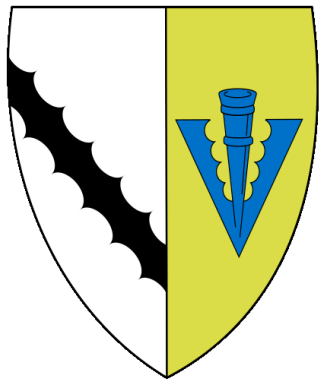
Max Carroll

Type Error Debugging in Hazel

Computer Science Tripos, Part II



Sidney Sussex College College
University of Cambridge
December 9, 2024

A dissertation submitted to the University of Cambridge in partial fulfilment for a Bachelor of Arts

Declaration of Originality

Declaration Here.

Proforma

Candidate Number:	**Candidate Number Here**
College:	**Sidney Sussex College**
Project Title:	**Type Error Debugging in Hazel**
Examination:	**Computer Science Tripos, Part II – 05/2025**
Word Count:	**5354** [1]
Code Line Count:	**Code Count** [2]
Project Originator:	**The Candidate**
Supervisors:	**Patrick Ferris, Anil Madhavapeddy**

Original Aims of the Project

Aims Here. Concise summary of proposal description.

Work Completed

Work completed by deadline.

[1] *Calculated by* `texcount`. *Including: tables and footnotes. Excluding: the front matter, bibliography, and appendices*

[2] *Calculation method here*

Special Difficulties

Any Special Difficulties encountered

Acknowledgements

Acknowledgements Here.

Contents

Chapter 1

Introduction

Brief overview here

1.1 Motivation

Include motivating examples. Give a brief overview of how type errors would be debugged with the tools produced by this project.

1.2 Previous Work

1.3 Contributions

Brief overview of the project contributions.

1.4 Dissertation Outline

Explain structure – what is covered in each chapter briefly.

Chapter 2

Preparation

In this chapter I present the core semantics and a larger overview of the Hazel language.

2.1 The Hazel Language

What is Hazel and who is developing it here.

2.1.1 Overview & Vision

Detail the vision of the Hazel project and main features of Hazel.

Finish with the state of the subset of Hazel for which the project was implemented.

2.1.2 Core Hazel: Formal Semantics

For reference, the established semantics and type system for Hazel is presented. Derived from Omar et al. [1]. The paper

itself goes into deeper depth into the intuition of the rules and the formal properties satisfied by the calculus. [1]

Syntax

The syntax, in Fig. 2.1, consists of *types* τ, *external expressions* e, and *internal expressions* d. Here, ? is the *dynamic type*, $(\!|e|\!)$ is a *non-empty hole* containing e, and $\langle \tau_1 \Rightarrow \tau_2 \rangle$, $\langle \tau_1 \Rightarrow ? \not\Rightarrow \tau_2 \rangle$ are casts and cast errors from τ_1 to τ_2 respectively. The *external language* is a locally inferred [2] surface syntax for the language, and is statically elaborated to (explicitly typed) *internal expressions*, in a similar way to Harper and Stone's [3] approach to defining Standard ML as elaboration to an explicitly typed internal langauge, *XML* [4].

$$
\begin{aligned}
\tau &::= b \mid \tau \to \tau \mid ? \\
e &::= c \mid x \mid \lambda x : \tau . e \mid \lambda x . e \mid e(e) \mid (\!|\,|\!)^u \mid (\!|e|\!)^u \mid e : \tau \\
d &::= c \mid x \mid \lambda x : \tau d \mid d(d) \mid (\!|\,|\!)^u_\sigma \mid (\!|d|\!)^u_\sigma \mid d\langle \tau \Rightarrow \tau \rangle \mid d\langle \tau \Rightarrow ? \not\Rightarrow \tau \rangle
\end{aligned}
$$

Figure 2.1: Syntax: *types* τ, *external expressions* e, *internal expressions* d. With x ranging over variables, u over hole names, σ over $x \to d$ *internal language* substitutions/environments, b over base types and c over constants.

External Language: Type System

The static semantics in Fig. 2.2 of the *external language* is a bidirectionally typed system in the style of Pierce and Turner [2], and Dunfield and Krishnaswami [5]. There are two typing

[1] **Come up with and detail some good intuitions of what each type of value & expression is and what the cast calculus and elaboration does.**

judgement modes: $\Gamma \vdash e \Rightarrow \tau$ which synthesises a type τ, algorithmically thought of as an output, and $\Gamma \vdash e \Leftarrow \tau$ which analyses against a type τ as an input.

$\boxed{\Gamma \vdash e \Rightarrow \tau}$ e synthesises type τ under context Γ

$$\text{SConst}\frac{}{\Gamma \vdash c \Rightarrow b} \qquad \text{SVar}\frac{x : \tau \in \Gamma}{\Gamma \vdash x \Rightarrow \tau} \qquad \text{SFun}\frac{\Gamma, x : \tau_1 \vdash e \Rightarrow \tau_2}{\Gamma \vdash \lambda x : \tau_1 . e \Rightarrow \tau_1 \to \tau_2}$$

$$\text{SApp}\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \qquad \tau_1 \blacktriangleright_{\to} \tau_2 \to \tau \qquad \Gamma \vdash e_2 \Leftarrow \tau_2}{\Gamma \vdash e_1(e_2) \Rightarrow \tau} \qquad \text{SEHole}\frac{}{\Gamma \vdash (\!|\,|\!)^u \Rightarrow ?}$$

$$\text{SNEHole}\frac{\Gamma \vdash e \Rightarrow \tau}{\Gamma \vdash (\!|e|\!)^u \Rightarrow ?} \qquad \text{SAsc}\frac{\Gamma \vdash e \Leftarrow \tau}{\Gamma \vdash e : \tau \Rightarrow \tau}$$

$\boxed{\Gamma \vdash e \Leftarrow \tau}$ e analyses against type τ under context Γ

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

Figure 2.2: Bidirectional typing judgements for *external expressions*

These rules use a type consistency relation, $\sim$ in Fig. 2.3, with types being consistent if they are equivalent up to the locations of the dynamic type. The type consistency relation is standard in gradual type systems [6], [7], and is similar to a subtyping relation but is *not* transitive.

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

$$\text{TCDyn1}\frac{}{? \sim \tau} \qquad \text{TCDyn2}\frac{}{\tau \sim ?} \qquad \text{TCRfl}\frac{}{\tau \sim \tau} \qquad \text{TCFun}\frac{\tau_1 \sim \tau_1' \qquad \tau_2 \sim \tau_2'}{\tau_1 \to \tau_2 \sim \tau_1' \to \tau_2'}$$

Figure 2.3: Type consistency

Finally, a (function) type matching relation, $\blacktriangleright_{\to}$ in Fig. 2.4, matches the argument and return types from a function type, which for the dynamic type is $? \blacktriangleright_{\to} ? \to ?$.

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

$$\text{MADyn}\ \frac{}{? \blacktriangleright_{\to} ? \to ?} \qquad \text{MAFun}\ \frac{}{\tau_1 \to \tau_2 \blacktriangleright_{\to} \tau_1 \to \tau_2}$$

Figure 2.4: Type Matching

Elaboration

Elaboration to the *internal language* is possible for well-typed *external expressions* and consists of cast insertion, maintaining a hole context, and inserting initial identity hole environments. Each of these are used in the internal language type assignment $\Delta; \Gamma \vdash e : \tau$ and the dynamic semantics. Fig. 2.5 defines the elaboration judgements and Fig. 2.6 defines the internal language type assignment categorical judgement [8].

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

$$\textsc{ESConst}\ \frac{}{\Gamma \vdash c \Rightarrow b \rightsquigarrow c \dashv \emptyset} \qquad \textsc{ESVar}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x \Rightarrow \tau \rightsquigarrow x \dashv \emptyset}$$

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

$$\textsc{ESApp}\ \frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \quad \tau_1 \blacktriangleright_{\to} \tau_2 \to \tau \quad \Gamma \vdash e_1 \Leftarrow \tau_2 \rightsquigarrow d_1 : \tau_1' \dashv \Delta_1 \quad \Gamma \vdash e_1 \Leftarrow \tau_2 \rightsquigarrow d_2 : \tau_2' \dashv \Delta_2}{\Gamma \vdash e_1(e_2) \Rightarrow \tau \rightsquigarrow (d_1\langle \tau_1' \Rightarrow \tau_2 \to \tau \rangle)(d_2\langle \tau_2' \Rightarrow \tau_2 \rangle) \dashv \Delta_1 \cup \Delta_2}$$

$$\textsc{ESEHole}\ \frac{}{\Gamma \vdash (\!|\,|\!)^u \Rightarrow ? \rightsquigarrow (\!|\,|\!)^u_{\mathrm{id}(\Gamma)} \dashv u :: (\!|\,|\!)[\Gamma]}$$

