) if possible.

3.1.3 The Hazel interface

In Figure 3.1 the web interface for the Hazel live environment is shown. The left panel marked (1) is a informational panel showing the list of keyboard shortcuts to perform actions. Since Hazel is a structured editor, simply typing the program as plaintext will not work; one must use the appropriate shortcuts the construct and edit the program. (2) is the code view. Below the code, a gray box indicates the result of evaluating the expression. The program result updates in real time with every edit action, assuming that evaluation is turned on. (3) is the context inspector, which shows information about a hole if a hole is selected. It shows the hole environment and typing context, followed by the path to the hole and the number of hole instances. In this case, the third hole in the result is selected, in which x has value 3. Lastly, (4) shows a history of the edit actions. Hovering or clicking on a past edit state

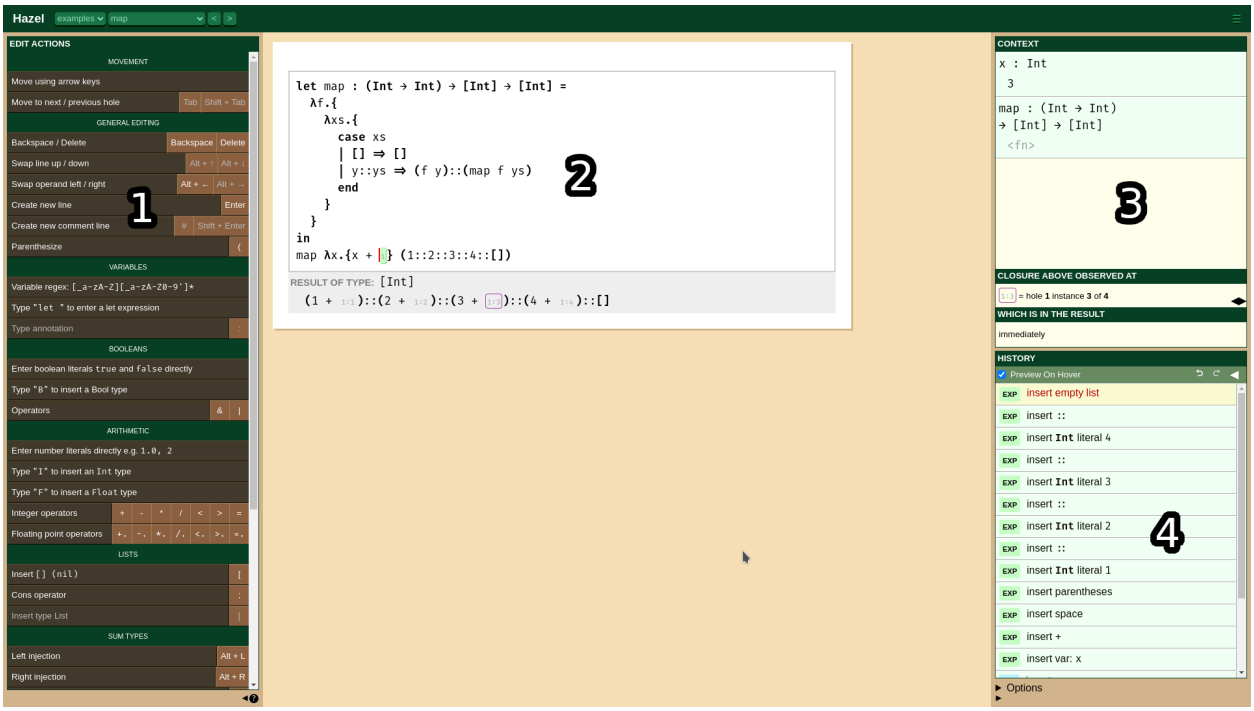


Figure 3.1: The Hazel interface, annotated

will revert the program to that edit state.

3.1.4 Implications of Hazel

The main proposed use case of Hazel is its use in programming education, particularly for teaching functional programming, as it provides much useful feedback to the programmer for error conditions, allowing them to focus instead on semantic errors in their algorithm. This is being explored with the Hazel Tutor project [28].

Another research direction is in its use as a structural and graphical editor. For example, live GUIs [35] are being explored to enhance the editing experience by providing live, compositional, graphical interfaces, in addition to the benefits that Hazel's core calculi provide.

The result of a Hazel evaluation may contain holes, and thus not be fully evaluated. The Hazelnut Live paper [2] suggests the idea of hole-filling: since each hole in the result contains its lexical environment, we may "resume" evaluation without restarting evaluation from the

beginning if a hole is filled – this property is similar to that of computational notebooks. The problem with notebook execution is that it is stateful and running operations out-of-order may cause irreversible state changes that cause irreproducible results. On the other hand, resuming an evalution with fill-and-resume will produce the same result as if the program was run ordinarily from start to finish[1] while avoiding re-evaluation of previous sections.

3.2 Introduction to OCaml and Reason syntax

Previously, we have been introducing concepts using a pseudo-mathematical notation. When describing Hazel and its implementation, it may be useful to use sample code or pseudocode from the implementation to describe various aspects of Hazel.

Hazel is implemented in Reason (alternatively, ReasonML), which is a dialect of OCaml that offers a JavaScript-like syntax. Except for code samples in Appendix D, the notation used throughout this report will be limited to referring to function names and types. Module names are denoted `PascalCase`, whereas function and type names are `snake_case`. Conventionally, OCaml modules that export a type export a single type called `t`. As an example, `DHExp.t` refers to the primarily-relevant type from the `DHExp` module, the type that represents internal expressions d. On the other hand, `Evaluator.evaluate` refers to the `evaluate` function in the `Evaluator` module. All functions and types will be prefixed with their module names for clarity.

3.3 Hazel semantics

Hazel is rigorously defined using a bidirectional semantics. A high-level overview of the foundational papers on Hazelnut (syntax and static semantics) and Hazelnut Live (elaboration and dynamic semantics) is provided here, but a thorough explanation is deferred to the original papers.

[1]This is a property known as *commutativity* and described in [2].

3.3.1 Hazelnut syntax

The grammar of Hazel's external language is defined as follows.

$$\tau ::= \tau \to \tau \mid b \mid ⦇⦈$$
$$e ::= c \mid x \mid \lambda x : \tau . e \mid e\ e \mid e : \tau \mid ⦇⦈ \mid ⦇e⦈$$

This is very similar to $\Lambda^{?}_{\to}$. The $*$ type is rewritten as ⦇⦈ and pronounced the "hole type." An expression form for type ascription is added. Most notably, there is the addition of empty and non-empty expression holes, which are denoted ⦇⦈ and ⦇e⦈, respectively.

3.3.2 Hazelnut action and typing semantics

Hazelnut [1] defines a bidirectional typing judgment for the external language. **[TODO: reproduce this in an appendix]** The judgments are very similar to $\Lambda^{?}_{\to}$. Unsurprisingly, hole expressions synthesize the hole type, and they analyze against any type. Note that in the case of a non-empty hole, the encapsulated expression must still synthesize a type, i.e., they are well-typed.

Hazelnut defines an action semantics for the structural editor, which describes the behavior of editing and maneuvering around a program. A program's edit state comprises an external expression with a superimposed cursor. There are four main actions carried out by the user: `move`, `construct`, and `delete`. These actions are described by bidirectionally-typed action judgments that transform a (well-typed) edit state to another (well-typed) edit state. There are a number of metatheorems that enforce desirable properties of action semantics in a structural editor, such as *sensibility* (the result of an action on a well-typed expression is a well-typed expression), *movement erase invariance* (movement actions should not change the external expression, but only the position of the cursor), *reachability* (the cursor should be able to move to any valid location to any other valid location), *constructability* (every valid edit state should be constructable from the initial edit state), *action determinism* (every

sequence of edit actions should have only one valid output state), etc. These metatheorems are proved using the Agda theorem proving assistant [11].

[**TODO: talk about missing forms from the Hazelnut formulation: let expressions, case expressions, binary sum types, lists, other hole types, fixpoint**]

3.3.3 Hazelnut Live elaboration judgment

Elaboration is the process of converting an expression from the external language to the internal language. Notably, both the external and internal languages share the same type system. The internal language and the elaboration process is very similar to the cast calculus $\Lambda^{\langle\tau\rangle}_{\to}$and the elaboration process from $\Lambda^{?}_{\to}$.

The elaboration algorithm is also bidirectionally-typed, and thus involves two mutually-recursive judgments: a *synthetic elaboration judgment* $\Gamma^- \vdash e^- \Rightarrow \tau^+ \rightsquigarrow d^+ \dashv \Delta^+$, and an *analytic elaboration judgment* $\Gamma^- \vdash e^- \Leftarrow \tau^- \rightsquigarrow d^+ : \tau'^+ \dashv \Delta^+$.[**TODO: reproduce elaboration judgments in appendix**]

Δ is the *hole context*, used to store the typing context and actual type of each hole. Each hole (whether in synthetic or analytic position) is recorded in the hole context, and is given the identity mapping as its original environment[2].

The elaboration judgment will produce as output a type for the internal expression, which may be different from the type of the external expression. In particular, elaborated holes will produce different types depending on whether they are in synthetic or analytic position.

3.3.4 Hazelnut Live final judgment and dynamic semantics

Hazelnut Live introduces a new d final judgment for the internal language, used to indicate an irreducible expression. This subsumes the set of fully-evaluated expressions in $\Lambda^{\langle\tau\rangle}_{\to}$, which include plain values or *boxed values*, values which are casted "out" of their original type but

[2]This is amended in this work, in which holes will not initially be given an environment because the environment is not substitution-based.

not yet casted "into" the destination type. However, expressions containing holes also cannot further evaluate and comprise the second class of final expressions: *indeterminate* values.

[**TODO: reproduce final judgment**]

Hazelnut Live defines a small-step semantics for its internal language very similar to that of $\Lambda^{\langle\tau\rangle}_{\to}$. To avoid the rapid proliferation of rules due to the small-step semantics, a notational convenience called the *evaluation context* $\mathcal{E}$, which recursively evaluates subexpressions. The rules are modified to accomodate indeterminate expressions.

[**TODO: reproduce small-step semantics in an appendix**]

3.3.5 Hole instance numbering

Hazelnut Live introduces *hole instances* with some motivation, but with no details of its implementation. In Chapter 5, we will motivate hole instances in greater detail, describe the current implementation, and reformulate the problem of hole instance tracking to accomodate environments and memoization.

Chapter 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.

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

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

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

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

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

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

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

$$\frac{}{\sigma \vdash (\!|)^u_\varnothing \Downarrow (\!|)^u_\sigma} \text{ EvalB-EHole} \qquad \frac{\sigma \vdash d \Downarrow d'}{\sigma \vdash (\!|d|\!)^u_\varnothing \Downarrow (\!|d'|\!)^u_\sigma} \text{ EvalB-NEHole}$$

Figure 4.1: Big-step semantics for the environment model of evaluation

4.1.1 Evaluation rules

Omar et al. [2] 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 4.1.

The evaluation model threads a run-time environment σ[1] 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.

[1]The symbol σ was chosen to represent the environment as it was used to represent hole environments in [2]. The relationship between these two environments will be discussed in Section 4.1.2.

Evaluation judgments are shown for a subset of the Hazel language, similar to the internal language described in [2]. 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 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.

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.

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 $\mathrm{id}(\Gamma)$, 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 infinitely recursive substitution will occur.

Recursion with the environment model also requires self-reference. 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 8.2.

Both pure methods were implemented; their tradeoffs are described below. The final

implementation (and the rules shown in Figure 4.1) 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[2]. 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 expression is evaluated again, with the self-reference[3]. 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[4]. In the more general case of mutual recursion, this would likely make implementation very tricky, and it would be more practical to use impure

[2]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.

[3]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 4.1. It also does not take many evaluation steps and is thus not an expensive operation.

[4]§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.

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

Figure 4.2: Evaluation rule for simple recursion using self-recursive data structures

`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 4.2. 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, 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 post-processed result to be slightly different[5]. The second problem is solved by the fast equality checker for memoized environments described in Section 5.5, which is useful even for

[5]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.

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 Chapter 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 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 expres-

sions 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 post-processing 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[6]. This is not captured in the original description of Hazelnut Live [2] 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 [2] 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 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.

