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

TODO

ABSTRACT

TODO

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

LIST OF THEOREMS

TABLE OF NOMENCLATURE

τ	Hazel type
Γ	Typing context
Δ	Hole context

Table 1: Hazel expression and hole typing

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

Table 2: Hazel internal language

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

Table 3: Hazel evaluation and postprocessing judgments

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

Table 4: Hazel postprocessing

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 [**TODO: need reference(s): structural editors**] have been proposed to improve the programming experience and introduce editor services, such as the elimination of syntax errors or graphical editing.

Hazel [2] 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 [3], which governs the structural editor and the static semantics (typing rules) of the language. The dynamic semantics (evaluation semantics) are described in [4].

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 the standard conversion from evaluation using

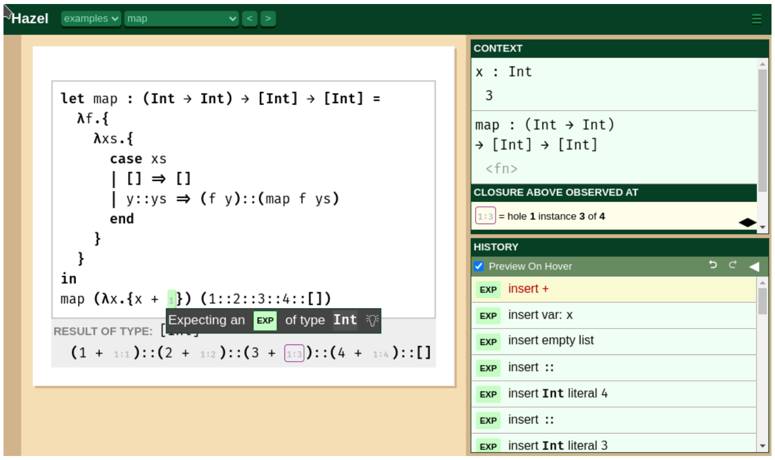

Figure 1.1: A screenshot of the Hazel live programming environment. Screenshot taken of the dev branch demo on 02/06/2022 [1].

the substitution model (simpler to reason about) to the environment model (more performant) [**TODO: need reference(s): standard conversion**], 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 related to Hazel's structure (such as hole closure numbering), and also implement the fill-and-resume performance enhancement described in [4]. The novelty of this work lies in the novelty and optimization opportunity of Hazel's hole-based static and dynamic semantics.

1.2 The contribution of this work

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

- The evaluation semantics of the Hazel language using the environment model (which replaces the substitution model as implemented on the trunk branch and described in [4]). While most of this is standard, we aim to keep the implementation pure (which is less trivial in the case of recursion), introduce uniquely-numbered environments (for later use in memoization), and describe the evaluation of holes (which are unique to Hazel).

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

- Fill-and-resume, as originally proposed in [4]. This algorithm is described at a high level in the original description and not yet implemented until this thesis work. We provide the implementation and a lower-level description of said implementation.

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

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

The performance of this work is measured primarily in terms of empirical performance gains (via evaluation-step counting and benchmarking), and discussed with respect to the theoretical performance. This proof of correctness of the algorithms was not mechanized in the Agda proof assistant as was much of the core of Hazelnut [3] and Hazelnut Live [4]. Instead, correctness of implementation is validated by standard software testing procedures with manual test cases, and 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 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 assessment of the work completed in terms of correctness and the theoretical performance. Chapter 8 is a discussion of future research directions that may be spawned off from this work. Chapter 9 concludes with a summary of findings and future work. The Appendices contain additional information about the Hazel project not directly related to the primary contribution of this project, as well as selected source code snippets.

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, which Hazel is heavily based on. In this section, the syntax and static semantics of the λ-calculus are explored; even though not directly tied to the work performed in this, 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 [**?**]. Section 2.3 provides some detail on different types of programming language implementations, which is standard material in an introductory compilers text such as [5]. 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 a large part of this thesis work. Section 2.5 provides an overview of structural editors. Finally, Section 2.6 provides some background on contextual modal type theory (CMTT), which forms the conceptual precedent for the hole-filling operation.

2.1 Specifications of programming languages

To be able to rigorously work with programming languages, as like any mathematical activity, we need to be able to precisely define the behavior of programming languages that serve as our interface to computation. We typically define the definition (or specification) of a programming language 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 single programming language may have several implementations of the specification, which may have differing support for language features and different performance characteristics. Common classifications of programming language implementations are discussed in Section 2.4.