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

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

$\boxed{\Gamma \vdash e \Leftarrow \tau \rightsquigarrow d : \tau' \dashv \Delta}$ e analyses against type τ and elaborates to d of consistent type τ'

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

$$\textsc{EASubsume}\ \frac{e \neq (\!|\,|\!)^u \quad e \neq (\!|e'|\!)^u \quad \Gamma \vdash e \Rightarrow \tau' \rightsquigarrow d \dashv \Delta \quad \tau \sim \tau'}{\Gamma \vdash e \Leftarrow \tau \rightsquigarrow d : \tau' \dashv \Delta}$$

$$\textsc{EAEHole}\ \frac{}{\Gamma \vdash (\!|\,|\!)^u \Leftarrow \tau \rightsquigarrow (\!|\,|\!)^u_{\mathrm{id}(\Gamma)} : \tau \dashv u :: \tau[\Gamma]}$$

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

Figure 2.5: Elaboration judgements

$$id(x_1 : \tau_1, \ldots, x_n : \tau_n) := [x_1/x_1, \ldots, x_n/x_n]$$

$\Delta; \Gamma \vdash \sigma : \Gamma'$ iff $\mathrm{dom}(\sigma) = \mathrm{dom}(\Gamma')$ and for every $x : \tau \in \Gamma'$ then: $\Delta; \Gamma \vdash \sigma(x) : \tau$

Figure 2.7: Identity substitution and substitution typing

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

$$\text{TACons}\frac{}{\Delta; \Gamma \vdash c : b} \qquad \text{TAVar}\frac{x : \tau \in \Gamma}{\Delta; \Gamma \vdash x : \tau} \qquad \text{TAFun}\frac{\Delta; \Gamma, x : \tau_1 \vdash d : \tau_2}{\Delta; \Gamma \vdash \lambda x : \tau_1 . d : \tau_1 \to \tau_2}$$

$$\text{TAApp}\frac{\Delta; \Gamma \vdash d_1 : \tau_2 \to \tau \quad \Delta; \Gamma \vdash d_2 : \tau_2}{\Delta; \Gamma \vdash d_1(d_2) : \tau} \qquad \text{TAEHole}\frac{u :: \tau[\Gamma'] \in \Delta \quad \Delta; \Gamma \vdash \sigma : \Gamma'}{\Delta; \Gamma \vdash (\!|)^u_\sigma : \tau}$$

$$\text{TANEHole}\frac{\Delta; \Gamma \vdash d : \tau' \quad u :: \tau[\Gamma' \in \Delta \quad \Delta; \Gamma \vdash \sigma : \Gamma']}{\Delta; \Gamma \vdash (\!|d|\!)^u_\sigma : \tau} \qquad \text{TACast}\frac{\Delta; \Gamma \vdash d : \tau_1 \quad \tau_1 \sim \tau_2}{\Delta; \Gamma \vdash d\langle \tau_1 \Rightarrow \tau_2 \rangle : \tau_2}$$

$$\text{TACastError}\frac{\Delta; \Gamma \vdash d : \tau_1 \quad \tau_1 \text{ ground} \quad \tau_2 \text{ ground} \quad \tau_1 \neq \tau_2}{\Delta; \Gamma \vdash d\langle \tau_1 \Rightarrow ? \not\Rightarrow \tau_2 \rangle : \tau_2}$$

Figure 2.6: Type assignment judgement for *internal expressions*

Where ground types are base types or one-level unrollings of the dynamic type (each being the *least specific* type for each compound type).

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

$$\text{GBase}\frac{}{b \text{ ground}} \qquad \text{GDynFun}\frac{}{? \to ? \text{ ground}}$$

Figure 2.8: Ground types

This elaboration is proven to produce unique internal expressions and hole contexts, and to preserve well-typedness.

Internal Language: Dynamic Semantics

In order to support the ability to evaluate expressions around holes and cast errors, Hazel defines multiple syntax-directed classes of final forms in Fig. 2.9. *Final forms* are irreducible expressions.

- Values – Constants or functions.
- Boxed values – Values or boxed values in one of the two cast forms. These must be unboxed (downcast) before reducing.
- Indeterminate forms – Irreducible terms containing holes or are casts errors. Substitution of holes may make these reducible.
- Final – All final forms.

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

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

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

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

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

$$\text{BVVal}\frac{d \text{ val}}{d \text{ boxedval}} \qquad \text{BVFunCast}\frac{\tau \to \tau_2 \neq \tau_3 \to \tau_4 \quad d \text{ boxedval}}{d\langle \tau_1 \to \tau_2 \Rightarrow \tau_3 \to \tau_4 \rangle \text{ boxedval}}$$

$$\text{BVDynCast}\frac{d \text{ boxedval} \quad \tau \text{ ground}}{d\langle \tau \Rightarrow ? \rangle \text{ boxedval}}$$

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

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

$$\text{ICastGD}\frac{d \text{ indet} \quad \tau \text{ ground}}{d\langle \tau \Rightarrow ? \rangle \text{ indet}} \qquad \text{ICastDG}\frac{d \neq d' \langle \tau' \Rightarrow ? \rangle \quad d \text{ indet} \quad \tau \text{ ground}}{d\langle ? \Rightarrow \tau \rangle \text{ indet}}$$

$$\text{ICastFun}\frac{\tau_1 \to \tau_2 \neq \tau_3 \to \tau_4 \quad d \text{ indet}}{d\langle \tau_1 \to \tau_2 \Rightarrow \tau_3 \to \tau_4 \rangle \text{ indet}} \qquad \text{ICastError}\frac{d \text{ final} \quad \tau_1 \text{ ground} \quad \tau_2 \text{ ground} \quad \tau_1 \neq \tau_2}{d\langle \tau_1 \Rightarrow ? \not\Rightarrow \tau_2 \rangle \text{ indet}}$$

Figure 2.9: Final forms

The small-step contextual dynamics [9] is defined on the internal expressions. The general idea is to consider two classes of casts: injections – casts from a ground type to the dynamic types, and projections – casts from the dynamic type to a ground type. These two classes of casts can be eliminated upon meeting if the ground types are equal or to a cast error if not. Function casts are dealt with by separating into two casts on the argument and return value. Finally, compound types can be cast to their least specific ground type specified by the

ground matching relation in Fig. 2.10.

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

$$\frac{\tau_1 \to \tau_2 \neq ? \to ?}{\tau_1 \to \tau_2 \blacktriangleright_{\text{ground}} ? \to ?}$$

Figure 2.10: Ground type matching

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

$$\text{ITFun}\ \frac{}{(\lambda x : \tau.d_1)(d_2) \longrightarrow [d_2/x]d_1} \qquad \text{ITCastId}\ \frac{}{d\langle \tau \Rightarrow \tau \rangle \longrightarrow d}$$

$$\text{ITAppCast}\ \frac{\tau_1 \to \tau_2 \neq \tau_1' \to \tau_2'}{d_1\langle \tau_1 \to \tau_2 \Rightarrow \tau_1' \to \tau_2' \rangle(d) \longrightarrow (d_1(d_2\langle \tau_1' \Rightarrow \tau_1 \rangle))\langle \tau_2 \Rightarrow \tau_2' \rangle}$$

$$\text{ITCast}\ \frac{\tau\ \text{ground}}{d\langle \tau \Rightarrow ? \Rightarrow \tau \rangle \longrightarrow d} \qquad \text{ITCastError}\ \frac{\tau_1 \neq \tau_2 \quad \tau_1\ \text{ground} \quad \tau_2\ \text{ground}}{d\langle \tau_1 \Rightarrow ? \Rightarrow \tau_2 \rangle \longrightarrow d\langle \tau_1 \Rightarrow ? \not\Rightarrow ? \rangle \tau_2}$$

$$\text{ITGround}\ \frac{\tau \blacktriangleright_{\text{ground}} \tau'}{d\langle \tau \Rightarrow ? \rangle \longrightarrow d\langle \tau \Rightarrow \tau' \Rightarrow ? \rangle} \qquad \text{ITExpand}\ \frac{\tau \blacktriangleright_{\text{ground}} \tau'}{d\langle ? \Rightarrow \tau \rangle \longrightarrow d\langle ? \Rightarrow \tau' \Rightarrow \tau \rangle}$$

Figure 2.11: Instruction transitions

Variable substitution, used in ITFun, is capture avoiding and also substitutions are recorded in each hole's environment σ by (over)writing $[d/x]\sigma$.[2]

[2]Create a figure for this, maybe in appendix.