[6]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, ...)
  (* λ expressions and 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, ...)
  (* λ expressions and closures *)
  | Lam(x, τ, d)
  (* Generalized closure *)
  | Closure(σ, d)
  (* ... *) ;
```

(b) Generalized closures

Figure 4.3: Comparison of internal expression datatype definitions (in module DHExp) for non-generalized and generalized closures.

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 4.3. In Figure 4.3a 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 4.3b 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 [2]:

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

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

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

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

This implementation of closures is an improvement in three 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. Lastly, generalized closures play an important role in the fill-and-resume operation, in which (unevaluated) closures can contain arbitrary subexpressions and allow "resuming" evaluation in the stored 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 evaluation 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[7]. 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.

4.2.4 Pattern matching for closures

Pattern matching is not the primary focus of this work, but it warrants a brief discussion here. Since we introduce a new `DHExp.t` variant, we also need to implement all the methods that switch on a `DHExp.t`, such as pattern matching.

Pattern matching is implemented in the function `Evaluator.matches`, which has type `(DHPat.t, DHExp.t) => Evaluator.match_result`. If pattern matching succeeds, then an environment containing the matched binding(s) will be returned. Otherwise, pattern matching may be indeterminate (if either the pattern or bound expression is indeterminate), or it may fail. Note that the expression passed to `Evaluator.matches` is already evaluated.

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

Closures are a unique variant of `DHExp.t` in that they are a container type, whose contained expression determines its behavior during pattern matching. An evaluated closure[8] may only contain one of four types of expressions: λ-abstractions, holes, unmatched `let` expressions, or unmatched `case` expressions. The former is a boxed value and should match against variables only, and otherwise fail. The latter three are indeterminate and should match against variables and return an indeterminate match otherwise.

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.

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

[8] An evaluated closure is one for which the `re_eval` flag introduced in Section 6.2.2 is false. Thus far, all closures we have encountered are evaluated.

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[9] 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 closure environment[10]. 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

[9]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.

[10]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.

$\Uparrow_{[],2}$ 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`[11] 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[12].

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.

Theorem 4.4.1 (Use of id_σ as an identifier)**.** *The mapping $i_\sigma : \sigma \mapsto id_\sigma$ that maps an*

[11] 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`.

[12] 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 8.2.

$\boxed{\sigma \vdash d \Uparrow_{[]} d'}$ d postprocesses (λ-conversion) to d' outside the evaluation boundary

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

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

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

$$\frac{}{\sigma \vdash (\!|\,|\!)^u_{\varnothing} \Uparrow_{[]} (\!|\,|\!)^u_{\sigma}} \text{ PPO}_{[]}\text{-EHole} \qquad \frac{\sigma \vdash d \Uparrow_{[]} d'}{\sigma \vdash (\!|d|\!)^u_{\varnothing} \Uparrow_{[]} (\!|d'|\!)^u_{\sigma}} \text{ PPO}_{[]}\text{-NEHole}$$

$\boxed{d \Uparrow_{[]} d'}$ d postprocesses (λ-conversion) to d' within the evaluation boundary

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

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

$$\frac{\sigma \vdash d \Uparrow_{[]} d'}{[\sigma]\lambda x.d \Uparrow_{[]} \lambda x.d'} \text{ PPI}_{[]}\text{-Closure} \qquad \frac{d_1 \Uparrow_{[]} d_1' \qquad d_2 \Uparrow_{[]} d_2'}{d_1(d_2) \Uparrow_{[]} d_1'(d_2')} \text{ PPI}_{[]}\text{-Ap}$$

$$\frac{d_1 \Uparrow_{[]} d_1' \qquad d_2 \Uparrow_{[]} d_2'}{d_1 + d_2 \Uparrow_{[]} d_1' + d_2'} \text{ PPI}_{[]}\text{-Op} \qquad \frac{\sigma' = \{(x \leftarrow d') : (x \leftarrow d) \in \sigma, d \Uparrow_{[]} d'\}}{(\!|\,|\!)^u_{\sigma} \Uparrow_{[]} (\!|\,|\!)^u_{\sigma'}} \text{ PPI}_{[]}\text{-EHole}$$

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

TODO: closure needs to go recursive

Figure 4.4: Big-step semantics for λ-conversion post-processing

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 $\{\sigma_i, \sigma_{i+1}, \ldots, \sigma_{j-1}, \sigma_j\}$ (without loss of generality, assume σ_i is an earlier environment than σ_j). By construction, each element of the set $\{i_\sigma(\sigma_i), i_\sigma(\sigma_{i+1}), \ldots, i_\sigma(\sigma_j)\}$ is unique. Thus $i_\sigma(\sigma_1) \neq i_\sigma(\sigma_2)$. □

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 (postprocessed) 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 Chapter 5.

4.5.1 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.

The only major change to the data structures is the switch from using linked lists (`VarMap.t`) as the backing store for environments to using a binary search tree representation (`VarBstMap.t`). This improves performance of operations on large environments.

Hashtables were not used at all in the implementation; their effect on performance is unknown and is reserved for future work. While they allow for amortized $O(1)$ operations, they are stateful and thus difficult to copy, and do not allow for the structural sharing memory optimization. Since immutable data structures are efficiently each time they are modified, the costs of introducing hashtables will likely outweigh the costs.

4.5.2 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 memoized environments, and a very similar data structure to `HoleClosureInfo.t` or `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 5.3, 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 postpro-

cessing 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[13].

4.5.3 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[14]. 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[15]. The decision is somewhat arbitrary but may have small effects on the evaluation performance and elegance of implementation.

[13] We may offer the following intuition for this claim. The issue of exponential hole instance exponential blowup described in Section 5.3 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.

[14] 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`.

[15] 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 several calls to evaluate.

Chapter 5

Memoizing hole instance numbering using environments

5.1 Rationale behind hole instances and unique hole closures

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

$$[a \leftarrow [\varnothing]\,(\!|\,|\!)^1, x \leftarrow 3]\,(\!|\,|\!)^2 + [a \leftarrow [\varnothing]\,(\!|\,|\!)^1, x \leftarrow 4]\,(\!|\,|\!)^2$$

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

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

Listing 1: Illustration of hole instances

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 $⦇⦈^2$, the assignment of two separate holes to $⦇⦈^1$ 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 will be called a **unique hole closure**, or simply **hole closure**[1].

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

[1] "Hole closure" also is used to describe the generalized closure around hole expressions as described in Chapter 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.

```
let f = λ x . { ()¹ } in
f 2 + f 2
```

Listing 2: Illustration of physical equality for environment memoization

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).

5.2 The existing hole instance numbering algorithm

Hole numbering is a process that follows evaluation and operates on the evaluation result[2]. It assigns a hole instance number to each hole. The hole numbering algorithm is not discussed in the Hazelnut Live description. We also will not describe it as a set of judgments, for brevity. It is a breadth-first search of the result, recursing through holes. When a hole is encountered, it is assigned a unique hole instance number and added to a data structure `HoleInstanceInfo.t` that keeps track of all hole instances. Each hole instance's hole number, hole instance number, hole closure environment, and path[3] is stored in this data structure. The `HoleInstanceInfo.t` is in turn stored in the `Result.t` that stores all of the information about an evaluated program. The primary use of `HoleInstanceInfo.t` is for the context inspector. With this data structure, users may easily iterate all instances of a selected hole, examine the hole path of a selected hole, examine the environment of a selected hole, or navigate to another hole instance.

[2]The function in the existing codebase that performs hole renumbering is `Program.renumber`. We may refer to it throughout this text as "hole numbering," "hole renumbering," or "hole tracking."

[3]The path of a hole is the recursive list of hole parents that must be traversed in order to reach a hole. In other words, this is the path to a hole if we envision the result expression as a tree, in which each hole is a node that fathers all of its variable binding expressions.

5.3 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 3[4]. As we increase the number of consecutive `let` expressions, we get an exponential slowdown that makes evaluation impractical for $n > 10$. The results of running this program for several values of n is shown in tabular form in Table 7.2 and graphically at Figure 7.4a.

For now, let us consider the case when $n = 3$. When evaluating with environments[5], the result is shown in Figure 5.1.

The program slowdown happens in the hole numbering process. Recall from Section 5.2 that the hole numbering process is a simple tree traversal algorithm. Thus, each time a hole (with the same environment) is encountered, it and all of its descendant holes will be given more hole instance numbers. This leads to the hole numbering shown in Figure 5.2. We see that there are four instances of hole 1, two instances of hole 2, and one instance of hole 3. In sum, we see that there are eight total hole instances. It should be intuitive that the number of holes increases by powers of two; the total number of holes (including the instance of hole 4 in this case) will be 2^N.

Clearly this is undesirable from an efficiency perspective. It is also undesirable from the perspective that there is only one instantiation of each of the holes. While there are multiple paths to each node, we would like to change the representation to match that of the unique hole closures or hole instantiations as described in Section 5.1.

5.3.1 Hole instance path versus hole closure parents

Visually, we would like to change the hole tracking to use a representation more similar to Figure 5.1 rather than that of Figure 5.2. In the old representation, each hole instance is

[4]This was first brought to attention by a GitHub issue at `https://github.com/hazelgrove/hazel/issues/536`.

[5]When evaluating using the substitution model, evaluation also slows down exponentially, because the variables are eagerly substituted into the hole environments. We do not have a performance issue with evaluation with environments because of lazy variable lookups.

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

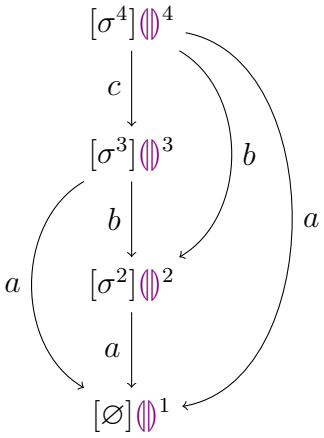


Figure 5.1: Structure of the result of the program in Listing 3

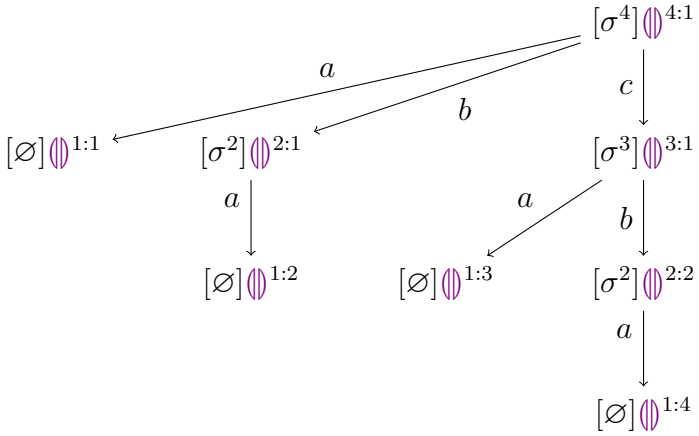


Figure 5.2: Numbered hole instances in the result of Listing 3

uniquely identified by a hole number and hole path.

In the new representation, hole instantiations are uniquely identified by hole number and environment, and are not uniquely identified by a path anymore. Thus, for each hole closure we instead keep track of a list of its parent holes.

We note that these two representations of a graph are equivalent[6], assuming that nodes sharing an environment are considered to be physically equal. The first describes the path to each node, while the latter is a well-known adjacency list. Either representation of a graph can be used to construct the other, but the latter is much more efficient in the case of a dense graph.

Changing the structure from using hole paths to hole parents forces a minor change to the Hazel UI. When a user selects a hole, rather than showing the path to the hole, the list of parents to the hole are shown instead.

5.4 Algorithmic concerns and a two-stage approach

To efficiently build the new hole-tracking data structure, we expect to have a fast lookup of hole numbers and environment identifiers. On the other hand, we want the interface of this data structure to be similar to the interface of `HoleInstanceInfo.t`: the user should be able to look up environments by hole number and hole closure number.

To efficiently handle both of these interfaces, we require two different data structures. The first is an auxiliary data structure `HoleClosureInfo_.t` that is a map $H : (u, \sigma) \mapsto (i, p)$ (where p indicates the list of hole closure parents). The second is the data structure that will be used for the context inspector and hole closure lookups, `HoleClosureInfo.t`, that is a map $H : (u, i) \mapsto (\sigma, p)$. The maps are implemented as hashmaps for efficient lookups and updates[7]. The first stage of this algorithm is to build the `HoleClosureInfo_.t`; the second

[6]This structure is more specifically a join-semilattice.

[7]Note that this is one of the few places where a hashtable implementation is appropriate in the context of this project, since we do not copy these data structures. However, there will likely not be a major performance benefit; the main benefit lies in memoizing environments.

stage is to convert it to a `HoleClosureInfo.t`[8].

For convenience, we do not use two different symbols for these two data structures; the difference is purely an implementation detail regarding the construction of the data structure. The conversion from `HoleClosureInfo_.t` to `HoleClosureInfo.t` is trivial and will not be described here in detail. It simply involves looping over the unique hole closures and changing the mapping to be indexed by hole number and hole closure number.

5.4.1 The hole numbering algorithm

The hole numbering algorithm is shown in Figure 5.3. Constants and variables are left unchanged by the hole numbering algorithm. For ordinary expressions with subexpressions, the algorithm recurses through subexpressions.

For hole expressions that have not been encountered before, the hole number and environment do not exist in H. A hole closure number $i = \text{hid}(H, u)$ is generated to be unique out of hole closures for the hole u. We recursively postprocess the environment. Then the hole closure is inserted into H, along with the postprocessed environment.

For hole expressions that have been encountered before, the hole number and environment do exist in H. We can use this to look up the hole closure number i, postprocessed environment σ', and list of parents $\{p_i\}$ for this hole closure. We update the list of parents to include the current parent p, and return the hole numbered with i. Note that the environment does not need to be re-postprocessed.

5.4.2 Hole closure numbering order

The order of the numbers assigned to hole closures is not specified in the algorithm shown in Figure 5.3, but it is a consideration in the implementation. In the existing implementation, the evaluation result is traversed in a breadth-first search (BFS) order. On the other hand, our implementation is more simply implemented using a depth-first search (DFS) order. This

[8]It is a good time to bring up whether algorithmic efficiency is a matter of concern at all

$\boxed{H, p \vdash d \Uparrow_i d' \dashv H'}$ d gets renumbered to d'

$$\frac{}{H, p \vdash c \Uparrow_i c \dashv H} \text{PP}_i\text{Const} \qquad \frac{}{H, p \vdash x \Uparrow_i x \dashv H} \text{PP}_i\text{Var}$$

$$\frac{H, p \vdash d \Uparrow_i d' \dashv H'}{H, p \vdash \lambda x : \tau . d \Uparrow_i \lambda x : \tau . d' \dashv H'} \text{PP}_i\text{Lam}$$

$$\frac{H, p \vdash d_1 \Uparrow_i d_1' \dashv H' \qquad H', p \vdash d_2 \Uparrow_i d_2' \dashv H''}{H, p \vdash d_1\ d_2 \Uparrow_i d_1'\ d_2' \dashv H'} \text{PP}_i\text{Ap}$$

$$\frac{H, p \vdash d \Uparrow_i d' \dashv H'}{H, p \vdash d : \tau \Uparrow_i d' : \tau \dashv H'} \text{PP}_i\text{Asc}$$

$$\frac{\begin{array}{c}(u, e) \notin H \qquad i = \text{hid}(H, u) \\ H, (u, i) \vdash \sigma \Uparrow_i \sigma' \dashv H' \qquad H'' = H, (u, e) \leftarrow (i, \{p\}, \sigma')\end{array}}{H, p \vdash [\sigma^e](\!|\!)^u \Uparrow_i [\sigma'^e](\!|\!)^{u:i} \dashv H''} \text{PP}_i\text{EHoleNew}$$

$$\frac{H = H', (u, e) \leftarrow (i, \{p_i\}, \sigma'^e) \qquad H'' = H, (u, e) \leftarrow (i, \{p_i\} \cup \{p\}, \sigma'^e)}{H, p \vdash [\sigma^e](\!|\!)^u \Uparrow_i [\sigma'^e](\!|\!)^{u:i} \dashv H''} \text{PP}_i\text{EHoleFound}$$

$$\frac{\begin{array}{c}(u, e) \notin H \qquad i = \text{hid}(H, u) \qquad H, (u, e), \vdash \sigma \Uparrow_i \sigma' \dashv H' \\ H'' = H, (u, e) \leftarrow (i, \{p\}, \sigma') \qquad H'', p \vdash d \Uparrow_i d' \dashv H'''\end{array}}{H, p \vdash [\sigma^e](\!|d|\!)^u \Uparrow_i [\sigma'^e](\!|d'|\!)^{u:i} \dashv H'''} \text{PP}_i\text{NEHoleNew}$$

$$\frac{\begin{array}{c}H = H', (u, e) \leftarrow (i, \{p_i\}, \sigma'^e) \\ H'' = H, (u, e) \leftarrow (i, \{p_i\} \cup \{p\}, \sigma'^e) \qquad H'', p \vdash d \Uparrow_i d' \dashv H'''\end{array}}{H, p \vdash [\sigma^e](\!|d|\!)^u \Uparrow_i [\sigma'^e](\!|d'|\!)^{u:i} \dashv H'''} \text{PP}_i\text{NEHoleFound}$$

$\boxed{H, p \vdash \sigma \Uparrow_i \sigma' \dashv H'}$ σ gets renumbered to σ'

$$\frac{}{H, p \vdash \varnothing \Uparrow_i \varnothing \dashv \varnothing} \text{PP}_i\text{EnvTriv}$$

$$\frac{H, p \vdash \sigma \Uparrow_i \sigma' \dashv H' \qquad H', p \vdash d \Uparrow_i d' \dashv H''}{H, p \vdash \sigma, x \leftarrow d \Uparrow_i \sigma', x \leftarrow d' \dashv H''} \text{PP}_i\text{Env}$$

Figure 5.3: Hole closure numbering postprocessing semantics

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

$$\frac{d \Uparrow_{[]} d' \qquad \varnothing, \varnothing \vdash d' \Uparrow_i d'' \dashv H}{d \Uparrow d'' \dashv H} \text{ PP-Result}$$

Figure 5.4: Overall postprocessing judgment

changes the order that hole closures are encountered and numbered. While the hole closure number is not specified explicitly ordered value, certain orderings may be more intuitive to the user. A choice of explicit hole numbering order is left to future work.

5.4.3 Unification with λ-conversion post-processing

The overall postprocessing operation is shown in Figure 5.4. The postprocessing is the composition of the λ-conversion postprocessing and the hole numbering postprocessing.

We can combine the two postprocessing steps into a single pass. This essentially involves incorporating the hole numbering rules into the closure and hole rules for λ-conversion post-processing. This is how the two postprocessing steps are implemented. For brevity and readability, a combined set of judgments is not described in this paper.

The hole numbering step limits the use of memoization in the postprocessing process. Environments cannot be entirely memoized because a hole must be re-postprocessed in order to add a new hole parent. In other words, if we memoize environments, then some hole parents would be missing because the holes in the environments would not be re-postprocessed. Thus we implement a partial memoization, where re-evaluation of an environment does not necessarily recurse into hole environments. This is not ideal, and a more elegant implementation of the postprocessing step is left for future work.

5.5 Fast structural equality checking

After the hole instance numbering is solved, there is an additional performance issue that is related to a recursive traversal of the evaluation result. After evaluation and hole numbering, there is an additional step (located in `Model.update_program`) that compares two evaluation results (`Result.t`) using a structural equality[9] check.

This step is also very slow, so we memoize it by environments. We implement a manual structural checking algorithm, `DHExp.fast_equals`. For any leaf node (node with no subexpressions), the value of the node is compared for equality. For branch nodes (nodes with subexpressions), the nodes are equal if subexpressions are equal and if the node's properties are equal. Importantly, the equality check for environments is simply to check if the environment identifiers are equal.

A comparison of the performance before and after this change can be visualized in Table 7.2.

[9]Structural equality (==) is implemented by recursively checking the value equality of two expressions, and can be thought of as a tree traversal.

Chapter 6

Implementation of fill-and-resume

6.1 Motivation

Consider the program shown in Listing 4. In this program, the calculation of $x = \text{fib } 30$ is arbitrarily chosen to represent a computationally-expensive operation. The result of this program is

$$[f \leftarrow [\varnothing]\lambda x.\{\ldots\}, x \leftarrow 832040](\!|\,|\!)^1$$

Now, if we want to "fill" hole 1 with the expression $x + 2$, then it would seem extremely wasteful to have to re-compute the value of x. After all, the computation remains exactly the