2.1.1 Syntax

The syntax of a programming language is defined by a grammar. The grammar described in the original paper on Hazelnut [3] is reproduced below as an example.

$$\begin{aligned} \tau &::= \tau \to \tau \mid \texttt{num} \mid (\!|\,|\!) \\ e &::= x \mid \lambda x.e \mid e\ e \mid e + e \mid e : \tau \mid (\!|\,|\!) \mid (\!|e|\!) \end{aligned}$$

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 (function literal); the primitive addition operation; a type ascription; an empty hole; or a non-empty hole. The latter two types are Hazel specific, as is the hole type.

Parentheses are not shown in this grammar, but they are optional except to affect order of operations.

Parts of this grammar will be revisited when discussing the λ-calculus described in Section 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 2.5.1), 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 interpret the rule as: "if all of $p_1, p_2, \ldots, p_n$ are true, then q must be true" (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; in our case the formal specification of the static and dynamic semantics of a programming language. 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.

A derivation of a statement in such a system by chaining together inference rules, such that the final consequent is the statement to be proved.

[TODO: example of a derivation in a simple fabricated system]

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 additional metatheorems, which must be proved for all of the inference rules in the logic system. In the foundational papers for Hazel's core semantics [3, 4], metatheorems are amply used to justify and verify the correctness of the rules. Agda [6], an interactive proof checker and dependently-typed programming language, is used for these proofs [7, 8].

[TODO: example of a metatheorem (from the original papers?)]

2.1.3 Static semantics

The *static semantics* of a programming language describes specifications of a language that occur prior to program evaluation. Static semantics typically primarily refers 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 2.4. 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.

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

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

There are a few noteworthy items here. The syntax $\Gamma, x : \tau$ indicates that 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* [9, 10, 11], or *local type inference*. Bidirectional typing involves two typing judgments: the *typing synthesis judgment* $\Gamma \vdash e \Rightarrow \tau$ (pronounced "given typing context Γ, expression e synthesizes type τ), and the *type analysis judgment* $\Gamma \vdash e \Leftarrow \tau$ (pronounced "given typing context Γ, expression e analyzes against type τ). The type synthesis judgment outputs a type (the exact or "narrowest" type of the expression), whereas the type analysis judgment takes a type as an input and "checks" the expression against that ("wider") type. With these two judgments, we may be able to loosen the antecedant judgments when synthesizing a type. We may re-express the above type system into a similar (and incomplete) 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 [9], who provides an overview of bidirectional typing, or to the formulation of Hazel's bidirectional typing [3]. Hazelnut is at its core a bidirectionally-typed "edit calculus" [3], 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. 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 [12], 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 language expression to a *value*, an irreducible expression.

Let us consider values in some more detail. Values are distinguished by the judgment $v \Downarrow v$; i.e., values evaluate to themselves. Following the notation from [3, 4], we can alternatively write this using the equivalent judgment $v\ \mathsf{value}$. 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 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 [13] (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, and $\to$ is the operation being defined.

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 [14] (alternatively, *big-step semantics*). In the big-step semantics, the evaluation judgment is $e \Downarrow v$, where e is an expression in the language, $\Downarrow$ is the evaluation operator, 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.

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

The implementation of an evaluator with a program stepper capability (as is commonly found in programming language debuggers) 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 [4]. 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 it is a stereotypical functional language. One must also have some knowledge of the *gradually-typed λ-calculus* (GTLC) introduced by Siek [15, 16]. 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 [17], but a brief overview will be provided here.

2.2.1 Introduction to functional programming

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

Functional languages are based on Alonzo Church's λ calculus [**TODO: need reference(s): lambda calculus**] 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 languages. This is as opposed to in imperative programming, in which the mutable "memory cell" interpretation of variables and side-effects complicates formalizations.

Hazel is one such (purely) functional programming languages. Other languages that are classified as functional include the ML family of languages, Haskell, Elm, and the LISP family of languages. Examples of imperative programming languages include C, C++, FORTRAN, Java, and Golang. A number of languages incorporate both functional and imperative styles, such as Javascript, Python, Scala, and Rust [**TODO: need reference(s): all of these languages and their classifications**].

[**TODO: show a simple example of programs in these paradigms?**]

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[1]), shown below.

