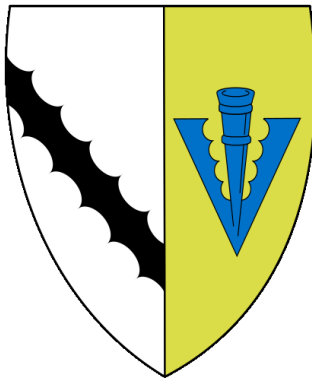


Max Carroll

Type Error Debugging in Hazel

Computer Science Tripos, Part II



Sidney Sussex College College
University of Cambridge
October 30, 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: **2996** ¹
Code Line Count: **Code Count** ²
Project Originator: **The Candidate**
Supervisors: **Patrick Ferris, Anil Mad-**
havapeddy

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

1	Introduction	1
1.1	Motivation	1
1.2	Previous Work	1
1.3	Contributions	1
1.4	Dissertation Outline	1
2	Preparation	2
2.1	The Hazel Language	2
2.2	The Project	13
2.3	Starting Point	14
2.4	Requirement Analysis	15
2.5	Software Engineering Methodology	15
2.6	Legality	15
3	TEMPORARY: Implementation Plan	16
3.1	Cast Slicing	16
3.2	Search Procedure	25
4	Implementation	26
5	Evaluation	27
6	Conclusions	28
	Bibliography	29

A	Formal Semantics and Proofs	34
B	Another Appendix	35
	Index	37
	Project Proposal	38
	Description	38
	Starting Point	40
	Success Criteria	40
	Work Plan	42
	Resource Declaration	45

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 language, *XML* [4].

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

Figure 2.1: Syntax: *types* τ , *external expressions* e , *internal expressions* d . With x ranging over variables, u over hole names, σ over $x \rightarrow d$ *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

¹Come 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.

$$\boxed{\Gamma \vdash e \Rightarrow \tau} \quad e \text{ synthesises type } \tau \text{ under context } \Gamma$$

$$\begin{array}{c}
\text{SConst} \frac{}{\Gamma \vdash c \Rightarrow b} \quad \text{SVar} \frac{x : \tau \in \Gamma}{\Gamma \vdash x \Rightarrow \tau} \quad \text{SFun} \frac{\Gamma, x : \tau_1 \vdash e \Rightarrow \tau_2}{\Gamma \vdash \lambda x : \tau_1. e \Rightarrow \tau_1 \rightarrow \tau_2} \\
\\
\text{SApp} \frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \quad \tau_1 \blacktriangleright \rightarrow \tau_2 \rightarrow \tau \quad \Gamma \vdash e_2 \Leftarrow \tau_2}{\Gamma \vdash e_1(e_2) \Rightarrow \tau} \quad \text{SEHole} \frac{}{\Gamma \vdash \textcircled{u} \Rightarrow ?} \\
\\
\text{SNEHole} \frac{\Gamma \vdash e \Rightarrow \tau}{\Gamma \vdash \textcircled{e} \Rightarrow ?} \quad \text{SAsc} \frac{\Gamma \vdash e \Leftarrow \tau}{\Gamma \vdash e : \tau \Rightarrow \tau}
\end{array}$$

$$\boxed{\Gamma \vdash e \Leftarrow \tau} \quad e \text{ analyses against type } \tau \text{ under context } \Gamma$$

$$\begin{array}{c}
\text{AFun} \frac{\tau \blacktriangleright \rightarrow \tau_1 \rightarrow \tau_2 \quad \Gamma, x : \tau_1 \vdash e \Leftarrow \tau_2}{\Gamma \vdash \lambda x. e \Leftarrow \tau} \quad \text{ASubsume} \frac{\Gamma \vdash e \Rightarrow \tau \quad \tau \sim \tau'}{\Gamma \vdash e \Leftarrow \tau'}
\end{array}$$

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.