```
let f : Int → Int =
  λ x . {
    case x of
    | 0 ⇒ 0
    | 1 ⇒ 1
    | n ⇒ f (n - 1) + f (n - 2)
    end
  }
in x = f 30
in (|)^1
```

Listing 4: A sample program with an expensive calculation stored in a hole's environment

same, and the only part that we are changing uses the result of the previous computation. Moreover, we realize that the end result stores the computed value of x in the hole's closure's environment. Rather than filling the expression $x + 2$ in the hole in the original program, we may instead fill the hole in the evaluated program result, and "resume" evaluation.

$$[f \leftarrow [\varnothing]\lambda x.\{\dots\}, x \leftarrow 832040](x + 2)$$

$$832040 + 2$$

$$832042$$

Here we observe that the generalized closures surrounding holes and other stopped evaluations allow us to capture the environment for future computation.

This is the *fill-and-resume* (*FAR*) operation that is described in Hazelnut Live [2]. It is described in terms of a substitution-based evaluation semantics, and with respect to its theoretic foundations in contextual modal type theory (CMTT), but does not specify any details with regard to its implementation, such as the extraction of the expression and hole for the fill operation. Hazelnut Live also describes the practical memoization problem with the suggestion to "cache more than one recent edit state to take full advantage of hole filling" – we present a structural diffing[1] algorithm that easily allows us to fill a hole from an arbitrary past edit state.

6.2 The FAR process

The fill-and-resume process can be broken into the following sequence.

1. Obtain a previous edit state with which to fill from the model.

2. Determine whether a fill operation is appropriate. If it is not, perform regular evaluation of the program, and do not continue to the following steps.

[1] "Diffing" taken to mean the action of performing a (structural) diff operation between two edit states.

3. If the fill operation is valid, then obtain the fill parameters (the internal expression to fill, and the hole number from the previous edit state with which to fill).

4. Pre-process the evaluation result to prepare for (re-)evaluation.

5. Re-evaluate.

6. Pos-process the evaluation result for display purposes.

7. Update the model with the evaluation results.

These steps will be described in greater detail in the following sections.

6.2.1 Detecting the fill parameters via structural diff

Following the notation from [2], the fill operation $[\![d_{fill}/u_{fill}]\!]d_{result}$ indicates the fill of hole u_{fill} with expression d_{fill} in the expression d_{result}. d_{result} is the past program result with which to fill. We need a method to determine the fill parameters u_{fill} and d_{fill}.

A naïve algorithm for detecting fill parameters

One way to approach the problem of obtaining the hole number and filled expression is at action time. When constructing an expression, we can check if the cursor lies in a hole (either directly in an empty hole, or if there is an ancestor non-empty hole). When deleting an expression, we can check if the cursor lies in a non-empty hole. However, this method is somewhat short-sighted. What happens if we wish to make multiple edits, e.g., fill a hole with the number 12? Then there are two actions, and the second action is not a hole fill action.

1

12

We may remedy this specific case by grouping together consecutive construction actions[2]. However, we may also consider more complicated edit sequences that involve movement and deletion actions, potentially outside of the hole. Consider the following edit sequence.

$$2 + 3 * (\!|\,|\!)^1$$

$$2 + (\!|\,|\!)^2 * (\!|\,|\!)^1$$

$$2 + (\!|\,|\!)^1$$

$$2 + 5$$

$$2 + (5)$$

$$2 + (\!|\,|\!)^1 * (5)$$

$$2 + 3 * (5)$$

$$2 + 3 * (5 + (\!|\,|\!)^1)$$

The final edit state in this sequence is actually a valid fill of the first edit state of the sequence[3], such that $u = 1$ and $d = 5 + (\!|\,|\!)^1$. However, an algorithm to trace the edit actions to determine that this is a valid fill may be difficult, since there is an incomprehensible mix of construct, delete, and movement edit actions. Even worse, the edits actually go outside the original hole, which likely makes the algorithm intractable. Thus, we wish for a more robust solution that is independent of the edit sequence between two states.

Structural diffing between two edit states

Instead of observing the edit action, we may instead attempt to find the root of the difference between any two edit states, and determine if that is the the difference gives valid fill parameters. This has the benefit of being a relatively simple algorithm, while overcoming the limitation of the previous method because it is path-independent.

[2]Grouping together of actions is already performed to some level by the undo history, for visual purposes.
[3]There are actually multiple valid fill operations here. Another valid fill operation occurs between the third edit state and all of the following edit states.

The structural diff algorithm takes two expressions as input and returns one of three diff judgments[4]. $d_1 \trianglerighteq d_2$ indicates *no diff* between d_1 and d_2. $d_1 \triangleright d_2$ indicates a *non-fill diff* from d_1 to d_2. $d_1 \blacktriangleright_d^u d_2$ indicates a *fill diff* of hole u with expression d from d_1 to d_2.

We also define two shorthand operators for notational convenience. $d_1 \not\trianglerighteq_d^u d_2$ indicates *some (non-empty) diff* from d_1 to d_2, which may or may not be a fill difference. $d_1 \triangleright_d^u d_2$ indicates *any diff* (potentially no diff). Both notations are used to avoid writing multiple similar rules, where the only change in the rules is diff judgment type.

[TODO: put the following in a figure, and move to another file]

$\boxed{d_1 \not\trianglerighteq_d^u d_2}$ Some (non-empty) diff between d_1 and d_2.

$$\frac{d_1 \triangleright d_2}{d_1 \not\trianglerighteq_{\varnothing}^{\varnothing} d_2}\ \text{SDiffNFDiffSome} \qquad \frac{d_1 \blacktriangleright_d^u d_2}{d_1 \not\trianglerighteq_d^u d_2}\ \text{SDiffFDiffSome}$$

$$\frac{d_1 \not\trianglerighteq_{\varnothing}^{\varnothing} d_2}{d_1 \triangleright d_2}\ \text{SDiffSomeNFDiff} \qquad \frac{d_1 \not\trianglerighteq_d^u d_2}{d_1 \blacktriangleright_d^u d_2}\ \text{SDiffSomeFDiff}$$

$\boxed{d_1 \triangleright_d^u d_2}$ Any (possibly-empty) diff between d_1 and d_2.

$$\frac{d_1 \trianglerighteq d_2}{d_1 \triangleright_{\varnothing}^{\cdot} d_2}\ \text{ADiffNoDiffAny} \qquad \frac{d_1 \triangleright d_2}{d_1 \triangleright_{\varnothing}^{\varnothing} d_2}\ \text{ADiffNFDiffAny} \qquad \frac{d_1 \blacktriangleright_d^u d_2}{d_1 \triangleright_d^u d_2}\ \text{ADiffFDiffAny}$$

$$\frac{d_1 \triangleright_{\varnothing}^{\cdot} d_2}{d_1 \trianglerighteq d_2}\ \text{ADiffAnyNoDiff} \qquad \frac{d_1 \not\trianglerighteq_{\varnothing}^{\varnothing} d_2}{d_1 \triangleright d_2}\ \text{ADiffAnyNFDiff} \qquad \frac{d_1 \not\trianglerighteq_d^u d_2}{d_1 \blacktriangleright_d^u d_2}\ \text{ADiffAnyFDiff}$$

We break up the diff judgments into cases. First, we may consider the case of two holes of different *expression forms*[5].

For convenience, we define the judgment $d_1 \sim d_2$ to mean that d_1 and d_2 are of the same expression form[6]. We may more concretely express this judgment by the following rules.

[4]The following notations for diffing are chosen somewhat arbitrarily. The triangle seems appropriate because it has a variant with an equals bar ($\trianglerighteq$), as well as a "no fill" ($\triangleright$) and "fill" variant ($\blacktriangleright$). The triangle is also horizontally asymmetric, which mirrors the fact that the diff relation is asymmetric.

[5]We use the term *expression form* or *expression variant* to indicate the variant types of `DHExp.t`. For example, empty holes and constants of the base type are different expression forms. Empty holes and non-empty holes are also different forms per the grammar.

[6]The relation $\sim$ is already defined to mean type consistency when applied to types. This interpretation applies when the relation is applied to internal expressions.

$$\frac{}{c_1 \sim c_2}\ \text{FEqConst} \qquad \frac{}{x_1 \sim x_2}\ \text{FEqVar} \qquad \frac{}{\lambda x.d_1 \sim \lambda x.d_2}\ \text{FEqLam} \qquad \frac{}{e_1\ e_2 \sim e_1'\ e_2'}\ \text{FEqAp}$$

$$\frac{}{e : \tau \sim e' : \tau'}\ \text{FEqAsc} \qquad \frac{}{(\!|)^u \sim (\!|)^{u'}}\ \text{FEqEHole} \qquad \frac{}{(\!|d|\!)^u \sim (\!|d'|\!)^{u'}}\ \text{FEqNEHole}$$

If the two expressions have different forms, then the current node is necessarily the diff root. It is a fill diff iff the left expression is a hole.

$$\frac{(\!|)^u \nsim d_2}{(\!|)^u \blacktriangleright^u_{d_2} d_2}\ \text{DFNEqEHole} \qquad \frac{(\!|d|\!)^u \nsim d_2}{(\!|d|\!)^u \blacktriangleright^u_{d_2} d_2}\ \text{DFNEqNEHole}$$

$$\frac{d_1 \neq (\!|)^u \qquad d_1 \neq (\!|d|\!)^u \qquad d_1 \nsim d_2}{d_1 \triangleright d_2}\ \text{DFNEqNonHole}$$

If the two expressions have the same form, then we need to check the root node and its subexpression(s). For expressions with no subexpressions, there is a non fill diff iff the expressions differ.

$$\frac{c_1 \neq c_2}{c_1 \triangleright c_2}\ \text{DFEqConstNEq} \qquad \frac{}{c \trianglerighteq c}\ \text{DFEqConstEq} \qquad \frac{x_1 \neq x_2}{x_1 \triangleright x_2}\ \text{DFEqVarNEq}$$

$$\frac{}{x \trianglerighteq x}\ \text{DFEqVarEq}$$

For expressions with a single subexpression, we first check if there are any differences, ignoring the subexpression. If there is a difference, then the current node is the non fill diff root. Otherwise, we pass through the diff from the child node.

$$\frac{x_1 \neq x_2}{\lambda x_1 : \tau_1 . d_1 \rhd \lambda x_2 : \tau_2 . d_2} \text{ DFEqLamNEq}_1 \qquad \frac{\tau_1 \neq \tau_2}{\lambda x : \tau_1 . d_1 \rhd \lambda x : \tau_2 . d_2} \text{ DFEqLamNEq}_2$$

$$\frac{d_1 \rhd_d^u d_2}{\lambda x : \tau . d_1 \rhd_d^u \lambda x : \tau . d_2} \text{ DFEqLamEq} \qquad \frac{\tau_1 \neq \tau_2}{d_1 : \tau_1 \rhd d_2 : \tau_2} \text{ DFEqAscNEq}$$

$$\frac{d_1 \rhd_d^u d_2}{d_1 : \tau \rhd_d^u d_2 : \tau} \text{ DFEqAscEq}$$

The last case to check for non-hole expressions are expressions with more than one subexpression. In this case, we first check if there exist any differences outside the subexpressions, which would result in a non fill diff rooted at the current node. Otherwise, if there are no subexpression diffs, then the result is no diff. If more than one subexpression has a diff, then the diff is a non fill diff rooted at the current node. The last case is when exactly one child has a diff, which would be passed through. This is illustrated below with the binary function application expression form.

$$\frac{d_1 \trianglerighteq d_1' \qquad d_2 \trianglerighteq d_2'}{d_1\ d_2 \trianglerighteq d_1'\ d_2} \text{ DFEqApEq}_1 \qquad \frac{d_1 \not\trianglerighteq_d^u d_1' \qquad d_2 \not\trianglerighteq_{d'}^{u'} d_2'}{d_1\ d_2 \trianglerighteq d_1'\ d_2} \text{ DFEqApEq}_2$$

$$\frac{d_1 \trianglerighteq d_1' \qquad d_2 \not\trianglerighteq_d^u d_2'}{d_1\ d_2 \not\trianglerighteq_d^u d_1'\ d_2} \text{ DFEqApEq}_3 \qquad \frac{d_1 \not\trianglerighteq_d^u d_1' \qquad d_2 \trianglerighteq d_2'}{d_1\ d_2 \not\trianglerighteq_d^u d_1'\ d_2} \text{ DFEqApEq}_4$$

In the minimal λ-calculus grammars specified for Hazel, the only expression form of plural subexpression arity is function application, but the following description extends to higher numbers of subexpressions (such as the case for `case` expressions with arbitrary numbers of rules.

The last case to consider is the comparison of two hole expressions of the same form. The empty hole case is very similar to the nullary subexpression case. The non-empty hole case is very similar to the unary subexpression case, except for a special rule that propagates non-fill diffs upwards to be a fill diff rooted in the current hole. This allows for diffs that are

not rooted directly in a hole to be filled in their nearest non-empty hole parent node.

$$\frac{u \neq u'}{(\!|\,|\!)^u \blacktriangleright^u_{(\!|\,|\!)^{u'}} (\!|\,|\!)^{u'}}\ \text{DFEqEHoleNEq} \qquad \frac{}{(\!|\,|\!)^u \trianglerighteq (\!|\,|\!)^u}\ \text{DFEqEHoleEq}$$

$$\frac{u \neq u'}{(\!|d|\!)^u \blacktriangleright^u_{(\!|d'|\!)^{u'}} (\!|d'|\!)^{u'}}\ \text{DFEqNEHoleNEq} \qquad \frac{d \trianglerighteq d'}{(\!|d|\!)^u \trianglerighteq (\!|d'|\!)^u}\ \text{DFEqNEHoleEq}_1$$