Context syntax:

$$E ::= \circ \mid E(d) \mid d(E) \mid (\!|E|\!)^u_\sigma \mid E\langle \tau \Rightarrow \tau \rangle \mid E\langle \tau \Rightarrow ? \Rightarrow \tau \rangle$$

$\boxed{d = E[d]}$ d is the context E filled with d' in place of $\circ$

$$\text{ECOuter}\frac{}{d = \circ[d]} \qquad \text{ECApp1}\frac{d_1 = E[d_1']}{d_1(d_2) = E(d_2)[d_1]} \qquad \text{ECApp2}\frac{d_2 = E[d_2]}{d_1(d_2) = d_1(E)[d_2']}$$

$$\text{ECNEHole}\frac{d = E[d']}{(\!|d|\!)^u_\sigma = (\!|E|\!)^u_\sigma[d']} \qquad \text{ECCast}\frac{d = E[d']}{d\langle \tau_1 \Rightarrow \tau_2 \rangle = E\langle \tau_1 \Rightarrow \tau_2 \rangle[d']}$$

$$\text{ECCastError}\frac{d = E[d']}{d\langle \tau_1 \Rightarrow ? \Rightarrow \tau_2 \rangle = E\langle \tau_1 \Rightarrow ? \Rightarrow \tau_2 \rangle[d']}$$

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

$$\text{Step}\frac{d_1 = E[d_2] \qquad d_2 \longrightarrow d_2' \qquad d_1' = E[d_2']}{d_1 \mapsto d_1'}$$

Figure 2.12: Contextual dynamics of the internal language

The instruction transitions, Fig. 2.11, and evaluation context, Fig. 2.12, specify a non-deterministic evaluation order, which is a more suitable choice for supporting dynamic hole instantiation, as will be used by the search procedure.[3]

Casts are associative, so bracketing is omitted.

Hole Substitutions

Hole substitutions will be extensively used by the search procedure, so their semantics is given here. These correspond to contextual substitutions for meta-variables in closures in contextual modal type theory [10]. Intuitively, these are delayed substitutions with the holes being closures with environment σ and meta-variable/hole name u. A hole $(\!|\,|\!)^u_\sigma$ being substituted

[3] **Define Substitution in appendices?**

with d (written $[\![d/u]\!](|)^u_\sigma$) is d with it's environment assignment σ applied, after first substituting the hole ($[\![d/u]\!]\sigma$) in σ.

Hole/contextual substitutions are mostly similar variable substitutions, except the substituted values may refer to bound variables without restriction. See Fig. 2.13 for full definition.

$\boxed{[\![d/u]\!]d' = d''}$ d'' is d' with each hole u substituted with d in the respective hole's environment σ.

$$
\begin{array}{lll}
[\![d/u]\!]c & = c & \\
[\![d/u]\!]x & = x & \\
[\![d/u]\!]\lambda x : \tau.d' & = \lambda x : \tau.[\![d/u]\!]d' & \\
[\![d/u]\!]d_1(d_2) & = ([\![d/u]\!]d_1)([\![d/u]\!]d_2) & \\
[\![d/u]\!](|)^u_\sigma & = [[\![d/u]\!]\sigma]d & \\
[\![d/u]\!](|)^v_\sigma & = (|)^b_{[\![d/u]\!]\sigma} & \text{if } u \neq v \\
[\![d/u]\!](d')^u_\sigma & = [[\![d/u]\!]\sigma]d & \\
[\![d/u]\!](d')^v_\sigma & = (|[\![d/u]\!]d'|)^b_{[\![d/u]\!]\sigma} & \text{if } u \neq v \\
[\![d/u]\!]d'\langle\tau\Rightarrow\tau'\rangle & = ([\![d/u]\!]d')\langle\tau\Rightarrow\tau'\rangle & \\
[\![d/u]\!]d'\langle\tau\Rightarrow?\not\Rightarrow\tau'\rangle & = ([\![d/u]\!]d')\langle\tau\Rightarrow?\not\Rightarrow\tau'\rangle &
\end{array}
$$

$\boxed{[\![d/u]\!]\sigma = \sigma'}$ σ' is σ with each hole u in σ substituted with d in the respective hole's environment.

$$
\begin{aligned}
[\![d/u]\!]\cdot &= \cdot \\
[\![d/u]\!]\sigma, d'/x &= [\![d/u]\!]\sigma, ([\![d/u]\!]d')/x
\end{aligned}
$$

Figure 2.13: Hole substitution

Think in detail about CMTT and hole refinement. It is very relevant to the search procedure. In particular think about `box` and `letbox` in CMTT.

2.1.3 Hazel Codebase

2.2 The Project

2.2.1 Cast Slicing

Cast slicing is, to my knowledge, a new concept and is a method in which selected casts can be linked back to source code by creating a slice of all code contributing to the cast.

Slicing methods have been researched extensively since Wadler first proposed static program slicing [11], and later others proposed dynamic program slicing [12] and error slicing [13]. Cast slicing combines ideas from both dynamic slicing, tracking of casts during evaluation, and error slicing, tracking casts during static elaboration. But is relatively less expressive in terms of slicing criterion, only considering casts, and no other expressions.

2.2.2 Search Procedure

Search procedure for witnesses of type errors has been considered by Seidel et al. [14] for OCaml. Their approach creates a non-deterministic semantics for ill-typed OCaml with a hole in place of a function argument being dynamically type inferred and instantiated in a most general manner until evaluation gets stuck. The hole instantiation at this point is the error witness.

Adapting this search procedure to Hazel, which natively supports evaluation of ill-typed expressions and hole substitution allows for stronger semantic foundations. In particular hole substitution and refinement, borrowing from contextual modal type theory[4].

Further, the use of dynamic types, casts, and the more general notion of holes will allow better handling of non-parametric

[4]See Chapter 5 of [1]

function types – stated as a problem in Seidel et al's. paper. Hence, very different semantics will need to be devised, but the general principle of dynamic inference and hole instantiation will remain the same.

2.3 Starting Point

Concepts

The foundations of most concepts in understanding Hazel from Part IB Semantics of Programming (and Part II Types later). The concept of gradual typing briefly appeared in Part IB Concepts of Programming Languages, but was not formalised. Dynamic typing, gradual typing, holes, and contextual modal type theory were not covered in Part IB, so were partially researched leading up to the project, then researched further in greater depth during the early stages. Similarly, Part IB Artificial Integlligence provided some context to search procedures, however nothing was directly relevant. Primarily, the OCaml search procedure for ill-typed witnesses Seidel et al. [14] and the Hazel core language [1] were researched over the preceding summer.

Tools and Source Code

My experience in OCaml was mostly from the Part IA Foundations of Computer Science course and small-scale personal projects. The Hazel source code had not been inspected in detail until after starting the project.

2.4 Requirement Analysis

2.5 Software Engineering Methodology

2.6 Legality

Chapter 3

TEMPORARY: Implementation Plan

3.1 Cast Slicing

3.1.1 Theoretical Foundations and Context

Context: Why are casts elaborated

See The Gradualizer [15] for details on this terminology. It doesn't perfectly fit Hazel but is close. A cast is inserted at a point of either, for a subexpression $e : \tau$ if either:

- Pattern Matching $\tau \blacktriangleright \tau'$ – Wrap e in a cast $\langle \tau \Rightarrow \tau'' \rangle$ where τ'' is the *cast destination of* τ'
- Flows $\tau \rightsquigarrow \tau'$ – Wrap e in a cast $\langle \tau \Rightarrow \tau' \rangle$ – These correspond to uses of type consistency.

A *cast destination* for τ is the result of applying flows recursively on all positive type positions and reversed flows on negative type positions in τ.

Flows relate to invocations of consistency and joins, with direction dictated by type polarity and modality:

- Producers flow to their final type
- Final types flow to the consumers
- Input variables are replaced with the final type.

Where the final type is an annotated *type*, output consumers, or the join of all producers.[1]

Type slicing a term

A logical interpretation of this would be useful (see the relation of contexts slices and type slice decomposition in implications/functions). The below formulation feels quite ad-hoc.

A *type slice* of an e in a context is an expression that has sub-expressions of e replaced with holes or type annotations replaced with the dynamic type but still type synthesises or analyses to the same type.[2]

An alternative formalisation could be: Instead of type synthesises or analyses the same type, it could contain all code that could be changed in order to NO LONGER synthesise/analyse to the type.

However, as typing often depends on the context, for which code involved may be outside the scope of the current term, an additional notion of a context slice will be attached. A context slice will contain all the variable bindings (lambdas or let bindings) and, importantly, their annotations. It is important to notice that this only works because Hazel is explicitly typed at variable bindings. *An implicitly type language, like ML, could be translated to an explicitly typed internal language, for example CoreXML [4]. Then, 'constraint slices' could track code*

[1] Therefore types generally have output mode.

[2] Additionally they contain other info for use decomposing slices and the encompassing. Discussed throughout.

which is required for the constraint system to derive each annotation; this idea is similar to that used in type error slicing for higher-order functional languages [13], [16].

Slices for compound types will need to be split into slices for component types (as casts are decomposed), a suitable extension of this needs to be produced that allows slices to be decomposed into the slices of component types.