$$\boxed{\tau_1 \sim \tau_2} \quad \tau_1 \text{ is consistent with } \tau_2$$

$$\begin{array}{c}
\text{TCDyn1} \frac{}{? \sim \tau} \quad \text{TCDyn2} \frac{}{\tau \sim ?} \quad \text{TCRfl} \frac{}{\tau \sim \tau} \quad \text{TCFun} \frac{\tau_1 \sim \tau'_1 \quad \tau_2 \sim \tau'_2}{\tau_1 \rightarrow \tau_2 \sim \tau'_1 \rightarrow \tau'_2}
\end{array}$$

Figure 2.3: Type consistency

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

$$\boxed{\tau \blacktriangleright_{\rightarrow} \tau_1 \rightarrow \tau_2} \quad \tau \text{ has arrow type } \tau_1 \rightarrow \tau_2$$

$$\text{MADyn} \frac{}{? \blacktriangleright_{\rightarrow} ? \rightarrow ?} \quad \text{MAFun} \frac{}{\tau_1 \rightarrow \tau_2 \blacktriangleright_{\rightarrow} \tau_1 \rightarrow \tau_2}$$

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 synthesises type τ and elaborates to d

$$\begin{array}{c}
\text{ESConst} \frac{}{\Gamma \vdash c \Rightarrow b \rightsquigarrow c \dashv \emptyset} \quad \text{ESVar} \frac{x : \tau \in \Gamma}{\Gamma \vdash x \Rightarrow \tau \rightsquigarrow x \dashv \emptyset} \\
\\
\text{ESFun} \frac{\Gamma, x : \tau_1 \vdash e \Rightarrow \tau_2 \rightsquigarrow d \dashv \Delta}{\Gamma \vdash \lambda x : \tau_1. e \Rightarrow \tau_1 \rightarrow \tau_2 \rightsquigarrow \lambda x : \tau_1. d \dashv \Delta} \\
\\
\text{ESApp} \frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \quad \Gamma \vdash e_2 \Leftarrow \tau_2 \rightarrow \tau \rightsquigarrow d_1 : \tau'_1 \dashv \Delta_1 \quad \tau_1 \blacktriangleright \rightarrow \tau_2 \rightarrow \tau}{\Gamma \vdash e_1(e_2) \Rightarrow \tau \rightsquigarrow (d_1 \langle \tau'_1 \Rightarrow \tau_2 \rightarrow \tau \rangle)(d_2 \langle \tau'_2 \Rightarrow \tau_2 \rangle) \dashv \Delta_1 \cup \Delta_2} \\
\\
\text{ESEHole} \frac{}{\Gamma \vdash \langle \rangle^u \Rightarrow ? \rightsquigarrow \langle \rangle_{\text{id}(\Gamma)}^u \dashv u :: \langle \rangle[\Gamma]} \\
\\
\text{ESNEHole} \frac{\Gamma \vdash e \Rightarrow \tau \rightsquigarrow d \dashv \Delta}{\Gamma \vdash \langle e \rangle^u \Rightarrow ? \rightsquigarrow \langle d \rangle_{\text{id}(\Gamma)}^u \dashv \Delta, u :: \langle \rangle[\Gamma]} \\
\\
\text{ESAsc} \frac{\Gamma \vdash e \Leftarrow \tau \rightsquigarrow d : \tau' \dashv \Delta}{\Gamma \vdash e : \tau \Rightarrow \tau \rightsquigarrow d \langle \tau' \Rightarrow \tau \rangle \dashv \Delta}
\end{array}$$