$$\frac{d \blacktriangleright^{u'}_{d''} d'}{(\!|d|\!)^u \blacktriangleright^{u'}_{d''} (\!|d'|\!)^u}\ \text{DFEqNEHoleEq}_2 \qquad \frac{d \triangleright d'}{(\!|d|\!)^u \blacktriangleright^u_{(\!|d'|\!)^u} (\!|d'|\!)^u}\ \text{DFEqNEHoleEqProp}$$

[TODO: metatheorems regarding the diff algorithm]

The diff algorithm begins by performing the structural diff between the elaborated program state of a past edit state d_{old} and the current edit state d_{cur}. A FAR operation is only valid if the diff judgment is a fill diff $d_{old} \blacktriangleright^u_d d_{cur}$, which gives us the FAR parameters u and d.

Performance tradeoffs of the two detection algorithms

In the most naïve form, the previous algorithm's efficiency is $O(\log E)$, where E is the number of expression nodes in the program, if we assume that the depth of an expression node is logarithmic with respect to the total number of expression nodes. That algorithm only has to traverse up the ancestors to decide whether the edit lies in a hole.

The structural diff algorithm presented in this section is $O(E)$, since it traverses each node (once) until it finds a difference. However, if one travels backwards multiple edit states, then the cost is $O(SE)$, where S is the number of edit states compared using this algorithm. While this is much more expensive than the previous algorithm, we assume that the program size is relatively small, causing delay only on the order of milliseconds. However, it may be able to find a valid fill-and-resume in many more cases than the previous algorithm, potentially saving a much longer repeated evaluation time.

6.2.2 Pre-processing the evaluation result for re-evaluation

Before beginning the re-evaluation, we would like to substitute all instances of the hole in the previous evaluation result d_{result} with the substituted expression. Each time a the hole u_{fill} is encountered, it is replaced with the fill expression d_{fill}. This way, we can simply reinvoke the evaluation function on the past program result and expect it to resume the evaluation.

There are a few nuances here that should be addressed. First of all, we acknowledge another benefit of the generalized closure variant. If the environment was still baked into the hole, and the hole was replaced with an expression, we would need an additional mechanism to remember the hole's environment. This becomes more messy when the hole doesn't lie directly in a hole closure (i.e., if the hole lies outside the evaluation boundary). Closures are simply recursed through in this pre-processing step, and their environments will still be available even when the hole gets replaced with the fill expression.

We note that the pre-processing acts on the un-post-processed previous evaluation result d_{result}. This is because the internal expression directly from evaluation and the result after post-processing have different properties or invariants. We do not want to invalidate the properties that are expected to be upheld during evaluation, such as the fact that the body of any λ-abstraction lies outside the evaluation boundary (whereas post-processing will modify function bodies).

Another issue to tackle is the problem that closures were previously considered to be final values, and would not be re-evaluated. Technically, we may re-evaluate closures; since the evaluation function is idempotent, this will not yield the incorrect result, but it is needlessly inefficient. However, we will need to re-evaluate closures during FAR re-evaluation, since the fill expression will necessarily lie within some closure.

[**TODO: write the previous statement as a metatheorem**]

The (re-)evaluation of a closure is given by the rule EClosure.

$$\frac{\sigma \vdash d \Downarrow d'}{\sigma' \vdash [\sigma]d \Downarrow d'} \text{ EClosure}$$

For efficiency reasons, we will only want to re-evaluate all closures in the result exactly once. To do this, we set a flag for the closure that indicates that it should be re-evaluated. This preprocessing step will recurse through d_{result} and set the flag to true for all closures. All closures that result from an evaluation judgment will have the flag set to false. The above closure evaluation rule is only for closures with the re-eval flag set to true. Closures with the re-eval flag set to false will act as values and evaluate to themselves, which is the same as the original evaluation behavior described in Section 4.2. We denote closures with the re-eval flag set to false using the established notation for closures $[\sigma]d$[7], and denote closures with the re-eval flag set to true by $[\![\sigma]\!]d$[8].

Finally, we may consider the issue of multiple instances of the same hole closure[9]. If we substitute the hole, then we lose the information about the holes instance, and thus cannot memoize the evaluations of the same hole instance by environment number. A solution to this is to introduce another `DHExp.t` variant `FillExp(HoleClosureId.t, DHExp.t)` that indicates the hole closure number as well as the expression to fill. This will be denoted using a hole with a subscript $(\!|d|\!)_i$. The preprocessing expression will fill a hole u_{fill} with this expression rather than the expression d_{fill} directly. During evaluation, this data structure will facilitate the memoization of hole closures.

[7]This allows previous discussions of closures to remain valid, since they take the interpretation that the re-eval flag is set to false.

[8]Using the double-square bracket notation also reinforces the fact that re-evaluation is tied with fill-and-resume, which also uses double-square brackets.

[9]Hole closure refers to the connotation from Section 5.1: instances of hole u_{fill} that share the same (physical) environment.

6.2.3 Modifications to evaluation to allow for re-evaluation

Re-evaluation during fill-and-resume is mostly the same as regular evaluation, but now we need to keep in mind the considerations from pre-processing the previous program result.

First of all, closures will have been marked for re-evaluation by the pre-processing step. This means that the closure environment will first be recursively re-evaluated, following by the closure body. This ensures that the entire program result is fully evaluated[10]. This is similar to the EClosure rule, except that now we add the consideration of the re-eval flag to avoid re-evaluating closures more than once, and also recurse evaluation through variable bindings in the bound environment. We introduce a (re-)evaluation judgment for environments, which simply maps the evaluate operation over the bindings of an environment. Closures with the re-eval flag set to false will have the regular evaluation rule.

$$\frac{}{\sigma' \vdash [\sigma]d \Downarrow [\sigma]d}\ \text{EEClosure} \qquad \frac{\sigma \Downarrow \sigma'' \qquad \sigma'' \vdash d \Downarrow d'}{\sigma' \vdash [\![\sigma]\!]d \Downarrow d'}\ \text{EREClosure}$$

$$\frac{\sigma \Downarrow \sigma' \qquad \varnothing \vdash d \Downarrow d'}{\sigma, x \leftarrow d \Downarrow \sigma', x \leftarrow d'}\ \text{EREEnv} \qquad \frac{}{\varnothing \Downarrow \varnothing}\ \text{EREEnvNull}$$

[TODO: metatheorem about how all closures from d_{result} will re-evaluated (exactly once), assuming that evaluation terminates]

The other difference that we have to deal with are memoizing the evaluation of the filled hole expressions. Per the discussion of the pre-processing step, all filled expressions will exist in a wrapper that indicates the hole number. We may simply memoize the results by that hole number. This requires us to thread some state throughout our evaluation[11].

[10] Note that there is now an "inversion" of evaluation order, in that we cannot expect the environment to be fully evaluated before it is encountered in a closure. In an ordinary evaluation, we would expect all the bindings in the environment to have been evaluated before they have been stored in the environment.

[11] Luckily, threading state through evaluation is useful for other purposes as well, such as keeping track of the current `EvalEnvId.t`, and keeping track of evaluation state. These are grouped together under one data structure, `EvalState.t`, for ease of implementation.

$$\frac{}{\sigma, (\rho, i \leftarrow d) \vdash (\!|d'|\!)_i \Downarrow d \dashv \rho}\ \text{EFillMemoNew}$$

$$\frac{(i \leftarrow d) \notin \rho \qquad \sigma, \rho \vdash d' \Downarrow d'' \dashv \rho'}{\sigma, \rho \vdash (\!|d'|\!)_i \Downarrow d'' \dashv \rho', i \leftarrow d''}\ \text{EFillMemoNew}$$

$$\frac{d \neq (\!|d|\!)_i \qquad \sigma \vdash d \Downarrow d'}{\sigma, \rho \vdash d \Downarrow d' \dashv \rho}\ \text{EFillOther}$$

In these judgments, we introduce a new *fill memoization context* $\rho : i \mapsto d$, a mapping of hole closure numbers to expressions[12]. A new *fill-memoized evaluation judgment* $\sigma^-, \rho^- \vdash d^- \Downarrow d'^+ \dashv \rho'^+$ describes the evaluation with memoization of hole fills. When a hole closure number is encountered for the first time, the expression is evaluated normally and added to ρ; otherwise, the evaluated result of the fill is simply looked up from ρ. Note that memoization requires ρ to be threaded throughout the evaluation, i.e., it is treated both as an input and output of the evaluation judgment.

This fill-memoized evaluation judgment subsumes the normal evaluation judgment. For all non-hole-fill expressions, it performs the normal evaluation judgment and returns the fill memoization context unchanged.

6.2.4 Post-processing resumed evaluation

The postprocessing algorithm remains unchanged from before. Note that an evaluated program result should never include either of the new forms $(\!|d|\!)_i$ or $[\![\sigma]\!]d$; these should all have been encountered and evaluated out. In other words, the evaluation result from re-evaluation during a FAR should be indistinguishable from the evaluation result from a regular evaluation, and thus the postprocessing process is unchanged.

[TODO: metatheorems to back up the above]

[12]The symbol ρ was chosen arbitrarily. It is simply the Greek letter before σ.

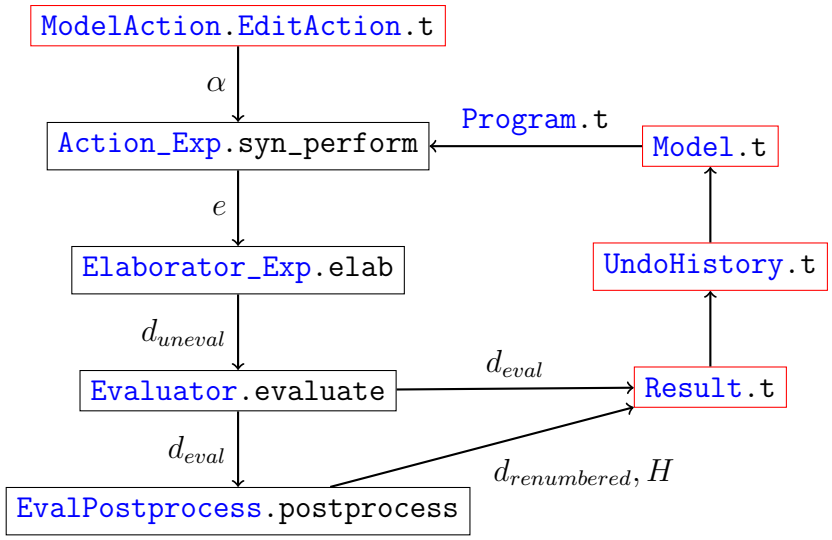


Figure 6.1: Previous action call graph

6.3 Entrypoint to the FAR algorithm

An abstracted call graph for the action model (the process of responding to an action, up to evaluation) in Hazel is diagramed in Figure 6.1. Red boxes indicate important data structures. Black boxes indicate important steps in the process of responding to an action. The lines between boxes roughly indicate the flow of the program and other important data structures related to the process.

An edit action triggers the bidirectional action semantics, which generates an updated program edit state. This is then elaborated to an internal expression, evaluated, and then postprocessed. The results of evaluation are stored in `Result.t`.

The updated abstract call graph is shown in Figure 6.2. Fill and resume is fundamentally an operation on internal expressions, so we attempt to detect a hole fill operation after elaboration of the current edit state. The structural diff operation requires information about at least one previous edit state from the model's `UndoHistory.t`. The result of the previous evaluation is preprocessed, before being evaluated (using the updated evaluation judgments) and postprocessed as usual.

Since this is a very high-level view of the program, many details are abstracted out. For

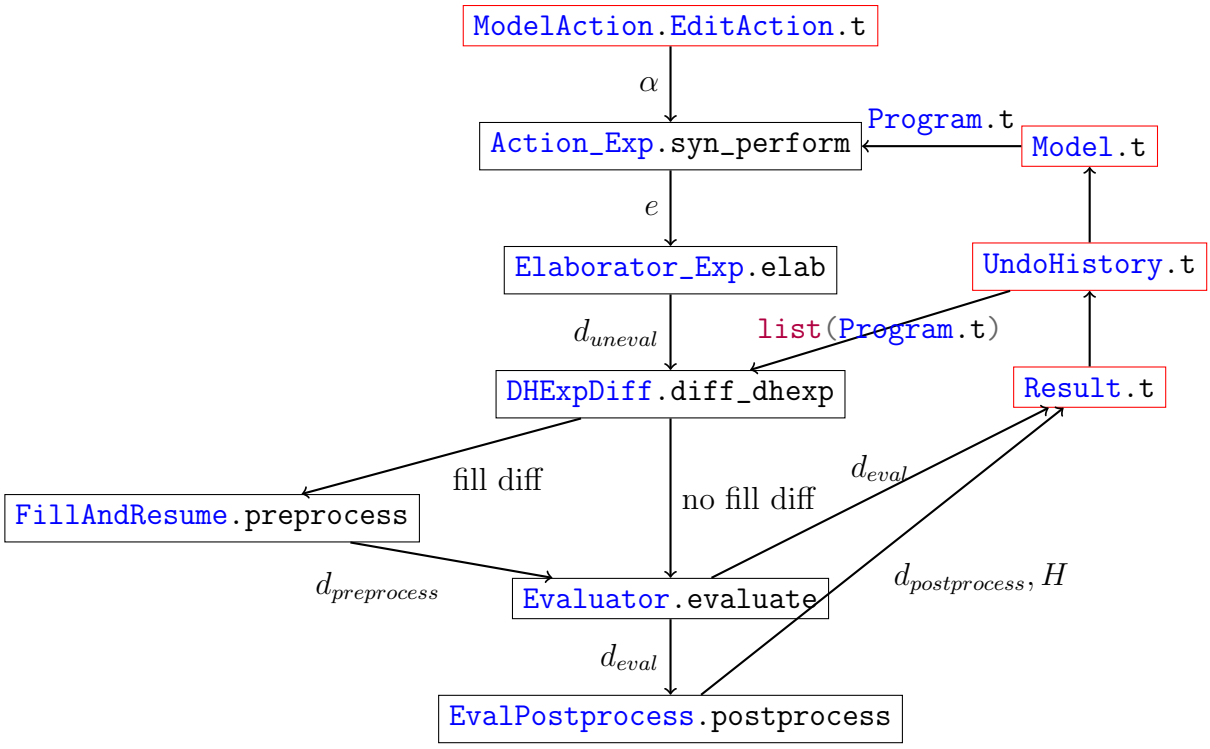


Figure 6.2: Current action call graph

example, the evaluation function is memoized by the function `Program.get_result`, which facilitates the normal evaluation process. We will need to modify this memoization to also memoize the FAR evaluation results.

6.4 FAR examples

We revisit the program from Listing 4, reproduced in Figure 6.3a. This hole fill operation is very simple, replacing a hole with a value. We also observe the behavior of the preprocessing operation, which marks closures for re-evaluation and encapsulates filled expressions with the hole closure number for memoization.

Now, consider the program shown in Figure 6.4. In this fill operation, we introduce a new static type error (non-empty hole). The non-empty hole is inserted automatically during the action semantics and appears naturally in the diff, so we do not need to perform any special handling for it. Similarly, in the case of Figure 6.5, a non-empty hole is removed by filling