Compound types generally involve type checking sub-terms are of the component types, so these component slices can be obtained during this. But, for types involving variable binding, e.g. function types $\tau = \tau_1 \to \tau_2$, the slice of the argument τ_1 should simply be the binding annotation, in a sense this is the extension of the context slices between τ and τ_2[3].

Importantly, a hole analyses against any type, so any expressions analysed against can be replaced by holes[4].

On the other hand, synthesis rules may require expanding a slice. Mostly, by composing component slices as discussed above.

A formal definition of this is given below. Treating a slice ς of e, τ as a pair of a context slice and expression slice (γ, ε) indexed by τ. The context slice γ is simply a subset of the typing context Γ and the slice ε is a term e' which is less precise[5] than e and synthesises τ under the context slice. For simplicity adding the notion of unsubstitutable, unnamed holes ⦇⦈:

[3] A logical interpretation or analogy of this would be useful

[4] **FIX**: if using analysis on a non-subsumable type then maybe all non-subsumption rules should also construct slices.

[5] Ideally, least precise

Syntax:

$$\varsigma ::= \tau[\gamma, \varepsilon] \mid (\varsigma \to \varsigma)[\gamma, \varepsilon]$$

$\boxed{\Gamma \vdash e \Rightarrow \tau \parallel \varsigma}$ e synthesises type τ under context Γ and produces slice ς)

$$\text{SSConst}\ \frac{}{\Gamma \vdash c \Rightarrow b \parallel b[\cdot, b]} \qquad \text{SSVar}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x \Rightarrow \tau \parallel \tau[x : \tau, x]}$$

$$\text{SSFun}\ \frac{\Gamma, x : \tau_1 \vdash e \Rightarrow \tau_2 \parallel s[\gamma, \varepsilon]}{\Gamma \vdash \lambda x : \tau_1 . e \Rightarrow \tau_1 \to \tau_2 \parallel (\tau_1[x : \tau_1, ⦇⦈] \to s[\gamma, \varepsilon])[\gamma \backslash \{x : \tau_1\}, \lambda x : \tau_1 . \varepsilon]}$$

$$\text{SSApp}\ \frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \parallel \varsigma_1 \qquad \tau_1 \blacktriangleright_{\to} \tau_2 \to \tau \qquad \varsigma_1 \blacktriangleright_{\to} (\varsigma_2 \to s[\gamma, \varepsilon])[\gamma_1, \varepsilon_1] \qquad \Gamma \vdash e_2 \Leftarrow \tau_2}{\Gamma \vdash e_1(e_2) \Rightarrow \tau \parallel s[\gamma_1, \varepsilon_1(⦇⦈)]} \qquad \text{SSEHole}\ \frac{}{\Gamma \vdash ⦇⦈^u \Rightarrow ? \parallel ?[\cdot, ⦇⦈]}$$

$$\text{SSNEHole}\ \frac{\Gamma \vdash e \Rightarrow \tau}{\Gamma \vdash ⦇e⦈^u \Rightarrow ? \parallel ?[\cdot, ⦇⦈]} \qquad \text{SAsc}\ \frac{\Gamma \vdash e \Leftarrow \tau}{\Gamma \vdash e : \tau \Rightarrow \tau \parallel \tau[\cdot, ⦇⦈ : \tau]}$$

Figure 3.1: Bidirectional typing judgements for *sliced* external expressions

Defining a slice matching relation analogously to type matching $\blacktriangleright_{\to}$.

Notice that for types involved with the typing context, like function arguments, where no such term with type τ_1 actually exists in code; the context slice gives required all info and the expression slice is filled with a dummy hole. **Formalise this better: could consider a term of type $\tau_1 \to ?$. Alternatively this may be better expressed as a transformation of DERIVATIONS themselves.**

For implementation and user purposes, the assumptions in the context slice γ derive from annotations, whose location in code would be tracked. This could be formalised in the above rules by annotating type assumptions with locations, or by considering a context slice could as a term with holes at every subterm except for the relevant bindings.

Analysis Contexts

Every type that is analysed must have been synthesised by some expression. This expression will be used as an 'analysis context' to allow proving formal properties and for use in determining the 'scopes of types' and in the search procedure. **Consider the weird cases like a synthesised function type; what is the expression of the argument??**

Slicing of Casts

Casts always go from expression types (for which a type slice exists) to either final types or consumers.

Producer types correspond to synthesised types. Consumer types correspond to analysed types. Final types are one of the above or additionally, joins of (producer) types.

Hence, as producer and consumers originate from type checking, their type slices can be obtained. The only remaining possibility is joins of types, so joins of type slices must be created.

First, I show how slices can be applied to elaboration rules, with casts now being between type slices $\langle \varsigma_1 \Rightarrow \varsigma_2 \rangle$. Notably, additional slicing is required in analysis mode, but not in synthesis mode. When in analysis mode, the slice for τ', that is the actual internal type of the elaborated expression, is required. When in synthesis mode, the slice for τ can simply be obtained via the external type slicing.

Formal definition below. Omitting the trivial synthesis cases not involving casts.

Redefining elaboration synthesis:

$$\text{ESApp}\ \frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \parallel \varsigma_1 \qquad \tau_1 \blacktriangleright_{\to} \tau_2 \to \tau \qquad \varsigma_1 \blacktriangleright_{\to} (\varsigma_2 \to \varsigma)[\gamma_1, \varepsilon_1] \quad \Gamma \vdash e_1 \Leftarrow \tau_2 \to \tau \rightsquigarrow d_1 : \tau_1' \dashv \Delta_1 \parallel \varsigma_1' \qquad \Gamma \vdash e_1 \Leftarrow \tau_2 \rightsquigarrow d_2 : \tau_2' \dashv \Delta_2 \parallel \varsigma_2'}{\Gamma \vdash e_1(e_2) \Rightarrow \tau \rightsquigarrow (d_1\langle \varsigma_1' \Rightarrow \varsigma_1 \rangle)(d_2\langle \varsigma_2' \Rightarrow \varsigma_2 \rangle) \dashv \Delta_1 \cup \Delta_2}$$

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

$\boxed{\Gamma \vdash e \Leftarrow \tau \rightsquigarrow d : \tau' \dashv \Delta \parallel \varsigma'}$ e analyses against type τ and elaborates to d of consistent type τ' with slice ς'

$$\text{SEAFun}\ \frac{\tau \blacktriangleright_{\to} \tau_1 \to \tau_2 \qquad \Gamma, x : \tau_1 \vdash e \Leftarrow \tau_2 \rightsquigarrow d : \tau_2' \dashv \Delta \parallel s[\gamma, \epsilon]}{\Gamma \vdash \lambda x.e \Leftarrow \tau \rightsquigarrow \lambda x : \tau_1.d : \tau_1 \to \tau_2' \dashv \Delta \parallel (\tau_1[x : \tau_1, (\!|\,|\!)] \to s[\gamma, \varepsilon])[\gamma \backslash \{x : \tau_1\}, \lambda x.\varepsilon]}$$

$$\text{SEASubsume}\ \frac{e \neq (\!|\,|\!)^u \qquad e \neq (\!|e'|\!)^u \qquad \Gamma \vdash e \Rightarrow \tau' \parallel \varsigma' \quad \Gamma \vdash e \Rightarrow \tau' \rightsquigarrow d \dashv \Delta \qquad \tau \sim \tau'}{\Gamma \vdash e \Leftarrow \tau \rightsquigarrow d : \tau' \dashv \Delta \parallel \varsigma'}$$

SEAEHole, SEANEHole are simply analogues of the synthesis slicing rules.

Figure 3.2: Elaboration judgements with slicing

Second, I describe how to calculate joins of slices. This depends on the formalisation of type slicing. If only a minimal code slice to produce the same type is required, then some branches with matching type may be replaced by holes. If we include any code that can cause type checking to change, a join of slices is just the union of the code they point at[6].

Semantically, in the above formulation it is a bit tedious to define and doesn't quite fit the pattern that the type slice actually corresponds to a term, here it corresponds to a union of branches. **The idea of a type slice should be refined to accept function argument types and joined types more intuitively**[7].

[6] Prove.

[7] Consider having join types be explicit. The lack of a link between these types and an expression is the issue.

Implementation-wise, slices will probably be code locations, and will be more granular than expressions.[8] Granularity at the level of annotations, and could potentially have special cases to distinguish cases where the slicing semantics inserts holes more clearly: for example, highlighting the brackets/hole being applied to a function in SSFun to make it clear that the type derives from *application* rather than just the function which has a very similar slice. *The hole based approach does have the benefit of automatically closing away irrelevant branches of code that is not highlighted, this could be a nice interaction design feature to implement.*

Context: How are casts evaluated

The Hazel cast calculus has two primary types of casts:

- Injections – Casts to the dynamic type from a ground type
- Projections – Casts from the dynamic type to the dynamic type

These types of casts can be eliminated upon meeting or transformed into a cast error if the two involved ground types are inconsistent.