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

$$\begin{array}{c}
\text{EAFun} \frac{\tau \blacktriangleright \rightarrow \tau_1 \rightarrow \tau_2 \quad \Gamma, x : \tau_1 \vdash e \Leftarrow \tau_2 \rightsquigarrow d : \tau'_2 \dashv \Delta}{\Gamma \vdash \lambda x. e \Leftarrow \tau \rightsquigarrow \lambda x : \tau_1. d : \tau_1 \rightarrow \tau'_2 \dashv \Delta} \\
\\
\text{EASubsume} \frac{e \neq \langle \rangle^u \quad e \neq \langle e' \rangle^u \quad \Gamma \vdash e \Rightarrow \tau' \rightsquigarrow d \dashv \Delta \quad \tau \sim \tau'}{\Gamma \vdash e \Leftarrow \tau \rightsquigarrow d : \tau' \dashv \Delta} \\
\\
\text{EAEHole} \frac{}{\Gamma \vdash \langle \rangle^u \Leftarrow \tau \rightsquigarrow \langle \rangle_{\text{id}(\Gamma)}^u : \tau \dashv u :: \tau[\Gamma]} \\
\\
\text{EANEHole} \frac{\Gamma \vdash e \Rightarrow \tau' \rightsquigarrow d \dashv \Delta}{\Gamma \vdash \langle e \rangle^u \Leftarrow \tau \rightsquigarrow \langle d \rangle_{\text{id}(\Gamma)}^u : \tau \dashv u :: \tau[\Gamma]}
\end{array}$$

Figure 2.5: Elaboration judgements

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

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

Figure 2.7: Identity substitution and substitution typing

$\Delta; \Gamma \vdash d : \tau$	d is assigned type τ
----------------------------------	-----------------------------

$\text{TACons} \frac{}{\Delta; \Gamma \vdash c : b}$	$\text{TAVar} \frac{x : \tau \in \Gamma}{\Delta; \Gamma \vdash x : \tau}$	$\text{TAFun} \frac{\Delta; \Gamma, x : \tau_1 \vdash d : \tau_2}{\Delta; \Gamma \vdash \lambda x : \tau_1. d : \tau_1 \rightarrow \tau_2}$
$\text{TAAp} \frac{\Delta; \Gamma \vdash d_1 : \tau_2 \rightarrow \tau \quad \Delta; \Gamma \vdash d_2 : \tau_2}{\Delta; \Gamma \vdash d_1(d_2) : \tau}$	$\text{TAEHole} \frac{u :: \tau[\Gamma'] \in \Delta \quad \Delta; \Gamma \vdash \sigma : \Gamma'}{\Delta; \Gamma \vdash \textcolor{red}{\textcircled{\text{d}}}_\sigma^u : \tau}$	
$\text{TANEHole} \frac{u :: \tau[\Gamma'] \in \Delta \quad \Delta; \Gamma \vdash d : \tau'}{\Delta; \Gamma \vdash \textcolor{red}{\textcircled{\text{d}}}_\sigma^u : \tau}$		$\text{TACast} \frac{\Delta; \Gamma \vdash d : \tau_1 \quad \tau_1 \sim \tau_2}{\Delta; \Gamma \vdash d \langle \tau_1 \Rightarrow \tau_2 \rangle : \tau_2}$
$\text{TACastError} \frac{\Delta; \Gamma \vdash d : \tau_1 \quad \tau_1 \text{ ground} \quad \tau_2 \text{ ground} \quad \tau_1 \neq \tau_2}{\Delta; \Gamma \vdash d \langle \tau_1 \Rightarrow ? \not\Rightarrow \tau_2 \rangle : \tau_2}$		

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).

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

$\text{GBase} \frac{}{b \text{ ground}}$	$\text{GDynFun} \frac{}{? \rightarrow ? \text{ ground}}$
--	--

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

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

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

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

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

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

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

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

$$\text{IEHole} \frac{}{\langle \rangle_\sigma^u \text{ indet}} \quad \text{INEHole} \frac{d \text{ final}}{\langle d \rangle_\sigma^u \text{ indet}} \quad \text{IAp} \frac{d_1 \neq d'_1 \langle \tau_1 \rightarrow \tau_2 \Rightarrow \tau_3 \rightarrow \tau_4 \rangle \quad d_1 \text{ indet} \quad d_2 \text{ final}}{d_1(d_2) \text{ indet}}$$

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

$$\text{ICastFun} \frac{\tau_1 \rightarrow \tau_2 \neq \tau_3 \rightarrow \tau_4 \quad d \text{ indet}}{d \langle \tau_1 \rightarrow \tau_2 \Rightarrow \tau_3 \rightarrow \tau_4 \rangle \text{ indet}} \quad \text{ICastError} \frac{d \text{ final} \quad \tau_1 \text{ ground} \quad \tau_2 \text{ ground} \quad \tau_1 \neq \tau_2}{d \langle \tau_1 \Rightarrow ? \neq \tau_2 \rangle \text{ indet}}$$

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.