```
let f : Int → Int =
  λ x . {
    case x of
    | 0 ⇒ 0
    | 1 ⇒ 1
    | n ⇒ f (n - 1) + f (n - 2)
    end
  }
in x = f 30
in ⦇⦈¹
```

(a) Previous edit state

```
let f : Int → Int =
  λ x . {
    case x of
    | 0 ⇒ 0
    | 1 ⇒ 1
    | n ⇒ f (n - 1) + f (n - 2)
    end
  }
in x = f 30
in x + 2
```

(b) Filled edit state

$[x \leftarrow 832040]⦇⦈^{1:1}$

(c) Previous program result

$u = 1, d = x + 2$

(d) Detected fill parameters

$[\![x \leftarrow 832040]\!]⦇x + 2⦈_1$

(e) Preprocessed program result

832042

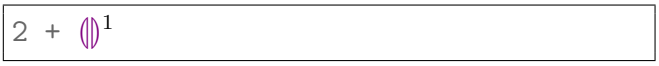
(f) Resumed program result

Figure 6.3: FAR simple example

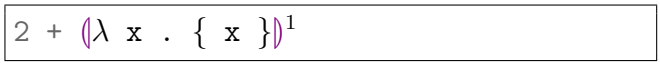
with a type-consistent expression. This allows static type errors to be "fixed" and evaluation to resume past where it had stopped.

In Figure 6.6 we fill a hole that does not exist directly in the result, but exists in a hole closure environment. This illustrates the need to recursively re-evaluate closure environments.

The program in Figure 6.7 shows the necessity of memoizing by hole closure number, in order to preserve the same result as if evaluating normally. In this example, we have two instances of hole 1 in the result. Since there is only one instantiation of hole 1, these are two instances of the same hole closure. If there were no memoization of filled expressions, then the filled expression $f\ 1$ would be evaluated twice. This is problematic because function application generates new environments (and new environment identifiers), so the two instances would have different closures and thus be considered two separate hole closures, even when they come from the same instantiation. Since the filled expression is memoized,



(a) Previous edit state



(b) Filled edit state

$2 + [\varnothing]⦇⦈^{1}$

(c) Previous program result

$u = 1, d = ⦇\lambda x.x⦈^{1:1}$

(d) Detected fill parameters

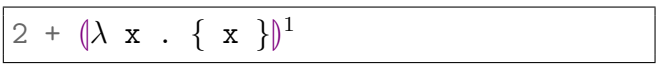
$2 + [\![\varnothing]\!]⦇⦇\lambda x.x⦈^{1}⦈_{1}$

(e) Preprocessed program result

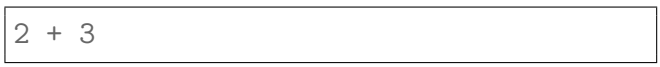
$2 + [\varnothing]⦇[\varnothing]\lambda x.x⦈^{1:1}$

(f) Resumed program result

Figure 6.4: FAR introduce static type error example



(a) Previous edit state



(b) Filled edit state

$2 + [\varnothing]⦇[\varnothing]\lambda x.x⦈^{1:1}$

(c) Previous program result

$u = 1, d = 3$

(d) Fill parameters

$2 + [\![\varnothing]\!]⦇3⦈_{1}$

(e) Preprocessed program result

5

(f) Resumed program result

Figure 6.5: FAR remove static type error example

(a) Previous edit state

(b) Filled edit state

$[x \leftarrow [\varnothing]⦇⦈^{1:1}]⦇⦈^{2:1}$

(c) Previous program result

$u = 1, d = 2$

(d) Fill parameters

$[\![x \leftarrow [\![\varnothing]\!]⦇2⦈_{1}]\!]⦇⦈^{2:1}$

(e) Preprocessed program result

$[x \leftarrow 2]⦇⦈^{2:1}$

(f) Resumed program result

Figure 6.6: FAR fill hole in hole environment example

```
let f = λ x . ⦶¹ in
let x = ⦶² in
x + x
```

(a) Previous edit state

```
let f = λ x . ⦶¹ in
let x = f 1 in
x + x
```

(b) Filled edit state

$$[f \leftarrow [\varnothing]\lambda x.\{⦶^{1:1}\}]⦶^{2:1} + [f \leftarrow [\varnothing]\lambda x.\{⦶^{1:1}\}]⦶^{2:1}$$

(c) Previous program result

$$u = 2, d = f\ 1$$

(d) Fill parameters

$$[\![f \leftarrow [\![\varnothing]\!]\lambda x.\{⦶^{1:1}\}]\!](\!|f\ 1|\!)_1 + [\![f \leftarrow [\![\varnothing]\!]\lambda x.\{⦶^{1:1}\}]\!](\!|f\ 1|\!)_1$$

(e) Preprocessed program result

$$[x \leftarrow 1]^i⦶^{1:1} + [x \leftarrow 1]^i⦶^{1:1}$$

(f) Resumed program result

Figure 6.7: FAR hole closure memoization example

the two instances of the hole have closures with the same environment identifier (indicated by the same superscript i on the closure environments).

The necessity of memoization in this case is related to the idea of the "inversion" of evaluation mentioned in Section 6.2.3: during normal evaluation, we expect that a variable may be referenced in multiple places after it is evaluated. However, during a resumed evaluation, we may encounter multiple unevaluated instances of the same instantiation; the first time it is encountered and evaluated is the de facto "first instantiation" of that expression.

6.4.1 Noteworthy non-examples

Dynamic type errors

One may wonder if dynamic type errors (cast failure) have any nice relation to fill-and-resume, but it turns out that they are not treated much different than other non-hole expressions. We may not fill a cast failure directly, because it is not in a hole. We may only remove a cast failure if it lies in a non-empty hole that is filled.

We may introduce new cast failures by filling an expression that is assigned to type hole (something of dynamic type). However, this is no different than introducing any well-typed expression and performing normal evaluation.

Thus the only interesting cases of introducing or remoing holes lies in the case of static type errors (non-empty holes) as described previously.

Infix operators

Filling holes in an infix operator sequence may not result in a hole fill due to infix operator precedences, despite the initial appearance. Consider the example shown in Figure 6.8. Say we construct a binary plus operator in the hole. One might expect the fill operation to be $[\![(\!|\,|\!)^2 + (\!|\,|\!)^1/1]\!]d_{result}$. However, we need to be careful: the multiplication operator has a higher precedence than the addition operator, causing the AST to have a different structure outside of where hole 1 was in the original edit state. It may be more clear if we write the latter edit state with its implicit parentheses: `(1 * ⦇⦈2) + (⦇⦈1 * 2)`.

If, however, hole 1 in the original edit state was a parenthesized expression, and the sum expression shape was constructed inside the parenthesized expression, then the AST wouldn't change (as a parenthesized expression has higher precedence than the outer multiplication operation) and it would be a valid fill operation.

(a) Previous edit state (b) Filled edit state

Figure 6.8: FAR infix operator fill

6.5 Tracking evaluation state

It may be useful to thread some state throughout the evaluation process. This means that changes to this state by an earlier invocation to evaluate will affect later evaluations.

The concept of keeping state has already appeared in several contexts. For postprocessing, the memoized hole closure info keeps track of seen environments (Section 4.4). For evaluation, we keep track of the next unique environment identifier and memoize the evaluation of filled expressions (Section 6.2.3). A theoretical discussion is revisited in Section 8.2.

This becomes relevant again (with respect to evaluation) because resuming evaluation requires us to store the state after evaluation, and restore the state after evaluation. Adding more state variables also makes the implementation messier by increasing the number of variables to pass to and return from each of the numerous calls to `Evaluator.evaluate`.

The nice solution to this is to group together all evaluation state under a single data structure, `EvalState.t`. This one state variable is passed around to all calls to evaluate, and is stored to and restored from `Result.t`, which stores all information about a program's evaluated result (including the evaluated expression and the hole closure information). Adding state then becomes trivial, because we only have to modify this data structure rather than all of the calls to `Evaluator.evaluate`.

For sake of brevity, we will not update the judgments to include this state.

6.5.1 Step counting

With the `EvalState.t` data structure, it is now trivial to keep track of evaluation statistics. Sample statistics might include step counting and number of times a particular evaluation

trace has been paused and resumed.

Step counting may be implemented simply by incrementing the count on each call to `Evaluator.evaluate`. This may be useful to stop long-running executions after a particular number of (possibly user-specified) execution steps in order to prevent program crashes. It is also useful for our purposes to track how efficient fill-and-resume is: during a fill operation, one may observe the difference in the number of steps before and after the resumed evaluation, and compare it to the case if the evaluation had begun from scratch.

6.6 Differences from the substitution model

The FAR operation is much simpler when evaluating with substitution rather than with environments. With substitution, we eliminate the need for pre- and post-processing steps, and dealing with re-evaluating closures. However, this comes at the cost of not being able to memoize repeated instances of the same hole closure. The original description of FAR from [2] is reproduced below. The equals (=) operator is used to indicate the FAR operation here; it is meant to symbolize the mix of substitution and evaluation that is presented here. The hole closure notation from [2], in which the hole closure is represented using a subscript, is also used here, since the substitution model doesn't have the concept of separate closures.

[TODO: move this to a figure later]

$\boxed{[\![u/d]\!]d' = d''}$ d''' is obtained by filling hole u with d in d'

$$\frac{}{[\![u/d]\!]c = c}\ \text{FillSubConst} \qquad \frac{}{[\![u/d]\!]x = x}\ \text{FillSubVar}$$

$$\frac{}{[\![u/d]\!]\lambda x : \tau.d' = \lambda x : \tau.[\![u/d]\!]d'}\ \text{FillSubLam} \qquad \frac{}{[\![u/d]\!]d_1\ d_2 = ([\![u/d]\!]d_1)\ ([\![u/d]\!]d_2)}\ \text{FillSubAp}$$

$$\frac{}{[\![u/d]\!]\,⦇⦈^{u}_{\sigma} = [[\![u/d]\!]\sigma]d}\ \text{FillSubEHole}_1 \qquad \frac{u \neq u'}{[\![u/d]\!]\,⦇⦈^{u'}_{\sigma} = ⦇⦈^{u'}_{\sigma}}\ \text{FillSubEHole}_2$$

$$\frac{}{[\![u/d]\!]\,⦇d'⦈^{u}_{\sigma} = [[\![u/d]\!]\sigma]d}\ \text{FillSubNEHole}_1 \qquad \frac{u \neq u'}{[\![u/d]\!]\,⦇d'⦈^{u'}_{\sigma} = ⦇[\![u/d]\!]d'⦈^{u'}_{[\![u/d]\!]\sigma}}\ \text{FillSubNEHole}_2$$

$$\frac{}{[\![u/d]\!]d'\langle\tau \Rightarrow \tau'\rangle = ([\![u/d]\!]d')\langle\tau \Rightarrow \tau'\rangle}\ \text{FillSubCast}$$

$$\frac{}{[\![u/d]\!]d'\langle\tau \Rightarrow ⦇⦈ \not\Rightarrow \tau'\rangle = ([\![u/d]\!]d')\langle\tau \Rightarrow ⦇⦈ \not\Rightarrow \tau'\rangle}\ \text{FillSubFailedCast}$$

$\boxed{[\![u/d]\!]\sigma = \sigma'}$ σ' is obtained by filling hole u with d in σ

$$\frac{}{[\![u/d]\!]\varnothing = \varnothing}\ \text{FillSubEnvNull} \qquad \frac{\sigma' = [\![u/d]\!]\sigma \qquad d'' = [\![u/d]\!]d'}{[\![u/d]\!]\sigma, x \leftarrow d' = \sigma', x \leftarrow d''}\ \text{FillSubEnv}$$

What remains the same between the environment model and the substitution model is the need for an algorithm to detect the fill parameters, for which the structural diff algorithm presented in Section 6.2.1 is still applicable.

Chapter 7

Evaluation of performance

To evaluate performance, benchmarks were carried out using the `TimeUtil.measure_time` utility. Benchmarks were carried out on Google Chrome 99 on Debian 10 on an Intel i3-2100 CPU. The times shown are a mean of three trials. Evaluation step counts are tracked in `EvalState.t` and count the number of calls to `Evaluator.evaluate` as described in Section 6.5.1.

There are a number of factors that may affect the consistency of the elapsed time benchmarks. Such factors include the quality of JSOO-generated Javascript, specifics of the Chrome V8 Javascript engine, inaccuracies in the Javascript timing function, and random system fluctuations.

Evaluation of performance of the environment model of evaluation and of the memoized hole tracking are performed on the `dev` branch and the `eval-environment` branches of the Hazel repository. The latter branch implements our changes.

The demonstration of FAR is performed on the `fill-and-resume-backend` branch. This branch is based on the `eval-environment` branch, and implements a one-step FAR operation. This branch does not achieve parity with the theoretical model of FAR presented in this work; unfinished work and future improvements to the current implementation of FAR are described in Section 9.1.

7.1 Evaluation of performance using the environment model

To evaluate the performance of evaluation using the environment model, we benchmark the performance of a computationally-expensive function, the tree-recursive Fibonacci function. This function is chosen because it is computationally expensive and does not have a deep recursion depth[1]. It is also a complete program, i.e., it does not have holes and the hole renumbering and postprocessing steps are not of concern here.

For this experiment, the builtin variables and functions are removed. Restoring the builtins would be very similar to the second program variation described in Section 7.1.2.

7.1.1 A computationally expensive fibonacci program

The quantitative results of this experiment are shown in Table 7.1. The results of evaluating Listing 5 for various values of n on the `dev` and `eval-environment` (abbreviated `e-e` in the legend) branches are shown in Figure 7.1. The `eval-environment` roughly decreases evaluation time by a small, roughly-constant factor for all values of n.

7.1.2 Variations on the fibonacci program

We try out a few variations of the fib(n) function, shown in Listing 5. Results are collected for $n \in \{22, 23, 24, 25, 26\}$. These numbers were chosen somewhat arbitrarily. They are large enough to allow for reproducible results, and small enough to prevent excessively long runtimes. The first variation is shown in Listing 6, which involves more global variables. The second variation is shown in Listing 7, in which an additional branch is added. This branch is never taken (as the third rule's pattern will always match), and it involves some instances of the variable f.

[1]This is because Hazel does not implement TCO, and thus would overflow the stack with too much (tail-)recursion.