Inserted casts don't necessarily fall into either of these two classes. If the cast is a non-ground injection/projection then they can be transformed by matching to a least specific ground type, e.g. $\tau_1 \to \tau_2 \blacktriangleright_{\text{ground}} ? \to ?$. This ground type can be inserted in the middle of such a cast (ITGround, ITExpand). If the cast is between compound types and reducible by a suitable elimination rule (if casts were ignored), then these casts will be decomposed.

See [17] for in depth intuition.

[8] Hence, the issues above are irrelevant to implementation.

Cast Splitting

There are only 3 rules that non-trivially modify casts: ITAppCast, ITGround, ITExpand. For ITGround and ITExpand, it makes sense to keep the cast slices for τ' the same as τ.[9] ITAppCast requires decomposing the function cast, luckily slices recursively include type slices of their type components.

$$\text{ITAppCast}\ \frac{\tau_1 \to \tau_2 \neq \tau_1' \to \tau_2'}{d_1\langle \varsigma_1 \to \varsigma_2 \Rightarrow \varsigma_1' \to \varsigma_2' \rangle(d_2) \to (d_1(d_2\langle \varsigma_1' \Rightarrow \varsigma_1 \rangle)\langle \varsigma_2 \Rightarrow \varsigma_2' \rangle)}$$

Recording dependencies between casts from this might be useful. This allows a user to see how a cast was formed.

Random notes and refs

Consider options of: annotating typing derivations, annotating casts (and splitting in evaluation), or reverse unevaluating casts [18].

See constraint free error slicing [19].

See the gradualizer to understand how casts work [15].

See Blame tracking to see how common type location transfer is handled [20].

Casting:

Casting goes from the terms actual type to coerce it into a new type. Casts are inserted by elaboration at appropriate points (where the type system uses consistency or pattern matching[10]). Casts between compound types may be decomposed by the cast calculus. See gradualizer dynamic semantics

[9] Justify this. Maybe casting to ground type was not in code so the cast should not highlight anything, but instead record it's *dependencies*?

[10] See the gradualizer for intuition and direction of consistency casts

[17] for intuition and [20] for intuition on polarity affecting blame.

Consider deeply type constructor polarities [21, pg. 473]. Covariant types give positive blame (blame value) and contravariant give negative (blame context).

3.1.2 Implementation Details and Optimisations

Slices – Location based

Location based slices would could be represented by intervals.

Due to the recursive nature of slices (component type slices must be available), then there will be a significant number of interval trees. Therefore, a persistent structure [22, chapter 2] MUST be used for space efficiency.

Efficiently displaying overlapping intervals is ideal. A sorted list achieves this, so a sorted tree structure of some sort could be used to maintain efficient insertion.

Hash consing may be similarly useful.

Displaying a slice here does need to be more efficient as there may be many intervals that aren't actually merged vs just walking a tree with holes.

Slices – Hole Based

Slices can be stored immutably as ASTs and splitting of slices will be handled in a space efficient way by default by OCaml's persistent nature.

Overlapping slices can appear, most notably from joining slices. Therefore, joins should be implemented to preserve space. Speed of merging and splitting is also relatively important.

Displaying a slice is a relatively uncommon procedure compared to joining and splitting. Hence, retrieving and aggregating locations from the slice does not need to be fast, nor space efficient.

A regular persistent tree is likely sufficient, but lazy joining and/or hash consing [23] might be useful.

3.2 Search Procedure

Look into Zippers [24], and other functional data structures that may be useful here.

3.2.1 Hole Refinement

A suitable notion of *type hole* should be used for this. These will differ from regular holes in that they will maintain runtime type information[11]. In particular they will be annotated by a dynamically inferred type variable, which would be progressively refined.

In particular the type given to a hole will be *entirely* determined by the dynamics, and not the static type checking, which may not be well-typed or may be unspecific (using dynamic types).

Look into Gradual type inference [25], Gradual dynamic type inference [26], Seidel et al. [14], OLEG [27, chapter 2] & Idris for ideas. Look into how Hazel may support type-driven code completion.

3.2.2 Hole Instantiation

Substitution semantics via CMTT contextual substitution.

[11] Or the evaluation environment will track these.

Generating of values to substitute will match the required type as annotated on the hole. These should be the most general partial values to meet the type, see OLEG refinement [27, chapter 2].

Generation of suitable concrete values of any type should be informed by ideas like the 'small scope hypothesis' [28][12] or just could be random as in QuickCheck [29], SmallCheck [30]. Allowing some meta-programming of the value generation algorithms would be an interesting idea, especially for user-defined types.

The idea of instantiating in order to effectively explore branches is a good idea. This is essentially symbolic execution, and exploring branches space-time-efficiently without overlap is the goal.[13]

Some heuristics may help with this search. But this project is unlikely to consider any in detail.

Note, both hole refinement and instantiation can be handled as a search procedure wrapper around the stepper.

3.2.3 Witness Generality

A witness is a hole instantiation that evaluates to a final form containing a cast error(s). *Witness generality* should ensure that generating a witness via the search procedure implies that for any type, a witness exists of this type.

To achieve this, holes should only be instantiated when absolutely required, i.e. lazily as in SmallCheck [30]. For example, even if a function has a cast requiring integer arguments, the hole instantiation should be delayed, as passing a hole is

[12]This is evaluated for bugs in general in OOP, not necessarily type bugs in FP.

[13]This is the main idea to think about, probably more important than user control of value generation.

still safe. When the hole passes into the function it will accumulate a cast to int, but the inferred type should remain ambiguous until evaluation gets stuck (such as at a case analysis). If this inferred type conflicts with the casts, hole instantiation to the inferred type may still be useful, and Hazel still supports evaluation.

Alternatively, annotations could be treated as correct, meaning casts would refine the inferred type, and thereby treat any value instantiation at this point as a witness. Note that the use of dynamic types still allows the procedure to find general witnesses.

3.2.4 Instruction Transitions

Lazy hole instantiation and generation can be formalised with a *non-deterministic* step semantics in Fig. **??**.

The below semantics treat casts as correct, and use them to refine types.

$$d ::= \ldots \mid (\!|\tau|\!)^u_\sigma$$

$$\text{GenConst}\ \frac{\text{primitive op } p}{p((\!|b|\!)^u_\sigma) \longrightarrow p(c)} \qquad \text{GenFun}\ \frac{\text{fresh } x}{(\!|\tau_1 \to \tau_2|\!)^u_\sigma(d) \longrightarrow (\lambda x : \tau_1.(\!|\tau_2|\!)^u_{\sigma,x/x})(d)}$$

$$\text{Refine}\ \frac{}{(\!|\tau|\!)^u_\sigma\langle\tau \Rightarrow \tau'\rangle \longrightarrow (\!|\tau'|\!)^u_\sigma\langle\tau \Rightarrow \tau'\rangle}$$

Figure 3.3: Search Procedure Instruction Transitions

Upcasting causes the holes to *lose* type information. This shouldn't be a problem because the only time values are actually generated, the required type can be inferred from the primitive operation, or by generating a dynamic typed function.

3.2.5 Backtracking & Non-determinism

If a particular instantiation does not result in a witness, then some form of backtracking is required. This is the primary addition to the small-step evaluator that needs to be added.

A simple way to do this would be wrapping the evaluator in a choice-pointing system – like in Prolog.

Doing this space-efficiently (saving choice points) would be the goal.

Look into non-deterministic evaluation from a logic programming perspective. Also backtracking in general. See the Warren Abstract Machine [31].

A *semi-persistent* data structure [32] will be needed for this.

In order to find witnesses that are shorter in trace length (and hence easier for the user to understand), it may be useful to perform iterative deepening. Otherwise, trace compression and input simplification (as in QuickCheck [29])

For efficiency reasons, it may be desirable to use some amount of immutability here.[14]

Also, `call/cc` or `shift/reset` could be useful (or maybe even effect handlers [33]) for managing the desired control flow.

3.2.6 Cast Dependence

A cast error for associated with a witness should depend upon the value instantiated. Unrelated cast errors are not proof that the witness is valid.

This notion of dependence can be modelled by counting only casts that are attached to sub-terms of an instantiated value. *If an unrelated cast error causes evaluation to get stuck, this could be reported in some way to the user.*

[14] Hazel code-base rules seem to not allow this though!

3.2.7 Evaluation Order

Making this work for multiple search holes in arbitrary position might be too difficult. The other option is to only support this attached to functions.

It is inefficient to evaluate the whole program in order to perform the search procedure. Instead evaluation should be directed first around search procedure holes[15].