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

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

Figure 2.10: Ground type matching

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

$$\begin{array}{c}
\text{ITFun} \frac{}{(\lambda x : \tau.d_1)(d_2) \longrightarrow [d_2/x]d_1} \quad \text{ITCastId} \frac{}{d\langle \tau \Rightarrow \tau \rangle \longrightarrow d} \\
\text{ITAppCast} \frac{\tau_1 \rightarrow \tau_2 \neq \tau'_1 \rightarrow \tau'_2}{d_1\langle \tau_1 \rightarrow \tau_2 \Rightarrow \tau'_1 \rightarrow \tau'_2 \rangle(d) \longrightarrow (d_1(d_2\langle \tau'_1 \Rightarrow \tau_1 \rangle))\langle \tau_2 \Rightarrow \tau'_2 \rangle} \\
\text{ITCast} \frac{\tau \text{ ground}}{d\langle \tau \Rightarrow ? \Rightarrow \tau \rangle \longrightarrow d} \quad \text{ITCastError} \frac{\tau_1 \neq \tau_2 \quad \tau_1 \text{ ground} \quad \tau_2 \text{ ground}}{d\langle \tau_1 \Rightarrow ? \Rightarrow \tau_2 \rangle \longrightarrow d\langle \tau_1 \Rightarrow ? \neq ? \rangle \tau_2} \\
\text{ITGround} \frac{\tau \blacktriangleright_{\text{ground}} \tau'}{d\langle \tau \Rightarrow ? \rangle \longrightarrow d\langle \tau \Rightarrow \tau' \Rightarrow ? \rangle} \quad \text{ITExpand} \frac{\tau \blacktriangleright_{\text{ground}} \tau'}{d\langle ? \Rightarrow \tau \rangle \longrightarrow d\langle ? \Rightarrow \tau' \Rightarrow \tau \rangle}
\end{array}$$

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

²Create a figure for this, maybe in appendix.

Context syntax:

$$\begin{aligned}
E &::= \circ \mid E(d) \mid d(E) \mid \langle E \rangle_\sigma^u \mid E\langle \tau \Rightarrow \tau \rangle \mid E\langle \tau \Rightarrow ? \nRightarrow \tau \rangle \\
\boxed{d = E[d]} &\quad d \text{ is the context } E \text{ filled with } d' \text{ in place of } \circ \\
\text{ECOouter} &\frac{}{d = \circ[d]} \quad \text{EApp1} \frac{d_1 = E[d'_1]}{d_1(d_2) = E(d_2)[d_1]} \quad \text{EApp2} \frac{d_2 = E[d_2]}{d_1(d_2) = d_1(E)[d'_2]} \\
\text{ECNEHole} &\frac{d = E[d']}{\langle d \rangle_\sigma^u = \langle E \rangle_\sigma^u[d']} \quad \text{ECCast} \frac{d = E[d']}{d\langle \tau_1 \Rightarrow \tau_2 \rangle = E\langle \tau_1 \Rightarrow \tau_2 \rangle[d']} \\
\text{ECCastError} &\frac{d = E[d']}{d\langle \tau_1 \Rightarrow ? \nRightarrow \tau_2 \rangle = E\langle \tau_1 \Rightarrow ? \nRightarrow \tau_2 \rangle[d']} \\
\boxed{d \mapsto d'} &\quad d \text{ steps to } d' \\
\text{Step} &\frac{d_1 = E[d_2] \quad d_2 \longrightarrow d'_2 \quad d'_1 = E[d'_2]}{d_1 \mapsto d'_1}
\end{aligned}$$

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 $\langle \rangle_\sigma^u$ being substituted

³Define Substitution in appendices?

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 theory⁴.