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

The static semantics of this syntax are very simple: every expression in Λ is well-fromed if all variables are bound by some binder[2].

[1] 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, but that is not the interpretation here.

[2] There are no typing rules in the static semantics, because there is only a single type: the recursive arrow

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 [3].

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

Λ is an example of a Turing-complete language. One of the key characteristics to this is the ability to compute 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 variables other than function binders (i.e., there is no construct such as OCaml's `let rec` expressions), one must pass a self-reference of a function to itself. For example, let us consider the example of a factorial function in Λ (for sake of illustration, extended with a conditional statement, integers, and simple integer operators).

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

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

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 [17]. Thus no type errors will occur when evaluating a (well-formed) expression in Λ.

[3] The substitution of the function variable during function application is known as β-reduction. Renaming of bound variables (a process known as α-conversion) is used to avoid substituting variables of the same name bound by a different binder.

programming language theory, such as [17].

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[4].

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.

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

The grammar is extended to include constants of the base type. The type of functions parameters must be annotated[5].

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.

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

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

[4] The stereotypical example of representing data using functions is 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.

[5] 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.

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 remains the same.

$$\frac{}{c \Downarrow c}\ \Lambda_{\rightarrow}\text{-EConst} \qquad \frac{}{\lambda x : \tau.e \Downarrow \lambda x : \tau.e}\ \Lambda_{\rightarrow}\text{-ELam} \qquad \frac{e_1 \Downarrow \lambda x : \tau.e}{e_1\ e_2 \Downarrow [e_2/x]e}\ \Lambda_{\rightarrow}\text{-EAp}$$

$$\frac{}{\text{fix}\, f : \tau.e \Downarrow [\text{fix}\, f : \tau.e/f]e}\ \Lambda_{\rightarrow}\text{-EFix}$$

We may characterize type systems by establishing certain desirable properties. One such property is *soundness*. Soundness means that if a program in $\Lambda_{\rightarrow}$ 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 $\tau ::= \tau \rightarrow \tau$.

There is an additional expression form in $\Lambda_{\rightarrow}$. 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. In this example, we assume that the base type $b \equiv \text{int}$, and that conditionals and primitive integer operations extend $\Lambda_{\rightarrow}$.

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

The fixpoint operator is introduced in Plotkin's System PCF [**TODO: need reference(s): can cite PFPL, also get citation for original pcf?**], and is used to implement recursion in Hazel's evaluator, which uses a substitution-based evaluation.

$\Lambda_{\rightarrow}$ is a practical foundation for many functional languages. Standard exercises include extending $\Lambda_{\rightarrow}$ with multiple common base types (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.

2.2.4 The gradually-typed λ-calculus

[TODO: references for notation: siek for original gradual typing (unknown type, type consistency, cast calculus), hazelnut notation for most notation specifics, matched arrow notation from other sources from hazelnut paper]

We have discussed $\Lambda_\rightarrow$, 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[6], and thus more quickly prototype or refactor programs.

The hybrid proposed by Siek is the *gradually-typed λ-calculus* $\Lambda^?_\rightarrow$ [15, 16]. In $\Lambda^?_\rightarrow$, all type annotations are optional and offer a "pay-as-you-go" benefit. A completely unannotated $\Lambda^?_\rightarrow$ 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 $\Lambda^?_\rightarrow$ program is equivalent to a $\Lambda_\rightarrow$ 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 $\Lambda^?_\rightarrow$ is almost exactly the same as $\Lambda_\rightarrow$, except that we add a new type ?, indicating an unspecified type. Now, λ-abstractions may be type-annotated using this type, and we define the notation $\lambda x.e \equiv \lambda x :?.e$.

The static semantics of $\Lambda^?_\rightarrow$ is expectedly also similar to $\Lambda_\rightarrow$. The only rule that differs is the rule for function application. We also write a new rule for subsumption, which states

[6]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.

that if $\Gamma \vdash e : \tau$, then e may also be assigned any consistent type.