The evaluation order could be formalised by adapting the evaluation context to have *multiple* markers, and define inserting several terms into those marks with *strong*[16] non-deterministic semantics, Fig. 3.4. Then constructing the context would put markers for each redex involving a search hole and evaluation would expand around them[17].

$$
\begin{aligned}
&E_0 ::= d \\
&E_1 ::= E \\
&E_n ::= E_k(E_{n-k}) \mid (\!|E_n|\!)^u_\sigma \mid E_n\langle \tau \Rightarrow \tau \rangle \mid E_n\langle \tau \Rightarrow ? \nRightarrow \tau \rangle \qquad \text{for } n > 1 \text{ and } k \leq n
\end{aligned}
$$

$$
\text{Step}\ \frac{d = E_n(d_1, \cdots, d_i, \cdots, d_n) \qquad d_i \longrightarrow d_i' \qquad d' = E_n(d_1, \cdots, d_i', \cdots, d_n)}{d \mapsto d'}
$$

Figure 3.4: Search Procedure Evaluation Context

Some form of *concurrent zipper* via delimited continuations [34], [35] for constant (or fast) access to the term context around each search hole would be ideal.

Finally, two search holes may meet. What to do in this case?

[15] Of course, only possible because Hazel is pure.

[16] allowing reductions inside functions

[17] This ordering seems hard to display in maths, but the implementation should be straightforward

3.2.8 Curried Functions & Attached Search Holes

For functions, it is useful to get a witness for multiple arguments. But, for curried functions, each application has extra syntax and is not instantiable by a single hole. In this case, if a general witness has not already been found and the final form is a function, then more search holes could be created. See saturaion in Seidel et al. [14].

A nice way to represent this would be to thinking of this would be *attaching the function to a search hole*[18], which instantiates to the function with n applications (then n may change during backtracking). Besides using application onto search holes, attaching functions could just use the existing non-empty hole machanism. Further, in a sense a regular search hole is the above but for $n = 0$; however, operationally it does not make sense to combine the two[19]. **Think about user interaction for this to guide this design.**[20]

3.2.9 Witness Result

A witness will be an initial instantiation leading to a final form containing a cast error(s). Or potentially a trace containing a cast error (if annotations are treated as correct), for which stopping evaluation before the cast error is removed is ideal.

A manner in which to get the execution of the actually found witness result will be needed. So, whole traces would

[18] Alternatively thought of as applying the function to the hole and instantiating accordingly

[19] Fundamentally, there is a difference between asking for a witness of the 1st arg of a function vs. the whole function.

[20] e.g. make clear the distinction between saturated witnesses and regular witnesses. Maybe directly annotate a term with a special application: `{hole}` as syntactic sugar for applying n remaining arguments for witness

need to be stored/accumulated. Consider the space considerations of this. Look into Hazel's stepper and history functionality.

The trace should ideally be presented in the standard call-by-value evaluation order, rather than the search procedure evaluation order.

Consider trace reduction and/or witness reduction (see quickcheck).

3.2.10 How do search holes react upon meeting?

Multiple holes can meet, and this is where this method can find witnesses that were NOT possible in Seidel. But, a way of choosing values to instantiate in these cases must be devised AND the backtracking must be devised to recalculate too much information.

3.2.11 Coverage & Time Limit

Due to this problem being undecidable and with potential infinite loops, a trace length limit will be imposed. Further, by using coverage metrics via symbolic evaluation, possible values to generate could be fully exhausted, showing that the code is actually safe.

3.2.12 Hazel Implementation

It appears the main branch of Hazel has not actually implemented fill-and-resume. However, an earlier branch has – haz3l-fill-and-resume.

This old branch does NOT include all features required by my core goal – missing *sum types* and *type aliases*. And advanced features like polymorphism are not all implemented

with fill-and-resume. To reach my extension goal I must implement some form of fill-and-resume for the dev branch. And the core goal requires adding *at least* sum types and type aliases to the old branch. **This is an unforseen extra task.**

Maybe a hacky fill-and-resume that works just well enough for the search procedure could be implemented.

3.3 Type Error Witnesses Interaction in General

There are various classes of type errors, and witnesses will be evaluation traces to cast errors directly witnessing the type error.[21]:

3.3.1 Static Error Witnesses & Type Witnesses

Each expression synthesises a type. A witness (evaluation to a value) can be provided for (some) of these. **Do this.**

Some expressions analyse against a type. The witness of the origin of the analysed type will have a type witness derived from synthesis type witnesses of it's components?

Then for each class of type errors:

- Inconsistent against analysis type – Use the type witness of the expression which imposed the analysis.
- Inconsistent branches – Compare synthesised type witnesses of the inconsistent branches.
- Bad application –

[21] Come up with a formal way of saying this.

Type Synthesis Witnesses

Consider:

```
let sqsum = fun xs -> case xs
  | [] => 0
  | h :: t => sqsum(t) @ (h * h) end
in ?
```

and also with `sqsum` annotated with `[?] -> ?` or `[String] -> ?`.

How to treat vars?

Treat them as instances to use the search procedure on. There are various instances for free vars:

- Let binding – These always have a direct definition *(even recursive bindings, via fix)*, and hence can be substituted directly.
- Fun binding – Use of such a binding implies the current expression is a component part of the function itself. This function can be searched using the search procedure. Essentially, this is taking the variable as an input to the procedure.
- Pattern binding – These can be desugared into a one of the above bindings followed by deconstructing the bound value. i.e. in a fun binding it is a restriction on possible instantiations.

Note that the values by the above may NOT be possible in practice. i.e. if they do not result from a witness of the variable. Highlight these?

Static Error Witnesses

Essentially when putting errors into holes at compile time, tag the holes with their analysed type and original expression and context.

Then a search procedure stage can be added which performs the search on each hole treating free vars as above. For each type error we do:

- Inconsistent type between synthesised and analysed:

  We can get a type witness of it's synthesised type, instantiating any holes within appropriately (make this very clear in the visualisation[22]).

  The witness of this hole is then the analysis context using this witness (and any other required witnesses).

- Inconsistent branches: Get a type witness of each branch and compare.

- Bad function position (inconsistent with arrow): Get witness of the expected arrow. The result will be a something applied to a non-function (also make sure the cast slice error produced here works).

- No type: e.g. bad application, bad token, free constructor – Treat as holes.

Treatment of holes:

Treat as search procedure inputs and instantiate accordingly if progress is not being made.

[22] e.g. an error hole containing an int 1 could be instantiated to a string "str" if required

On the interaction of multiple errors

Think about cases where error holes may be dependent? Or where recursion may result in a hole depending on itself?

3.3.2 Treatment of dynamic typed regions:

There may still be traces leading to a cast error that are not statically caught due to presence of dynamically typed regions.

Selecting functions and searching for type errors (not linked to static errors) could be conducted as in Seidel. This could be generalised to selection any binding, with nullary functions (constants) just being evaluated directly with no inputs.

3.3.3 Interaction

- Click on static errors to obtain witness.
- Allow checking any dynamic/partially dynamic function for potential type errors via search procedure. This could be a button when selecting such a function.
- Allow getting a type witness of any expression. Again, could be a button. This is essentially just evaluating at the cursor, but with lazy function input instantiation.

3.3.4 Proofs

The ideal proofs here would be, alongside the expected ones for evaluation of search procedure holes (generality, progress):

That cast errors should relate directly to the source static type error. The witness of a static type error will contain a cast attached directly to this term (or one of it's evaluated derivatives).

Chapter 4

Implementation

Implementation Here.

Chapter 5

Evaluation

Evaluation Here.

Evaluate small scope hypothesis for this problem. Note that small inputs don't necessarily correlate with small evaluation traces.

Chapter 6

Conclusions

Conclusions Here.

Bibliography

[1] C. Omar, I. Voysey, R. Chugh, and M. A. Hammer, "Live functional programming with typed holes," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–32, Jan. 2019, ISSN: 2475-1421. DOI: 10.1145/3290327. [Online]. Available: http://dx.doi.org/10.1145/3290327.

[2] B. C. Pierce and D. N. Turner, "Local type inference," *ACM Transactions on Programming Languages and Systems*, vol. 22, no. 1, pp. 1–44, Jan. 2000, ISSN: 1558-4593. DOI: 10.1145/345099.345100. [Online]. Available: http://dx.doi.org/10.1145/345099.345100.

[3] R. Harper and C. Stone, "A type-theoretic interpretation of standard ml," in *Proof, Language, and Interaction*. The MIT Press, May 2000, pp. 341–388, ISBN: 9780262281676. DOI: 10.7551/mitpress/5641.003.0019. [Online]. Available: http://dx.doi.org/10.7551/mitpress/5641.003.0019.

[4] R. Harper and J. C. Mitchell, "On the type structure of standard ml," *ACM Transactions on Programming Languages and Systems*, vol. 15, no. 2, pp. 211–252, Apr. 1993, ISSN: 1558-4593. DOI: 10.1145/169701.169696. [Online]. Available: http://dx.doi.org/10.1145/169701.169696.