```
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: A computationally expensive Hazel program with no holes

However, the two variations show the difference between evaluation with environments and evaluation with substitution. If we introduce additional global variables as in the first variation, we observe the behavior in Figure 7.2. The performance of the evaluation with substitution is virtually unchanged, while the performance of evaluation with environments increases. The increase is not linear. We expect the slowdown to be logarithmic with respect to the number of elements in the environment. On the other hand, if more variables are introduced in an unevaluated branch as in the second variation, then we observe the opposite effect. This is shown in Figure 7.3. The evaluation time when using substitution increases linearly with respect to the number of extra substitutions.

Combined, these variations show the expected behavior. The raw performance difference is not very large. Substitution "eagerly" evaluates by replacing all usages of the variable at binding time, whereas using environments "lazily" evaluates at lookup time. Introducing additional global variables increases the number of variables in each environment, slowing down evaluation with environments. However, this does not slow down substitution because these variables are never encountered after being bound. On the other hand, substitution necessarily traverses unevaluated branches, whereas evaluation with environment never reaches those branches.

```
let a = 0 in
let b = 0 in
let c = 0 in
let d = 0 in
let e = 0 in
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 6: Adding global bindings to the program in Listing 5

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

Listing 7: Adding variable substitutions to unused branches to the program in Listing 5

		Variables in unused branch					Extra global variables				
n	Regular	2	4	6	8	10	2	4	6	8	10
22	334	394	509	539	658	677	339	305	302	339	336
23	478	599	695	835	1116	1107	524	442	452	452	455
24	775	929	1214	1332	1518	1686	744	700	729	794	708
25	1233	1502	1874	2310	2398	2723	1171	1189	1134	1104	1231
26	2019	2391	2939	3399	3872	4417	1841	1747	1761	1773	1780

(a) `dev` branch

		Variables in unused branch					Extra global variables				
n	Regular	2	4	6	8	10	2	4	6	8	10
22	255	267	276	242	245	243	330	384	417	435	519
23	406	374	376	358	366	330	497	576	573	593	660
24	578	558	559	591	561	569	775	857	911	912	1037
25	851	883	871	864	888	908	1209	1363	1469	1473	1684
26	1318	1388	1382	1398	1399	1415	1935	2262	2302	2356	2492

(b) `eval-environment` branch

Table 7.1: Time (ms) to compute fib(n)

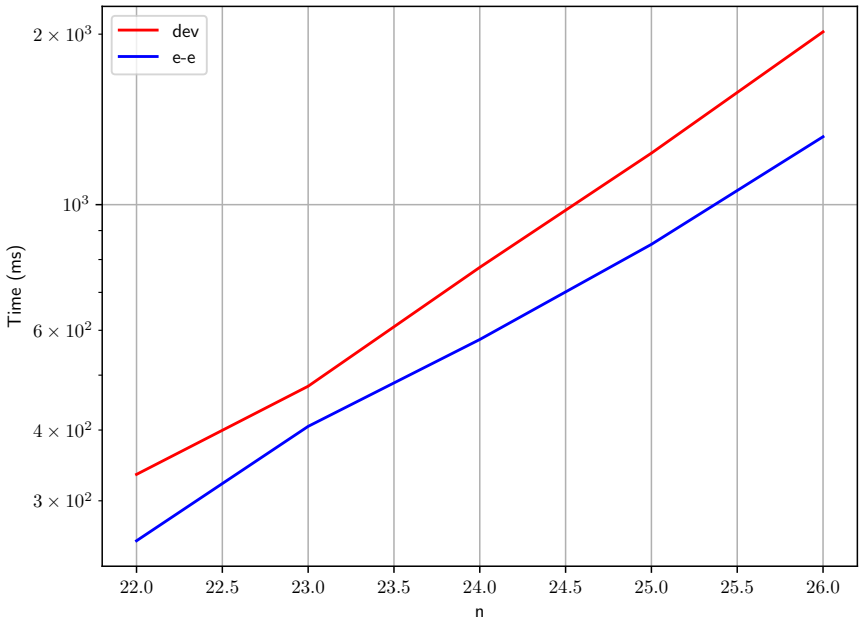


Figure 7.1: Performance of evaluating fib(n)

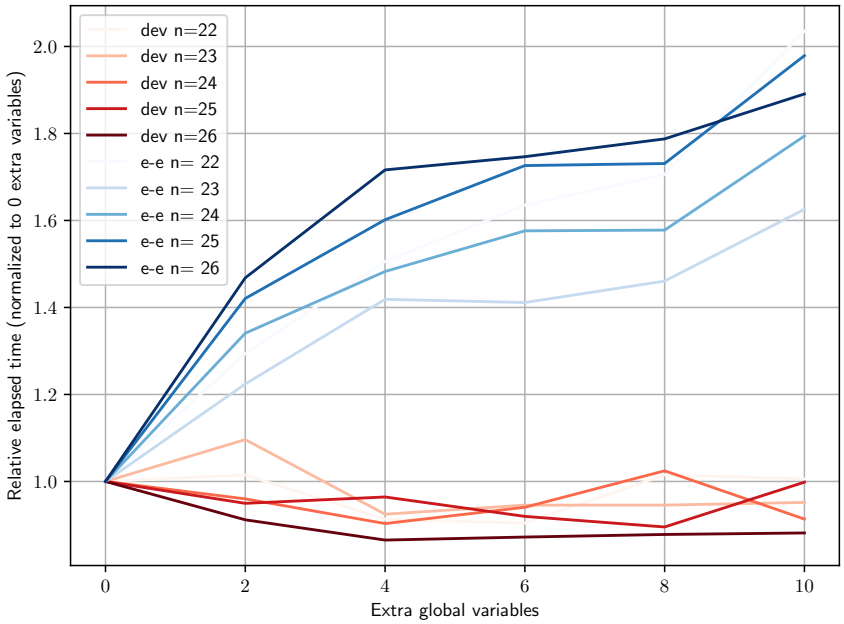


Figure 7.2: Performance of evaluating fib(n) with extra global variables

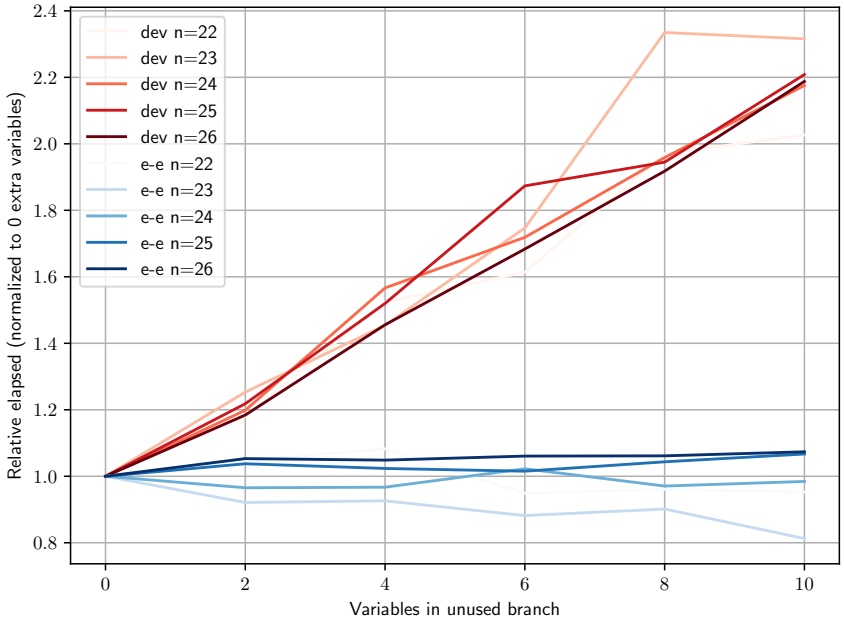


Figure 7.3: Performance of evaluating fib(n) with an unused branch

7.2 Postprocessing performance

Consider the set of programs described by Listing 3, which motivate the memoization of hole tracking (Section 5.3) and the fast structural equality checking algorithm (Section 5.5).

We illustrate the performance issues by evaluating the performance issue. The results are shown in tabular form in Table 7.2, and visually in Figure 7.4a. Due to the exponential blowup in elapsed time, we stop recording performance 15 `let` statements.

The exponential performance blowup occurs in three places: in evaluation (due to eager substitution of holes), in hole numbering during postprocessing, and in the structural equality check in `Model.update_program`. All three occur due to the lack of memoization of hole closurse with a shared environment.

We compare the performance to evaluation on the `eval-environment` branch which implements memoization of environments in hole numbering and structural equality. This is shown in Table 7.2, and visually in Figure 7.4b. The evaluation time is greatly improved; total evaluation time remains roughly constant and never exceeds 7ms.

7.3 FAR performance

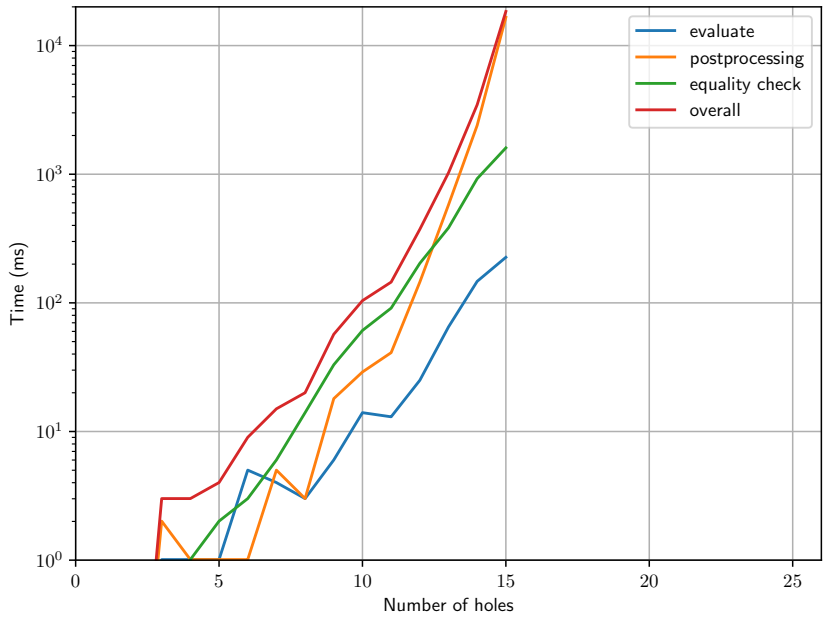
We explore two sample edit histories and the effect of FAR on the number of evaluation steps. The current implementation does not look back multiple edit states, so any fill operations will only occur if there is a valid fill from the previous edit state.

7.3.1 A motivating example

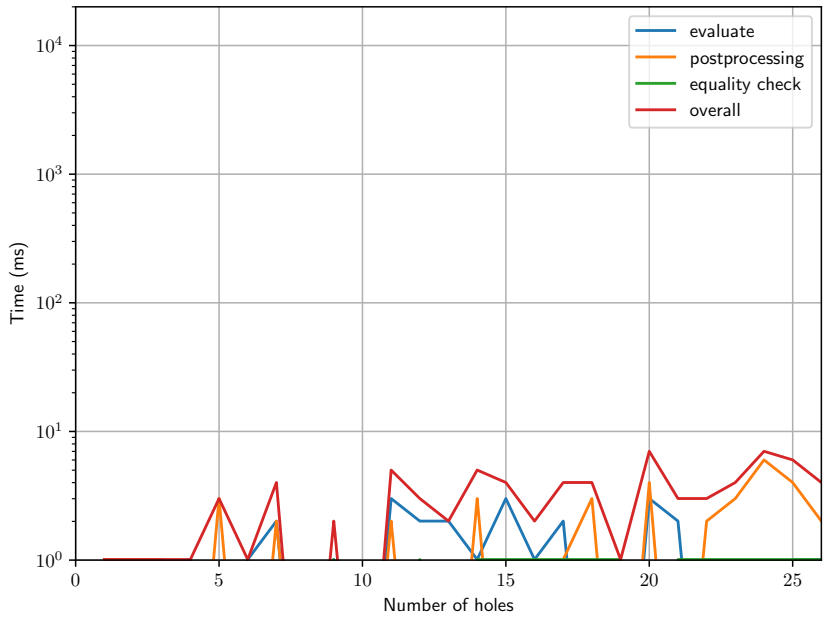
We have an example edit sequence shown in Table 7.3. This shows a possible edit sequence around the motivating example in Listing 4. In it, we have an expensive calculation, and hope to resume the result without redoing the expensive computation. The table shows the sequence of edit states (excluding movement edits). For each edit state, the number of steps

	`dev` branch			`eval-environment` branch		
	Evaluate	Postprocessing	Equality	Evaluate	Postprocessing	Equality
1	0	0	0	0	1	0
2	0	0	0	0	1	0
3	1	2	0	0	1	0
4	1	1	1	1	0	0
5	1	1	2	0	3	0
6	5	1	3	1	0	0
7	4	5	6	2	2	0
8	3	3	14	0	0	0
9	6	18	33	1	0	1
10	14	29	61	0	0	0
11	13	41	91	3	2	0
12	25	145	203	2	0	1
13	65	578	383	2	0	0
14	147	2399	924	1	3	1
15	226	16597	1603	3	0	1
16				1	0	1
17				2	1	1
18				0	3	1
19				0	0	1
20				3	4	0
21				2	0	1
22				0	2	1
23				0	3	1
24				0	6	1
25				1	4	1
26				1	2	1

Table 7.2: Performance of program illustrated in Listing 3



(a) `dev` branch



(b) `eval-environment` branch

Figure 7.4: Performance of evaluating program in Listing 3

is shown if regular evaluation is shown, as well as the number of steps for FAR if there is a valid fill operation. The difference in evaluation steps between the FAR evaluation and the regular evaluation is shown (Step Δ), as well as the cumulative difference in evaluation steps. For operations with no valid fill operation, the step difference is zero.

We observe that for the operations before the introduction of the heavy computation f 25, there is not a significant difference in the number of steps between normal evaluation and FAR evaluation (when applicable). However, after the three steps after the introduction of this computation are valid FAR operations, and the FAR operation is extremely less expensive. A visualization for this is shown in Figure 7.5. With Figure 7.5a, all edit states after the introduction of the heavy computation also involve the fibonacci calculation. However, in Figure 7.5b, with FAR we see that subsequent calculations roughly only evaluate the filled expression. The cumulative difference in evaluation steps quickly adds up, as evidenced by Table 7.3.

7.3.2 Decreased performance with FAR

Another sample program is shown in Table 7.4. This explores the result of a simpler and perhaps more average program, without an expensive computation. We observe that in this case, the fill operations are typically more expensive than regular operations. This is largely due to the lack of memoization of the re-evaluation of environments in closures, which is described as future work in Section 9.1.2. However, the number of evaluation steps are reasonably on the same order of magnitude as normal evaluation.