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

Two new judgments are introduced here. The first is the *matched arrow judgment* $\tau_1 \blacktriangleright_{\rightarrow} \tau_2 \rightarrow \tau_3$, 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.

$$\frac{}{(\!|\,|\!) \blacktriangleright_{\rightarrow} (\!|\,|\!) \rightarrow (\!|\,|\!)}\ \text{MAHole} \qquad \frac{}{\tau_1 \rightarrow \tau_2 \blacktriangleright_{\rightarrow} \tau_1 \rightarrow \tau_2}\ \text{MAArr}$$

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[7].

$$\frac{}{(\!|\,|\!) \sim \tau}\ \text{TCHoleTyp} \qquad \frac{}{\tau \sim (\!|\,|\!)}\ \text{TCTypHole} \qquad \frac{\tau_1 \sim \tau_1' \qquad \tau_2 \sim \tau_2'}{(\tau_1 \rightarrow \tau_2) \sim (\tau_1' \rightarrow \tau_2)}\ \text{TCArr}$$

[TODO: explain elaboration to the internal cast calculus]

[TODO: explain the dynamic semantics]

[7]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's original formulation of $\Lambda^{?}_{\rightarrow}$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 [15].

2.3 Implementations of programming languages

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

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

2.4 Compiler vs. interpreter implementations

There are two general classes of programming language implementations: *interpreters* and *compilers* [5]. 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) [5]. The bytecode outputted from the compilation process is used for evaluation. Compiled implementations tend to produce better runtime efficiency, since the compilation steps are performed separate of the evaluation, and because there is little to no runtime overhead.

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

In certain contexts (especially in the ML spheres), the term *elaboration* [18] is used to the process of transforming the *external language* (a well-formed, textual program) into the *internal language* (IR). The interior language may include additional information not present in the external language, such as types generated by type inference or bidirectional typing.

The distinction between compiled and interpreted languages is not a very clear line: some implementations feature just-in-time (JIT) compilation that allow "on-the-fly" compilation (e.g., common implementations of the JVM and CLR [19]), 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 [20], which is outside the scope of this project.

2.4.1 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 never evaluated directly; they are substituted by their values when bound, and their values are evaluated. The substitution model is useful for teaching purposes because it is simple and close to its mathematical definition: a variable can be thought of as an equivalent stand-in for its value.

However, for the purpose of computational efficiency, a model in which values are lazily expanded ("looked-up") only when needed is more efficient. This is called the *environment model of evaluation*, and generally is more efficient because the runtime does not need to perform an extra substitution pass over subexpressions 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 substitution model does not imply a lazy (i.e., normal-order, call-by-name, call-by-need) evaluation [21] as in languages such as Haskell or Miranda, in which bound variables are (by default) not evaluated until their value is required. Laziness is conceptually tied to substitution, but the substitution model does not require laziness. Like most programming languages, Hazel only has strict (i.e., applicative-order, call-by-value) evaluation: the expressions bound to variables are evaluated at the time of binding.

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

2.4.2 Closure conversion

[TODO: implement this section]

2.5 Approaches to programming interfaces

[TODO: need to fill out this section]

2.5.1 Structure editors

2.5.2 Graphical editors

2.5.3 Applications to programming education

2.5.4 Criticisms of non-textual editors

2.6 Contextual Modal Type Theory

[TODO: implemen this section; background for hole filling]

Chapter 3

An overview of the Hazel programming environment

Hazel is the experimental language that implements the Hazelnut bidirectionally-typed edit static semantics with holes and the Hazelnut Live dynamic semantics, and it is also the name of the reference implementation. It is intended to serve as a proof-of-concept of the semantics with holes that attempt to mitigate the gap problem; however, the implementation is becoming increasingly practical with additional research efforts. The reference implementation is an interpreter written in OCaml and transpiled to Javascript using the `js_of_ocaml` (JSOO) library [22] so that it may be run client-side in the browser. A screenshot of the reference implementation is shown in Figure 1.1 [1]. The source code may be found on GitHub [2].

Hazel's syntax and semantics resembles languages in the ML (Meta Language) family of languages [23] such as OCaml or SML/NJ, although Hazel does not support polymorphism at this time. Hazel can be characterized as a purely functional, statically-typed, bidirectionally-typed, strict-order evaluation, structured editor language. Hazel semantically differs most significantly from other ML languages in the last respect due to its theoretic foundations in solving the gap problem.

3.1 Introduction to OCaml and Reason

Previously, we have been introducing concepts using a pseudo-mathematical notation. Henceforth, 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.