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

⁴See 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 Intelligence 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

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 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].*

¹Therefore types generally have output mode.

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

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 $\tau = \tau_1 \rightarrow \tau_2$, 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 holes⁴.

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 precise⁵ than e and synthesises τ under the context slice. For simplicity adding the notion of unsubstitutable, unnamed holes $\llbracket \cdot \rrbracket$:

³A 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.

⁵Ideally, least precise

Syntax:

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

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

$$\begin{array}{c} \text{SSConst} \frac{}{\Gamma \vdash c \Rightarrow b \parallel b[\cdot, b]} \quad \text{SSVar} \frac{x : \tau \in \Gamma}{\Gamma \vdash x \Rightarrow \tau \parallel \tau[x : \tau, x]} \\ \\ \text{SSFun} \frac{\Gamma, x : \tau_1 \vdash e \Rightarrow \tau_2 \parallel s[\gamma, \varepsilon]}{\Gamma \vdash \lambda x : \tau_1. e \Rightarrow \tau_1 \rightarrow \tau_2 \parallel (\tau_1[x : \tau_1, \text{⓪}] \rightarrow s[\gamma, \varepsilon])[\gamma \setminus \{x : \tau_1\}, \lambda x : \tau_1. \varepsilon]} \\ \\ \text{SSApp} \frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \parallel \varsigma_1 \quad \tau_1 \blacktriangleright \rightarrow \tau_2 \rightarrow \tau \quad \varsigma_1 \blacktriangleright \rightarrow (\varsigma_2 \rightarrow s[\gamma, \varepsilon])[\gamma_1, \varepsilon_1] \quad \Gamma \vdash e_2 \Leftarrow \tau_2}{\Gamma \vdash e_1(e_2) \Rightarrow \tau \parallel s[\gamma_1, \varepsilon_1](\text{⓪})} \quad \text{SSEHole} \frac{}{\Gamma \vdash \text{⓪}^u \Rightarrow ? \parallel ?[\cdot, \text{⓪}]} \\ \\ \text{SSNEHole} \frac{\Gamma \vdash e \Rightarrow \tau}{\Gamma \vdash (e)^u \Rightarrow ? \parallel ?[\cdot, \text{⓪}]} \quad \text{SAsc} \frac{\Gamma \vdash e \Leftarrow \tau}{\Gamma \vdash e : \tau \Rightarrow \tau \parallel \tau[\cdot, \text{⓪} : \tau]} \end{array}$$

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

Defining a slice matching relation analogously to type matching $\blacktriangleright \rightarrow$.

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 $\tau_1 \rightarrow ?$. 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.

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:

$$\begin{array}{c}
\text{ESApp} \frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \parallel \varsigma_1 \quad \tau_1 \blacktriangleright \rightarrow \tau_2 \rightarrow \tau \quad \varsigma_1 \blacktriangleright \rightarrow (\varsigma_2 \rightarrow \varsigma)[\gamma_1, \varepsilon_1] \quad \Gamma \vdash e_1 \Leftarrow \tau_2 \rightarrow \tau \rightsquigarrow d_1 : \tau'_1 \dashv \Delta_1 \parallel \varsigma'_1 \quad \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 \quad \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'} \quad e \text{ analyses against type } \tau \text{ and elaborates to } d \text{ of consistent type } \tau' \text{ with slice } \varsigma' \\
\\
\text{SEAFun} \frac{\tau \blacktriangleright \rightarrow \tau_1 \rightarrow \tau_2 \quad \Gamma, x : \tau_1 \vdash e \Leftarrow \tau_2 \rightsquigarrow d : \tau'_2 \dashv \Delta \parallel s[\gamma, \varepsilon]}{\Gamma \vdash \lambda x. e \Leftarrow \tau \rightsquigarrow \lambda x : \tau_1. d : \tau_1 \rightarrow \tau'_2 \dashv \Delta \parallel (\tau_1[x : \tau_1, \emptyset] \rightarrow s[\gamma, \varepsilon])[\gamma \setminus \{x : \tau_1\}, \lambda x. \varepsilon]} \\
\\
\text{SEASubsume} \frac{e \neq \emptyset^u \quad e \neq (e')^u \quad \Gamma \vdash e \Rightarrow \tau' \parallel \varsigma' \quad \Gamma \vdash e \Rightarrow \tau' \rightsquigarrow d \dashv \Delta \quad \tau \sim \tau'}{\Gamma \vdash e \Leftarrow \tau \rightsquigarrow d : \tau' \dashv \Delta \parallel \varsigma'}
\end{array}$$