We note that we only provide evaluation step counts rather than benchmark times for this discussion of FAR. We do not find that benchmarking evaluation is necessary, as it is more or less unchanged from regular evaluation (except for treatment of closures). It may be useful to benchmark the structural diffing algorithm, but this may be more important once a multi-step FAR is implemented. The current structural diffing with a one-step FAR is a linear pass over the expression tree, which should have performance characteristics similar

to other linear passes such as type-checking or elaboration, which are not a performance concern.

Program	Steps	Steps (w/ FAR)	Step Δ	Cumulative Step Δ

`let f = ... in`
`let a =` ⦇⦈^{1} `in`
⦇⦈^{2}
	7	-	0	0

`let f = ... in`
`let a = f in`
⦇⦈^{2}
	12	21	9	9

`let f = ... in`
`let a = f` ⦇⦈^{3} `in`
⦇⦈^{2}
	17	-	0	9

`let f = ... in`
`let a = f 2 in`
⦇⦈^{2}
	58	69	11	20

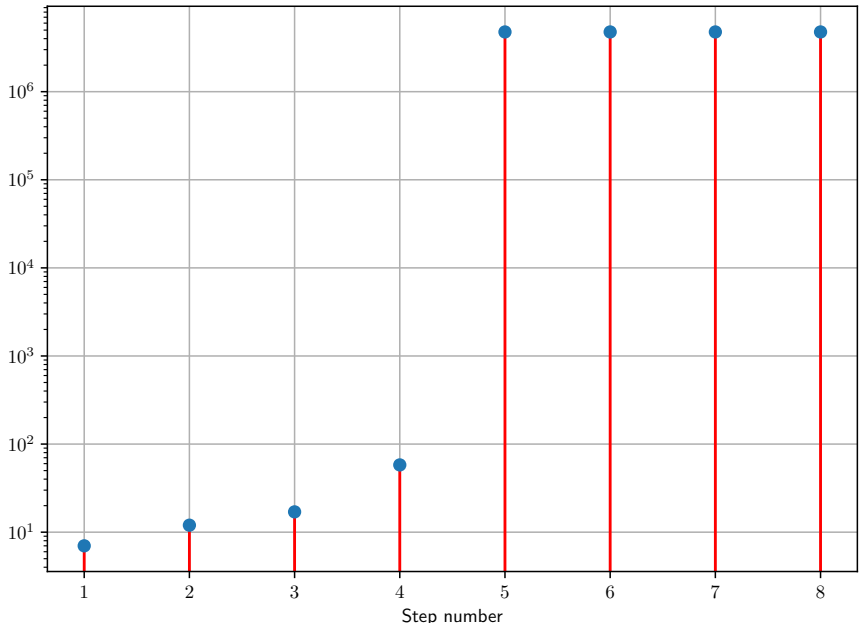
`let f = ... in`
`let a = f 25 in`
⦇⦈^{2}
	4762964	-	0	20

`let f = ... in`
`let a = f 25 in`
⦇⦈^{2} `+` ⦇⦈^{4}
	4762966	12	-4762954	-4762934

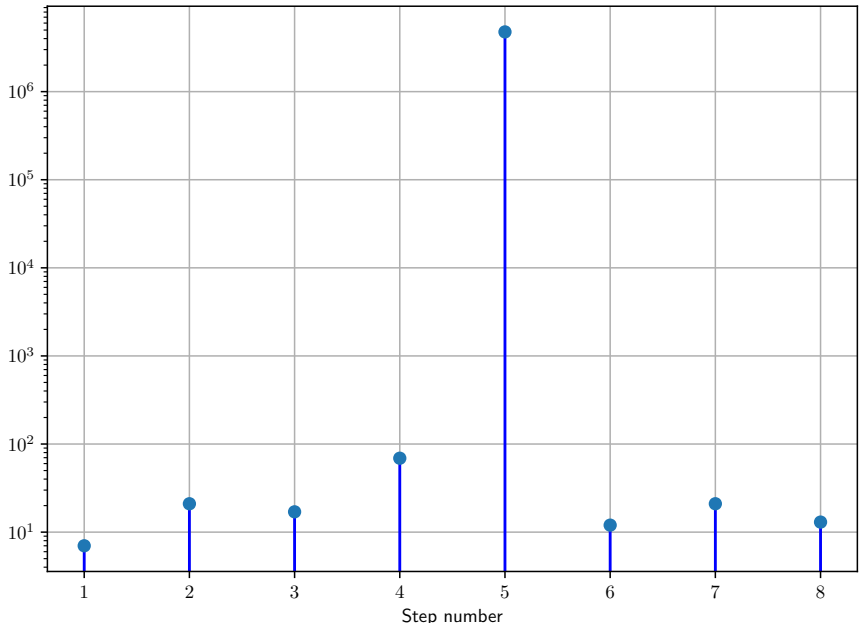
`let f = ... in`
`let a = f 25 in`
⦇⦈^{2} `+ 2`
	4762966	21	-4762954	-9525879

`let f = ... in`
`let a = f 25 in`
`a + 2`
	4792967	13	-4792954	-14288813

Table 7.3: A program edit history with an expensive computation



(a) Normal evaluation



(b) With one-step FAR

Figure 7.5: Number of evaluation steps per edit in Table 7.3

Program	Steps	Steps (w/ FAR)	Step Δ	Cumulative Step Δ
⦇⦈^{1}	1	-	0	0
`let` ⦇⦈^{6} `=` ⦇⦈^{5} `in` ⦇⦈^{7}	2	3	1	1
`let x =` ⦇⦈^{5} `in` ⦇⦈^{7}	3	-	0	1

`let x =` ⦇⦈^{5} `in`
`let` ⦇⦈^{12} `=` ⦇⦈^{11} `in`
⦇⦈^{13}
	4	5	1	2

`let x =` ⦇⦈^{5} `in`
`let y =` ⦇⦈^{11} `in`
⦇⦈^{13}
	5	-	0	2

`let x =` ⦇⦈^{5} `in`
`let y = 4 in`
⦇⦈^{13}
	6	9	3	5

`let x =` ⦇⦈^{5} `in`
`let y = 4 in`
⦇⦈^{13} `*` ⦇⦈^{14}
	8	8	0	5

`let x =` ⦇⦈^{5} `in`
`let y = 4 in`
`x *` ⦇⦈^{14}
	9	14	5	10

`let x =` ⦇⦈^{5} `in`
`let y = 4 in`
`x * y`
	10	11	1	11

`let x = 3 in`
`let y = 4 in`
`x * y`
	11	6	-5	6

Table 7.4: A sample edit history for a simple program

Chapter 8

Discussion of theoretical results

8.1 Expected performance differences between evaluating with substitution versus environments

Section 7.1 discusses the empirical results of a few experiments on the performance of evaluation using the substitution method as originally described and implemented, and evaluation using environments as described in this paper. The result of that section show that the performance gain is highly dependent on the program, and it is not unexpected to see the performance of the substitution model of evaluation to outperform the same program evaluated using the environment model. Thus it is difficult to provide a general result about which model is preferred, strictly for performance reasons. Instead, we only provide the highly-parameterized results in the results of the two variations of the fibonacci program.

A way to remedy this is to collect information about expected or average programs, as described in Section 9.2. The same is true to determine average performance characteristics of FAR, although this would be related to collecting editing behavior, not characteristics of the program itself.

A non-performance reason may tip the scale in favor of either using the substitution model or the evaluation model. As observed throughout this text, the substitution model

is much simpler to formally state and implement, and may be sufficient in most use cases. However, environments allow for memoization. Environments also allow an implementation to be more amenable to lower-level implementations, because it does not require the runtime to keep a representation of the internal language (whereas substitution does). However, it is also important to keep in mind that environments in this implementation are closely tied to an immutable data structure (due to the frequency of copying and the efficiency of structural sharing), which may frustrate low-level implementations.

8.2 Purity of implementation

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.

The evaluation process becomes stateful. Every call to `Evaluator.evaluate` takes an `EvalState.t` as both input and output. This maintains state but is still technically pure, much like a state monad. This state includes the environment identifier generator, the fill memoization context, and evaluation statistics such as evaluation steps. This complicates the formalization and the implementation, as we now have additional inputs and outputs to each evaluation judgment. This is also likely less performant than if some global state were used instead. This global state would have to be reset at the beginning of each evaluation or resumed evaluation.

8.3 FAR for notebook-style editing

One of the primary qualities of computational notebooks is the ability to run sections of the code at a time, primarily for computational purposes. We may reproduce this behavior in a limited way.

We may model a notebook-style program as a linear sequence of sections or cells. Each cell, can be modeled as a set of variable bindings: the "outputs" of evaluating the section. We may interpret a `let` statement in Hazel to be a single-output section.

Consider the sample notebook-style program shown in Figure 8.1a. Here, we have two sections, which compute three different variables. If we add a third section containing the statement `a = x + y;`, then we may obtain the value of a by only executing the third section and not re-evaluating the first two sections, because the workspace is saved.

Now, consider the comparable program in Figure 8.1b. If the user types in the expression `x + y` or `let a = x + y in ⦇⦈1`, then a fill operation is detected, and the new expression is computed with the environment stored in hole 1's closure. In fact, if the user continues to make any changes below the fourth `let` expression, they will all be considered to be valid fill operations against the edit state shown in Figure 8.1b, so the first four statements never have to be re-evaluated. As the user continues to edit downwards, new holes will be created and the newest edits may be considered fill operations from more recent edits.

Additionally, we have a reproducibility benefit here. In most notebook-style editors, such as MATLAB's live editor, running sections out of order may cause a different program output than if the program had been run in order. For example, if the user runs the section section twice after evaluating the first section Figure 8.1a is run twice in a row, then the output would be different than if the program was run from start to finish. On the other hand, the Hazel program in Figure 8.1b will always give the same result as if evaluation had begun from the top, as guaranteed by the commutativity property of FAR.

The limitation of using Hazel as a computational notebook is that the fill-and-resume operation is limited to cases where there is a hole in a previous edit state as a parent

```
%% section 1
x = 5;
y = 4;
%% section 2
z = 3 * x + y;
x = 5 * z;
%% section 3
% a = ...
```

(a) Sample notebook-style program in MATLAB