[**TODO: describe syntax of OCaml/Reason, e.g., modules, types**]

3.2 Hazelnut static semantics

3.2.1 Expression and type holes

3.2.2 Bidirectional typing

3.2.3 Example of bidirectional type derivation

3.3 Hazelnut Live dynamic semantics

[**TODO: fit this description into the rest of the sections**]

The internal language is highly similar to the external language. The primary difference is the introduction of the cast calculus taken from the GTLC described in Section 2.2.4. Practically, there is also the introduction of the fixpoint to allow for recursion, as the fixpoint is not exposed to users in the external language.

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 elaboration algorithm is bidirectionally-typed, and thus involves two mutually-recursive judgments: a *synthetic elaboration judgment* $\Gamma \vdash e \Rightarrow \tau \rightsquigarrow d : \tau \dashv \Delta$, and an *analytic elaboration judgment* $\Gamma \vdash e \Leftarrow \tau \rightsquigarrow d : \tau' \dashv \Delta$. Δ is the *hole context*, used to store the typing context and actual type of each hole, and is an output of the judgments. As with bidirectional typing, the synthetic judgment outputs the synthesized type alongside the elaborated internal expression d; the analyzed judgment takes a type τ to analyze against, and returns the elaborated internal expression d along with its actual type τ'. Having the actual type of analyzed expressions is useful for generating dynamic casts as described in the GTLC.

3.3.1 Example of elaboration

3.3.2 Example of evaluation

3.3.3 Example of hole instance numbering

3.4 Hazel programming environment

3.4.1 Explanation of interface

3.4.2 Implications of Hazel

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

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

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

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

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

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

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

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

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

4.1.1 Evaluation rules

Omar et al. [4] 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 [4]. 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 [4]. 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[2].

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

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

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 $\text{id}(\Gamma)$, 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 inifinitely recursive substitution will occur.

Recursion with the environment model also requires self-reference, but this can be achieved in two ways: by accounting for the fixpoint form, or by using self-referential data structures. In OCaml, self-referential (mutually recursive) data can be achieved using the `let rec` keyword or by using `ref`s (mutable data cells); however, the latter will affect the purity of the implementation, as discussed in Section 4.5.1.

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[3]. 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[4]. 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[5]. In the more general case of mutual recursion, this

[3] 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.

[4] 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).

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

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

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.

[5] §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

would likely make implementation very tricky, and it would be more practical to use impure `ref`s to achieve self-reference. However, for the simple case of a simply recursive function, we may recognize `let`-bindings which introduce a function, and statically construct the mutual recursion using the rule shown in Figure 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[6]. The second problem is solved by the fast equal-

[6] 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.

ity checker for memoized environments described in Section 5.5.3, which is useful even for non-recursive environments. We may also say that using recursive data structures without mutable `ref`s is limited by the language limitations, necessitating workarounds even for the simply-recursive case, and potentially much more complicated workarounds for the mutual recursion case.

The performance improvement is described in 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 expressions for which evaluation has reached ("inside" the boundary), and for those that remain unevaluated ("outside" the boundary). This definition will be useful for describing the 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[7]. This is not captured in the original description of Hazelnut Live [4] 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 [4] 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

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

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

(a) Non-generalized closures

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

(b) Generalized closures

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

and evaluate to themselves.

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

4.2.2 Generalization of existing hole types

Consider the abbreviated definition of the internal expression variant type in Figure 4.3. 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 [4]:

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

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

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

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

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

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

4.2.3 Alternative strategies for evaluating past the evaluation boundary

Without generalized closures, unevaluated expressions (body expressions of λ abstractions, unmatched `let` expressions, and unmatched `case` expressions) may be filled by a modified form of evaluation, which is only different in that a failed lookup (due to unmatched variables) will leave the variable unchanged[8]. However, this is essentially the same as substitution, and is expensive to do during evaluation. Also, while this speculative execution would be reasonable for `let` expressions, it would be highly undesirable for `case` expression, where it is easy to imagine an example where speculative execution leads to infinite recursion.

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

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

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

The symbol chosen to denote postprocessing is $\Uparrow_{[]}$. The choice of symbol is somewhat arbitrary, but we may read it as "reverting" some expressions generated by and useful for evaluation (i.e., closures) to a more context-inspector-friendly form, which is in some sense

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

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 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 $\Uparrow_{[],2}$ 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 $\sigma = (\mathrm{id}_\sigma, x \mapsto d)$, in which id_σ indicates a unique environment identifier.

To ensure that there is a bijection between environment identifiers and environments,

[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.

[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`.