[5] J. Dunfield and N. R. Krishnaswami, "Complete and easy bidirectional typechecking for higher-rank polymorphism," in *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, ser. ICFP'13, ACM, Sep. 2013. DOI: 10.1145/2500365.2500582. [Online]. Available: http://dx.doi.org/10.1145/2500365.2500582.

[6] "Gradual typing for functional languages," in.

[7] J. Siek and W. Taha, "Gradual typing for objects," in *ECOOP 2007 – Object-Oriented Programming*. Springer Berlin Heidelberg, 2007, pp. 2–27, ISBN: 9783540735892. DOI: 10.1007/978-3-540-73589-2_2. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-73589-2_2.

[8] F. PFENNING and R. DAVIES, "A judgmental reconstruction of modal logic," *Mathematical Structures in Computer Science*, vol. 11, no. 04, Jul. 2001, ISSN: 1469-8072. DOI: 10.1017/s0960129501003322. [Online]. Available: http://dx.doi.org/10.1017/S0960129501003322.

[9] R. Harper, *Practical Foundations for Programming Languages*. Cambridge University Press, Mar. 2016, ISBN: 9781316576892. DOI: 10.1017/cbo9781316576892. [Online]. Available: http://dx.doi.org/10.1017/CBO9781316576892.

[10] A. Nanevski, F. Pfenning, and B. Pientka, "Contextual modal type theory," *ACM Transactions on Computational Logic*, vol. 9, no. 3, pp. 1–49, Jun. 2008, ISSN: 1557-945X. DOI: 10.1145/1352582.1352591. [Online]. Available: http://dx.doi.org/10.1145/1352582.1352591.

[11] M. D. Weiser, "Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method.," 1979. DOI: 10.7302/11363. [Online]. Available: http://deepblue.lib.umich.edu/handle/2027.42/180974.

[12] B. Korel and J. Laski, "Dynamic program slicing," *Information Processing Letters*, vol. 29, no. 3, pp. 155–163, Oct. 1988, ISSN: 0020-0190. DOI: 10.1016/0020-0190(88)90054-3. [Online]. Available: http://dx.doi.org/10.1016/0020-0190(88)90054-3.

[13] F. Tip and T. B. Dinesh, "A slicing-based approach for locating type errors," *ACM Transactions on Software Engineering and Methodology*, vol. 10, no. 1, pp. 5–55, Jan. 2001, ISSN: 1557-7392. DOI: 10.1145/366378.366379. [Online]. Available: http://dx.doi.org/10.1145/366378.366379.

[14] E. L. Seidel, R. Jhala, and W. Weimer, "Dynamic witnesses for static type errors (or, ill-typed programs usually go wrong)," in *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP'16, ACM, Sep. 2016, pp. 228–242. DOI: 10.1145/2951913.2951915. [Online]. Available: http://dx.doi.org/10.1145/2951913.2951915.

[15] M. Cimini and J. G. Siek, "The gradualizer: A methodology and algorithm for generating gradual type systems," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '16, ACM, Jan. 2016, pp. 443–455. DOI: 10.1145/2837614.2837632. [Online]. Available: http://dx.doi.org/10.1145/2837614.2837632.

[16] C. Haack and J. Wells, "Type error slicing in implicitly typed higher-order languages," *Science of Computer Programming*, vol. 50, no. 1–3, pp. 189–224, Mar. 2004, ISSN: 0167-6423. DOI: 10.1016/j.scico.2004.01.004. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2004.01.004.

[17] M. Cimini and J. G. Siek, "Automatically generating the dynamic semantics of gradually typed languages," in

Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, ser. POPL '17, vol. 7, ACM, Jan. 2017, pp. 789–803. DOI: 10.1145/3009837.3009863. [Online]. Available: http://dx.doi.org/10.1145/3009837.3009863.

[18] R. Perera, U. A. Acar, J. Cheney, and P. B. Levy, "Functional programs that explain their work," *ACM SIGPLAN Notices*, vol. 47, no. 9, pp. 365–376, Sep. 2012, ISSN: 1558-1160. DOI: 10.1145/2398856.2364579. [Online]. Available: http://dx.doi.org/10.1145/2398856.2364579.

[19] T. Schilling, "Constraint-free type error slicing," in *Trends in Functional Programming*. Springer Berlin Heidelberg, 2012, pp. 1–16, ISBN: 9783642320378. DOI: 10.1007/978-3-642-32037-8_1. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-32037-8_1.

[20] P. Wadler and R. B. Findler, "Well-typed programs can't be blamed," in *Programming Languages and Systems*. Springer Berlin Heidelberg, 2009, pp. 1–16, ISBN: 9783642005909. DOI: 10.1007/978-3-642-00590-9_1. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-00590-9_1.

[21] B. C. Pierce, *Types and Programming Languages*, 1st. The MIT Press, 2002, ISBN: 0262162091.

[22] C. Okasaki, *Purely Functional Data Structures*. Cambridge University Press, Apr. 1998, ISBN: 9780511530104. DOI: 10.1017/cbo9780511530104. [Online]. Available: http://dx.doi.org/10.1017/CBO9780511530104.

[23] J.-C. Filliâtre and S. Conchon, "Type-safe modular hash-consing," in *Proceedings of the 2006 workshop on ML*, ser. ICFP06, ACM, Sep. 2006. DOI: 10.1145/1159876.1159880. [Online]. Available: http://dx.doi.org/10.1145/1159876.1159880.

[24] G. HUET, "The zipper," *Journal of Functional Programming*, vol. 7, no. 5, pp. 549–554, Sep. 1997, ISSN: 1469-7653. DOI: 10.1017/s0956796897002864. [Online]. Available: http://dx.doi.org/10.1017/S0956796897002864.

[25] R. Garcia and M. Cimini, "Principal type schemes for gradual programs," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15, Mumbai, India: Association for Computing Machinery, 2015, pp. 303–315, ISBN: 9781450333009. DOI: 10.1145/2676726.2676992. [Online]. Available: https://doi.org/10.1145/2676726.2676992.

[26] Y. Miyazaki, T. Sekiyama, and A. Igarashi, "Dynamic type inference for gradual hindley–milner typing," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019. DOI: 10.1145/3290331. [Online]. Available: https://doi.org/10.1145/3290331.

[27] C. McBride, "Dependently typed functional programs and their proofs," Doctoral Thesis, 2000.

[28] A. Andoni, D. Daniliuc, S. Khurshid, and D. Marinov, "Evaluating the "small scope hypothesis"," Oct. 2002.

[29] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of haskell programs," in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '00, New York, NY, USA: Association for Computing Machinery, 2000, pp. 268–279, ISBN: 1581132026. DOI: 10.1145/351240.351266. [Online]. Available: https://doi.org/10.1145/351240.351266.

[30] C. Runciman, M. Naylor, and F. Lindblad, "Smallcheck and lazy smallcheck: Automatic exhaustive testing for small values," in *Proceedings of the First ACM SIGPLAN*

Symposium on Haskell, ser. Haskell '08, Victoria, BC, Canada: Association for Computing Machinery, 2008, pp. 37–48, ISBN: 9781605580647. DOI: 10.1145/1411286.1411292. [Online]. Available: https://doi.org/10.1145/1411286.1411292.

[31] M. Pirog and J. Gibbons, "A functional derivation of the warren abstract machine," 2011, Submitted for publication. [Online]. Available: http://www.comlab.ox.ac.uk/jeremy.gibbons/publications/wam.pdf.

[32] S. Conchon and J.-C. Filliâtre, "Semi-persistent data structures," in *Programming Languages and Systems*. Springer Berlin Heidelberg, pp. 322–336, ISBN: 9783540787396. DOI: 10.1007/978-3-540-78739-6_25. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-78739-6_25.

[33] N. Xie and D. Leijen, "Effect handlers in haskell, evidently," in *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell*, ser. Haskell 2020, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 95–108, ISBN: 9781450380508. DOI: 10.1145/3406088.3409022. [Online]. Available: https://doi.org/10.1145/3406088.3409022.

[34] O. Kiselyov. "Generic zippers. "[Online]. Available: https://okmij.org/ftp/continuations/zipper.html.

[35] P. Panchekha. "Multi zippers. "[Online]. Available: https://pavpanchekha.com/blog/zippers/multi-zippers.html#org1556357.

[36] "Hazel project website. "[Online]. Available: https://hazel.org/.

[37] "Hazel source code. "[Online]. Available: https://github.com/hazelgrove/hazel.

Appendix A

Formal Semantics and Proofs

Example Appendix Here

Appendix B

Another Appendix

Another Example Appendix Here

Look at books for index structure reference. List key concepts, e.g. Core Hazel, Zipper, CMTT, Gradual Types.