```
let x = 5 in
let y = 4 in
let z = 3 * x + y in
let x = 5 * z in
()[1]
```

(b) Sample notebook-style program in Hazel

Figure 8.1: Comparison of notebook-style programs in MATLAB and Hazel

of the root of the diff. If a user programs by always appending to or editing near the bottom of their program, FAR will be very beneficial because most edits will lead to valid fill operations. It may be difficult to quantify the real performance benefit due to large variations in programming styles. Different heuristics for choosing a past edit state to compare, as discussed in Section 9.1.3, will also affect the performance of FAR.

8.4 Summary of generalized concepts

The work performed for FAR leads us to the following nice generalizations of some of the concepts we've encountered through this work.

8.4.1 Generalized closures and the evaluation boundary

Generalized closures form an integral part of this implementation. Previously, the term "closure" tends to refer to a function closure, but we find that allowing for a general container expression with a bound environment is extremely useful for our implementation. Not only do they tidy up the implementation by "factoring out" environments from the numerous expression forms that require them[1], but they characterize the evaluation boundary: expressions in the evaluated result that exist within a closure are "paused" evaluations that may be

[1]This is true even in the case of evaluation with substitution, by separating environments from hole closures.

resumed later. Separating holes from closures also helps to facilitate fill-and-resume because we wish to substitute the hole with d_{fill} without discarding the environment.

8.4.2 A generalization of non-empty holes

Non-empty holes play a central hole in Hazel's ability to provide continuous feedback, as well as in the ability to resume computation in FAR. We may interpret an empty hole as encapsulating a well-formed expression in some incompatible outer expression.

It may be helpful to also imagine that the entire program lies in a non-empty hole. In this interpretation, regular program evaluation (from the start) may be considered a degenerate case of fill-and-resume, where the root of the diff is a non-fill diff that gets propagated up to this non-empty hole. This hole will also nicely serve as the parent for all holes in a non-complete program in the `HoleClosureInfo.t`, although the parent of this hole would then not be well-defined.

8.4.3 FAR as a generalization of evaluation

It is possible to express every evaluation operation as a FAR operation, assuming that we have the ability to look back an unlimited number of edit states. The intuition behind this is that the initial state of a program is the empty hole $(\!|\,|\!)^1$. It is trivial to prove that using the rules given by the fill diff that every edit state produces either a no diff or a fill diff judgment against this edit state.

For practical reasons, it may be less efficient to perform a fill diff against an arbitrary number of states, or even to store the entire history of a program. Also, the whole history of a program may not be available. In the case of a parsed program or an example program, which is specified as an edit state rather than a history of edit actions, we would need additional machinery to produce a possible series of edit actions that leads to this state from the empty state.

Chapter 9

Future work

9.1 Improvements to FAR

9.1.1 Finishing the implementation of FAR

The implementation of FAR completed for this thesis is a limited proof-of-concept. The FAR detection mechanism (the structural diffing algorithm) and preprocessing steps are complete. However, due to limited time constraints, the implementation has not achieved parity with the theoretical exploration of FAR explored in this paper.

- A list of past edit states should be made accessible to the FAR detection algorithm. The current structure of the Hazel toplevel and the undo history makes this a little trickier because there may be multiple unrelated histories caused by switching programs ("cards") in the top panel of the Hazel UI.
- The result of FAR should be memoized alongside the results from regular evaluation. The current implementation of normal evaluation is memoized and is not aware of the FAR operation.
- The result of evaluating the program (`Result.t`) is not currently stored in the model, but the program's edit state (`Program.t`) is. This means that when the program result

is needed in the Hazel UI, the program is re-evaluated. This is usually not a problem because of memoization, but FAR results are not memoized. It would be better to store the results of evaluation in the model and undo history to avoid this trouble.

- There is a slight issue with the current description of the `FillExp` variant $(\!|d|\!)_i$. This requires us to have access to the hole closure identifier i, which comes from the postprocessed result. However, we perform the FAR operation on the un-postprocessed result. To remedy this, we may begin evaluation from the post-processed result, which may cause us to change some of our assumptions about the evaluation boundary. Another solution would be to have an alternative postprocessing operation that renumbers holes but leaves alone the evaluation boundary.

9.1.2 Memoization for environments during re-evaluation

We introduced fill expressions $(\!|d|\!)_i$ in Section 6.2.2 in order to memoize the evaluation of filled expressions during re-evaluation.

It will also be beneficial to memoize the re-evaluation of closure environments. To illustrate why this is the case, consider the case of Figure 6.7. In this example, the environment is evaluated twice, even though it is the same physical environment. Each environment will be re-evaluated each time it is encountered, leading to the same exponential blowup problem encountered when dealing with postprocessing in Listing 3. The implementation of this memoization is the same as before; we will need a new environment state variable mapping environment identifiers to evaluated environments.

9.1.3 Choosing the edit state to fill from

There are a number of possible design decisions when searching for a valid hole fill. Firstly, one must decide the maximum number of edit states to search: should it be a fixed number of edit states, or should it be given a fixed time budget? Is it best to cache edit states that

recently led to a fill operation (à la LRU cache)? Is the most recent edit state that leads to a valid fill usually the best candidate, or even a good candidate? Would it be best to allow for user-configurable settings, or perhaps even for the user to manually select the previous edit from which to fill?

It will be useful to collect empirical data about user-editing behaviors, as discussed in Section 9.2, in order to better tune these parameters. Alternatively, we may allow for some user configuration of FAR, as discussed in Section 9.1.4.

9.1.4 User-configurable FAR

The FAR procedure as described is a completely automatic process. However, the choice of which past edit state to choose may be tricky, as described in Section 9.1.3. It may be desirable for the user to manually set a past edit state as an "anchor" from which to set FAR, and thus override an automatic choice of past edit state. This reifies the concept of sections or section breaks from computational notebooks. However, since a user-chosen edit state may not lead to a valid fill diff, edit states that lead to an invalid fill diff will still have to be evaluated from scratch.

It may also be desirable to disable FAR completely, whether for debugging purposes or because it does not help performance much in the given circumstance. This should be a toggleable parameter in the sidebar.

9.1.5 UI changes for notebook-style evaluation

We have motivated the use of Hazel with FAR for notebook-style evaluation in Section 8.3. It may be useful to the programmer to have additional visual aids to help enforce the benefits of resumed evaluation.

It may be useful to format the code editor using a series of sections. These sections will be syntactic sugar for sequences of let statements, and edits to the code in a section will only cause the current and subsequent sections to re-evaluate. This also presents a method

to choose the edit state from which to fill that is intuitive to the user.

Another useful feature to evaluate FAR is to show evaluation statistics in the UI. The two evaluation statistics described in Section 6.5.1 may be useful information to the user. These may inform the user what types of operations are more expensive when FAR is taken into account, and may actually promote a style of editing that maximizes the usage of FAR.

9.2 Collection of editing statistics

The effects of switching evaluation to use environments instead of substitution, and the effect of different methods of choosing a previous edit state for FAR may drastically change the expected performance gain. We may only quantify expected differences in performance when concerned with specific types of programs or editing patterns.

Thus it will be useful to collect empirical statistics about user editing patterns, in order to gain a useful insight into the general expected effect on performance of this work. This information will likely help inform many other Hazel subprojects. Since Hazel is hosted online and is accessible to all, collecting usage statistics is technically very easy. Other online programming language environments, such as the Hedy incremental programming environment for programming education [36, 37], have collected user editing statistics to help direct their work.

9.3 Mechanization of metatheorems and rules

The metatheory of Hazelnut and Hazelnut Live were proved using the Agda proof assistant [11, 12]. A similar proof of the metatheory presented in this work is left to future work. Due to time constraints, we are satisfied with using the intuition presented as justifications for the metatheorems in this thesis.

Chapter 10

Conclusions and recommendations

This thesis explores several advancements to the evaluation model in Hazel, an implementation of a live programming editor with typed holes. We develop rules and a metatheory for the propesd changes as well as a working implementation for most of the rules provided.

The first proposed change is the switch from using variable substitution at binding time to storing variables in an environment at lookup time. This implementation leads to the introduction of closures to the internal language of Hazel, as well as a postprocessing step to restore the same result that would have been achieved by evaluation using the substitution model. Initially, we begin with the conventional function closures and the hole closures introduced in Hazelnut. Later, we introduce generalized closures, which subsume function and hole closures, and represent any paused evaluation. Generalized closures play a critical role in FAR.

The second major change is the use of the environment model to memoize operations that occur on shared environments. Environments are uniquely numbered to allow for lookup and memoization. Memoization is applied to the hole renumbering process (in the process changing the useful interpretation from hole instances to hole closures), and to speed up the structural equality checking. Memoization may also be helpful with the re-evaluation with filled expressions and environments during the FAR re-evaluation process, but this has not

been fully implemented yet.

The third major contribution is a set of practical considerations to implementing the fill-and-resume (FAR) optimization, as originally proposed in Hazelnut Live. FAR promotes resuming evaluation from a previous evaluation result when an edit action is performed, as opposed to restarting evaluation from the beginning on every edit action. Firstly, we describe a structural diffing algorithm for detection of a valid fill operation. This algorithm is intuitive and robust to work between an arbitrary past edit state and the current edit state (a n-step fill operation). We also provide the basic semantics for the fill (preprocessing) and resume (re-evaluation) operations. A 1-step FAR operation is implemented as a proof of concept.

We evaluate the performance of these methods empirically, via a series of benchmarks of sample programs. We compare the difference between the current main development branch on the `dev` branch to an updated evaluation model implementing the changes proposed in this thesis, in the `eval-environment` and `fill-and-resume-backend` branches. The results qualitatively match the expectation. Evaluation with environments is beneficial for performance when lazy variable lookups are reduced and the environment size is small. Substitution may be beneficial for performance when the number of substitution passes is small. The memoization of environments solves the performance issue in the program that motivated the memoization of environments in the hole numbering and structural equality checking processes. FAR provides a great improvement in efficiency when there is a valid fill operation and expensive re-computations can be avoided. However, FAR may sometimes be more expensive than regular evaluation from the beginning, due to the recursive re-evaluation of environments. Future work on this environment memoization and the implementation of the n-step fill operation should further improve the performance benefit of FAR.

We do not prove the correctness of the implementation. We instead provide a metatheory governing the implementation and provide a logical intuition for the correctness of the proposed metatheorems. We leave formal proofs of the metatheory and proofs of the com-

pletenes of the metatheorems to future work.

The primary goal of this work is to inform future development on Hazel or related research efforts, and the explanations and motivations have been written in enough detail to allow for others to independently reproduce this work. The implementation of the rules in this work are intended to act as a reference implementation and not necessarily be incorporated directly into the main development branches; the theoretical discussion is the more useful part of this work. We discuss practical tradeoffs of implementing evaluation with environments. Evaluation using environments leads to some improvement in performance in many programs where lazy variable lookup is beneficial, and leads to a major improvement in performance in some programs due to the effect of memoizing environments, but comes at the expense of a great deal of extra complexity that may obscure Hazel's original goal in approaching the gap problem. We also describe a possible implementation of FAR and entrypoint for the FAR algorithm, with possibilities for further memoizing re-evaluation. The empirical results that are presented may serves as a guideline for performance benefits, but it will be useful to collect user editing and program statistics to better evaluate the average or expected performance benefit of the proposed changes. Along the way, we introduce several novel generalizations that both simplify implementation and give nice theoretical interpretations, such as generalized closures and the presentation of n-step FAR as a generalization of evaluation.

Bibliography

[1] 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.

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

[3] hazelgrove. Hazel `dev` branch live demonstration. `https://hazel.org/build/dev/`, Mar 2022.

[4] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):1–15, 2010.

[5] Eyal Lotem and Yair Chuchem. Lamdu.

[6] Markus Voelter, Daniel Ratiu, Bernhard Schaetz, and Bernd Kolb. mbeddr: an extensible c-based programming language and ide for embedded systems. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 121–140, 2012.

[7] hazelgrove. hazel. `https://github.com/hazelgrove/hazel`, 2022.

[8] Robert Harper. *Practical foundations for programming languages*. Cambridge University Press, 2016.

[9] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[10] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda–a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*, pages 73–78. Springer, 2009.

[11] hazelgrove. agda-popl17. `https://github.com/hazelgrove/agda-popl17`, 2017.

[12] hazelgrove. hazelnut-dynamics-agda. `https://github.com/hazelgrove/hazelnut-dynamics-agda`, 2019.

[13] Jana Dunfield and Neel Krishnaswami. Bidirectional typing. *ACM Computing Surveys*, 54(5):1–38, jun 2022.

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

[15] Benjamin C Pierce and David N Turner. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):1–44, 2000.

[16] Adam Michael Gundry. *Type inference, Haskell and dependent types*. PhD thesis, University of Strathclyde, 2013.

[17] Gordon D Plotkin. *A structural approach to operational semantics*. Aarhus university, 1981.

[18] Gilles Kahn. Natural semantics. In *STACS*, 1987.

[19] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *IN SCHEME AND FUNCTIONAL PROGRAMMING WORKSHOP*, pages 81–92, 2006.

[20] Jeremy G Siek, Michael M Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[21] John Hughes. Why functional programming matters. *The computer journal*, 32(2):98–107, 1989.

[22] Jason Hickey, Anil Madhavapeddy, and Yaron Minsky. Real world ocaml. 2014.

[23] Robert Harper and Christopher A Stone. A type-theoretic interpretation of Standard ML. In *Proof, Language, and Interaction*, pages 341–388, 2000.

[24] Peter Sestoft. Runtime code generation with JVM and CLR. *Available at* ***http://www.dina.dk/sestoft/publications.html***, 2002.

[25] hazelgrove. hazelc branch in the hazel repository. `https://github.com/hazelgrove/hazel/tree/hazelc`, 2022.

[26] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical computer science*, 1(2):125–159, 1975.

[27] Philip Miller, John Pane, Glenn Meter, and Scott Vorthmann. Evolution of novice programming environments: The structure editors of carnegie mellon university. *Interactive Learning Environments*, 4(2):140–158, 1994.

[28] Hannah Potter and Cyrus Omar. Hazel tutor: Guiding novices through type-driven development strategies. *Human Aspects of Types and Reasoning Assistants. https://hazel.org/hazeltutorhatra2020. pdf*, 2020.

[29] Thomas RG Green. Cognitive dimensions of notations. *People and computers V*, pages 443–460, 1989.

[30] hazelgrove. tylr. `https://github.com/hazelgrove/tylr`, 2021.

[31] Sebastian Burckhardt, Manuel Fahndrich, Peli de Halleux, Sean McDirmid, Michal Moskal, Nikolai Tillmann, and Jun Kato. It's alive! continuous feedback in ui programming. *SIGPLAN Not.*, 48(6):95–104, jun 2013.

[32] Fernando Pérez and Brian E Granger. Ipython: a system for interactive scientific computing. *Computing in science & engineering*, 9(3):21–29, 2007.

[33] Sam Lau, Ian Drosos, Julia M Markel, and Philip J Guo. The design space of computational notebooks: An analysis of 60 systems in academia and industry. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–11. IEEE, 2020.

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

[35] Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. Filling typed holes with live GUIs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 511–525, 2021.

[36] Felienne Hermans. Hedy: a gradual language for programming education. In *Proceedings of the 2020 ACM conference on international computing education research*, pages 259–270, 2020.

[37] Marleen Gilsing and Felienne Hermans. Gradual programming in Hedy: A first user study. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–9. IEEE, 2021.

Appendix A

Additional contributions to Hazel

A.1 Additional performance improvements

A.2 Documentation and learning efforts

Appendix B

Code correspondence

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

Appendix 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

Appendix D

Selected code samples