TODO: this needs to be updated/corrected

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

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

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

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

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

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

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

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

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

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

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

TODO: closure needs to go recursive

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

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 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 $i_\sigma(\sigma_1) \neq i_\sigma(\sigma_2)$. $\square$

4.4.2 Modifications to the post-processing rules

During substitution postprocessing ($\Uparrow_{[]}$), a mapping $\mathrm{id}_\sigma \mapsto \sigma$ stores the set of substituted (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.

[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 4.5.1.

TODO TODO	
TODO	

Figure 4.5: Big-step semantics modifications for environment memoization

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 Purity

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

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

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

4.5.2 Data structures

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

4.5.3 Additional constraints due to hole closure numbering

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

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

To summarize, the current state of the implementation involves environments with unique identifiers so as to be more amenable to memoization, but the memoization during postprocessing is only performed for hole closures (i.e., the postprocessing will only not be repeated if the same hole number and environment are encountered multiple times, but will be repeated if the same environment occurs in different holes or in non-hole closures). For the sake of time, fully memoizing all environments and investigating the effects is left for future work, although the marginal benefit may not be very great[13].

4.5.4 Storing evaluation results versus internal expressions

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

The decision should be made whether to store this final judgment in the environment[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 5.2 is solved by memoizing hole environments, which has a clear benefit.

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

[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

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

5.1 Rationale behind hole instances and unique hole closures

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

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

Note that the two instances of $(\!|\!)^2$ have different environments, adn we thus distinguish

```
let a = (|)^1 in
let b = λ x . { (|)^2 } in
f 3 + f 4
```

Listing 1: Illustration of hole instances 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.

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.

[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

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

Listing 3: A seemingly innocuous Hazel program

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

5.2 Issues with the current implementation

Consider the program shown in Listing 3.

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

5.3 Hole instances and closures

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 5.1: Big-step semantics for the previous hole instance numbering algorithm

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

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

$\boxed{TODO}$ TODO

$$TODO$$

Figure 5.2: Big-step semantics for hole closure numbering

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

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

Figure 5.3: Big-step semantics for post-processing

5.3.1 Hole instance path versus hole closure parents

5.4 Algorithmic concerns and a two-stage approach

5.5 Memoization and unification with closure post-processing

5.5.1 Modifications to the instance numbering rules

5.5.2 Unification with closure post-processing

5.5.3 Fast evaluation result structural equality checking

5.6 Differences in the hole instance numbering

Chapter 6

Implementation of fill-and-resume

6.1 CMTT interpretation of fill-and-resume

6.2 Memoization of recent actions

6.3 UI changes for notebook-like editing

Chapter 7

Evaluation of methods

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

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

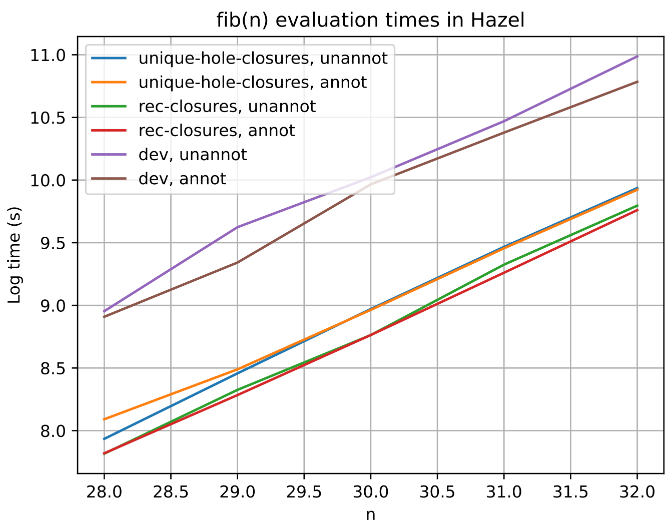

Figure 7.1: Performance of the different models of evaluation

Chapter 8

Future work

8.1 Mechanization of metatheorems and rules

8.2 FAR for all edits

8.3 Stateless and efficient notebook environment

Chapter 9

Conclusions and recommendations

Bibliography

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

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

[3] 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.

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

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

[6] 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.

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

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

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

[10] 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.

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

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

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

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

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

[16] 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.

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

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

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

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

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

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

[23] David MacQueen, Robert Harper, and John Reppy. The history of Standard ML. *Proceedings of the ACM on Programming Languages*, 4(HOPL):1–100, 2020.

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