Index

Project Proposal

Description

This project will add some features to the Hazel language [36]. Hazel is a functional research language that makes use of gradual types to support unusual features such as: holes (code placeholders) to give type meaning to incomplete programs. Importantly for this project, all Hazel programs, even ill-typed or incomplete programs, are evaluable. This allows dynamic reasoning about ill-typed programs via evaluation traces with the potential to improve the user's understanding of *why ill-typed programs go wrong*. See example below:

```
let rec sum : Int -> Int = fun n ->
  if n == 0 then
    true    // Type error statically caught and correctly loc
  else
    n + sum(n - 1)
in sum(2)
```

But evaluation is still possible; see below a (compressed) trace to a stuck value exhibiting a cast error:

$$\texttt{sum(2)} \mapsto^* \texttt{2 + sum(1)} \mapsto^* \texttt{2 + (1 + true}^{\langle \texttt{Bool} \Rightarrow \texttt{Int} \rangle}\texttt{)}$$

This project aims to exploit further this potential by providing some extra features to both: aid with finding values/inputs

that demonstrate why type-errors were found (type-error witnesses) and linking the evaluation traces back to source code. But is not expected to directly measure the usefulness of such evaluation traces themselves in debugging, nor is the design space for a Hazel debugger inspecting and interacting with traces to be explored.

Searching for type-error witnesses automatically is the main feature provided by this project, inspired by Seidel et al. [14]. The intended use of this is to automatically generate values (for example, function arguments) that cause ill-typed programs to 'go wrong' (lead to a cast error). More specifically, the search procedure can be thought of as evaluating a *special hole* which refines its type dynamically and non-deterministically instantiates itself to values of this type to find a value whose evaluation leads to a *general* cast error – 'general' meaning excluding trivial cast errors such as generating a value that doesn't actually have the refined expected type.

Such a search procedure is undecidable and subject to path explosion, hence the success criteria (detailed below) does not expect witnesses to be provided in general, even if they do exist. Sophisticated heuristics and methods to limit path explosion to support large code samples is not a core goal.

Formal semantics of this procedure and associated proofs is an extension goal, consisting of preservation proofs and witness generality proofs (formalising the notion of generality mentioned previously).

Secondly, *cast slicing* will track source code that contributed to any cast throughout the cast elaboration and evaluation phases. In particular, this allows a cast involved in a cast error relating to a type-error witness to point back to offending code. This is expected in some sense to be similar to blame tracking [20], error and dynamic program slicing [12], [13], although these are not directly relevant for this project.

Work required for the creation of an evaluation corpus of

ill-typed hazel programs, requiring manual work or creation of automated translation and/or fuzzing tools, is timetabled.

Starting Point

Only background research and exploration has been conducted. This consists of reading the Hazel research papers [36] and various other related research topics including: gradual types, bidirectional types, symbolic evaluation, OCaml error localisation and visualisation techniques.

More research, into the Hazel codebase in particular, and concrete planning is required and is timetabled accordingly.

Success Criteria

Core goals are the minimum expected goals that must be completed to consider this project a success. This corresponds to a working tool for a large portion of Hazel.

Extension goals will be timetabled in, but are relatively more difficult and not required for the project to be considered a success.

First, I give some definitions of terms:

- **Core Calculus** – The formal semantics core of Hazel as referred to by the Hazel research papers [1].
- **Basic Hazel** – A Hazel subset consisting of the core calculus, product and sum types, type aliases, bindings, (parametric) lists, bools, int, floats, strings, and their corresponding standard operations.
- **Full Hazel** – Hazel, including **Basic Hazel** plus pattern matching, explicit impredicative system-F style polymorphism and explicitly recursive types.

- **Core Corpus** – A corpus of ill-typed Hazel programs that are similar in complexity and size to student programs being taught a functional language, *e.g. (incorrect) solutions to the ticks in FoCS*. This will include examples in **Basic** or **Full Hazel** as required.
- **Extended Corpus** – A corpus of ill-typed Hazel programs that are larger in size, more akin to real-world code.
- **Evaluation Criteria** – Conditions for the search procedure to meet upon evaluation:
    1. Must have reasonable coverage – success in finding an *existing* witness which is correct and general.
    2. Must find witnesses in an amount of time suitable for interactive debugging – in-line with build-times for a debug build of existing languages.

Core Goals

- Success criteria for Cast Slicing – Cast slicing must be *correct* (slices must include all code involved in the cast) and work for *all casts*, including casts involved in cast errors. Informal reasoning in evidence of satisfying these conditions is all that will be required.
- Success criteria for the Search Procedure – The procedure must work for **Basic Hazel**, meeting the **Evaluation Criteria** over the **Core Corpus**. Analysis of some classes of programs for which witnesses could not be generated is also expected.

Extension Goals

- Search Procedure Extensions – Support for **Full Hazel** under the same criteria as above.
- Search Procedure Performance Extensions – Meeting of the **Evaluation Criteria** over an **Extended Corpus**
- Formal Semantics – The specification of a formal evaluation semantics for the search procedure over the **Core Calculus**. Additionally, a preservation and witness generality proof should be provided.

Work Plan

21st Oct *(Proposal Deadline)* – 3rd Nov

Background research & research into the Hazel semantics, cast elaboration, type system, and codebase. Produce implementation plan for cast slicing and the search procedure for the **Core Calculus**. This includes an interaction design plan, expected to be very minimal.

Milestone 1: Plan Confirmed with Supervisors

4th Nov – 17th Nov

Complete implementation of Cast Slicing for the **Core Calculus**. Write detailed reasoning for correctness, including plan for **Basic Hazel**. Add unit testing.

*Milestone 2: Cast slicing is complete for the **Core Calculus**.*

18th Nov – 1st Dec *(End of Full Michaelmas Term)*

Complete implementation of the search procedure for the **Core Calculus**.

*Milestone 3: Search Procedure is complete for the **Core Calculus**.*

2nd Dec – 20th Dec

Extension of both cast slicing and the search procedure to **Basic Hazel**.

*Milestone 4: Cast slicing & search procedure are complete for **Basic Hazel***

21st Dec – 24th Jan *(Full Lent Term starting 16th Jan)*

Basic UI interaction for the project. Drafts of Implementation chapter. Slack time. Expecting holiday, exam revision, and module exam revision. Should time be available, the **Formal Semantics** extension will be attempted.

Milestone 5: Implementation chapter draft complete.

25th Jan – 7th Feb *(Progress Report Deadline)*

Writing of Progress Report. Planning of evaluation, primarily including decisions and design of tools to be used to collect/create the **Core Corpus** and planning the specific statistical tests to conduct on the corpus. Collected corpus and translation method will be one of:

1. Manual translation of a small ill-typed OCaml program corpus into ill-typed Hazel.

2. Manual insertion of type-errors into a well-typed Hazel corpus.

3. Collection of a well-typed Hazel corpus.
   Tools: A Hazel type fuzzer to make the corpus ill-typed.

4. Collection of a well-typed OCaml corpus.
   Tools: OCaml -¿ Hazel translator/annotator which works with well-typed OCaml. A Hazel type fuzzer.

5. Collection of an ill-typed OCaml corpus.
   Tools: OCaml -¿ Hazel translator which works with ill-typed OCaml. *This would NOT be expected to be an implicitly typed Hazel front-end which maintains desireable properties like parametricity.*

Milestone 6: Evaluation plan and corpus creation method confirmed with supervisors.
Milestone 7: Underlying corpus (critical resource) collected.

8th Feb – 28th Feb

Implementation of the required tools for evaluation as planned. Some existing code or tools may be re-used, such as the OCaml type-checker.

Milestone 8: ***Core Corpus*** *has been collected.*

1st Mar – 15th Mar *(End of Full Lent Term)*

Conducting of evaluation tests and write-up of evaluation draft including results.

Milestone 9: Evaluation results documented.
Milestone 10: Evaluation draft complete.

16th Mar – 30th Mar

Drafts of remaining dissertation chapters. If possible, collection and evaluation of **Extended Corpus** using the same tools as the **Core Corpus**.

Milestone 11: Full dissertation draft complete and sent to supervisors for feedback.

31st Mar – 13th Apr

Act upon dissertation feedback. Exam revision.

Milestone 12: Second dissertation draft complete and send to supervisors for feedback.

14th Apr – 23rd Apr *(Start of Full Easter Term)*

Act upon feedback. Final dissertation complete. Exam revision.

Milestone 13: Dissertation submitted.

24th Apr – 16th May *(Final Deadline)*

Exam revision.

Milestone 14: Source code submitted.

Resource Declaration

- Underlying Corpus of either: Well-typed OCaml programs, Ill-typed OCaml programs, Hazel programs. For

use in evaluation. The required tools or manual translation to convert these into the ill-typed Hazel **Core Corpus** are detailed and allocated time in the timetable.

- Hazel source code. Openly available with MIT licence on GitHub [37].
- My personal laptop will be used for development, using GitHub for version control and backup of both code and dissertation. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure. A backup pc is available in case of such failure.