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. Simply speaking, a join of slices is just the union of the code they point at. Semantically, in the above formulation it is a bit tedious to define, but for any rule using join, the joined slice could be constructed by placing the slices in their appropriate spots and merging overlapping slices (with concrete terms replacing holes).

Implementation-wise, slices will probably be code locations, and will be more granular than expressions. 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. $\tau_1 \rightarrow \tau_2 \blacktriangleright_{\text{ground}}? \rightarrow ?$. 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.

Cast Splitting

There are only 3 rules that non-trivially modify casts: ITApp-Cast, ITGround, ITEXpand. For ITGround and ITEXpand, it makes sense to keep the cast slices for τ' the same as τ .⁶ ITAp-

⁶Justify this. Maybe casting to ground type was not in code so the cast should not highlight anything, but instead record it's *dependencies*?

pCast requires decomposing the function cast, luckily slices recursively include type slices of their type components.

$$\text{ITAppCast} \frac{\tau_1 \rightarrow \tau_2 \neq \tau'_1 \rightarrow \tau'_2}{d_1 \langle \varsigma_1 \rightarrow \varsigma_2 \Rightarrow \varsigma'_1 \rightarrow \varsigma'_2 \rangle (d_2) \rightarrow (d_1 (d_2 \langle \varsigma'_1 \Rightarrow \varsigma_1 \rangle) \langle \varsigma_2 \Rightarrow \varsigma'_2 \rangle)}$$

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 matching⁷). Casts between compound types may be decomposed by the cast calculus. See gradualizer dynamic semantics [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).

⁷See the gradualizer for intuition and direction of consistency casts

3.1.2 Implementation Structure

Research Into Hazel.

3.1.3 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 **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.

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 [22] might be useful.

3.2 Search Procedure

Track dynamic type refinement via provenances, so the same dynamic type is always refined in the same manner [23].

Look into Idris etc. etc.

Chapter 4

Implementation

Implementation Here.

Chapter 5

Evaluation

Evaluation Here.

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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/10.1016/0020-0190(88)90054-3). [Online]. Available: [http://dx.doi.org/10.1016/0020-0190\(88\)90054-3](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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/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] 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](https://doi.org/10.1145/1159876.1159880). [Online]. Available: <http://dx.doi.org/10.1145/1159876.1159880>.
 - [23] E. Zhao, R. Maroof, A. Dukkupati, A. Blinn, Z. Pan, and C. Omar, “Total type error localization and recovery with holes,” *Proceedings of the ACM on Programming Languages*, vol. 8, no. POPL, pp. 2041–2068, Jan. 2024,

ISSN: 2475-1421. DOI: [10.1145/3632910](https://doi.org/10.1145/3632910). [Online]. Available: <http://dx.doi.org/10.1145/3632910>.

- [24] “Hazel project website. ”[Online]. Available: <https://hazel.org/>.
- [25] “Hazel source code. ”[Online]. Available: <https://github.com/hazeltgrove/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

Core Hazel, [3](#)

Core Hazel syntax, [3](#)

External language type system, [3](#)

Hazel, [2](#)

Project Proposal

Description

This project will add some features to the Hazel language [24]. 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:

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

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 [24] 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 -i Hazel translator/annotator which works with well-typed OCaml. A Hazel type fuzzer.
5. Collection of an ill-typed OCaml corpus.
Tools: OCaml -i Hazel translator which works with ill-typed OCaml. *This would NOT be expected to be an implicitly typed Hazel front-end which maintains desirable 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 [25].
- 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.