Max Carroll

Type Error Debugging in Hazel

Computer Science Tripos, Part II

Sidney Sussex College College
University of Cambridge
October 28, 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:	**1225** [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 $? \blacktriangleright_{\to} ? \to ?$.

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

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

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 judgement.

Notice that cast errors – casts between distinct ground types – are well typed, where ground types are base types or one-level unrollings of the dynamic type (each being the *least specific* type for each compound type).

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.

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

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

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

$$\text{ESApp}\ \frac{\begin{array}{cc} \Gamma \vdash e_1 \Rightarrow \tau_1 & \tau_1 \blacktriangleright_{\to} \tau_2 \to \tau \\ \Gamma \vdash e_1 \Leftarrow \tau_2 \to \tau \rightsquigarrow d_1 : \tau_1' \dashv \Delta_1 & \Gamma \vdash e_1 \Leftarrow \tau_2 \rightsquigarrow d_2 : \tau_2' \dashv \Delta_2 \end{array}}{\Gamma \vdash e_1(e_2) \Rightarrow \tau \rightsquigarrow (d_1\langle \tau_1 \Rightarrow \tau_2 \to \tau \rangle)(d_2\langle \tau_2' \Rightarrow \tau_2 \rangle) \dashv \Delta_1 \cup \Delta_2}$$

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

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

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

$\boxed{\Gamma \vdash e \Leftarrow \tau_1 \rightsquigarrow d : \tau_2 \dashv \Delta}$ e analyses against type τ and elaborates to d of consistent type τ_2

$$\text{EAFun}\ \frac{\begin{array}{c} \tau \blacktriangleright_{\to} \tau_1 \to \tau_2 \\ \Gamma, x : \tau_1 \vdash e \Leftarrow \tau_2 \rightsquigarrow d : \tau_2 \dashv \Delta \end{array}}{\Gamma \vdash \lambda x.e \Leftarrow \tau \rightsquigarrow \lambda x : \tau_1.d : \tau_1 \to \tau_2' : \Delta \dashv}$$

$$\text{EASubsume}\ \frac{\begin{array}{cc} e \neq (\!|\!)^u & e \neq (\!|e'|\!)^u \\ \Gamma \vdash e \Rightarrow \tau' \rightsquigarrow d \dashv \Delta & \tau \sim \tau' \end{array}}{\Gamma \vdash e \Leftarrow \tau \rightsquigarrow d : \tau' \dashv \Delta}$$

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

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

Figure 2.5: Elaboration judgements

- Indeterminate forms – Irreducible terms containing holes or are casts errors. Substitution of holes may make these reducible.
- Final – All final forms.

The small-step contextual dynamics [8] is defined on the in-

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

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

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

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

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

Figure 2.6: Type assignment judgement for *internal expressions*

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

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

Figure 2.7: Identity substitution and substitution typing

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

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

Figure 2.8: Ground types

ternal 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.

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

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

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

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

$$\text{IEHole}\,\frac{}{⦇⦈^{u}_{\sigma}\ \text{indet}} \qquad \text{INEHole}\,\frac{d\ \text{final}}{⦇d⦈^{u}_{\sigma}\ \text{indet}} \qquad \text{IAp}\,\frac{\begin{array}{c} d_1 \neq d_1'\langle \tau_1 \to \tau_2 \Rightarrow \tau_3 \to \tau_4\rangle \\ d_1\ \text{indet} \qquad d_2\ \text{final} \end{array}}{d_1(d_2)\ \text{indet}}$$

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

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

Figure 2.9: Final forms

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.[2]

Casts are associative, so bracketing is omitted.

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

$\boxed{\tau \blacktriangleright_{\text{ground}} \tau'}$ 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

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 [9], and later others proposed dynamic program slicing [10] and error slicing [11]. 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

Context syntax:

$$E ::= \circ \mid E(d) \mid d(E) \mid (\!|E|\!)^u_\sigma \mid E\langle\tau\Rightarrow\tau\rangle \mid E\langle\tau\Rightarrow ?\not\Rightarrow\tau\rangle$$

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

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

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

$$\text{ECCastError}\,\frac{d = E[d']}{d\langle\tau_1\Rightarrow ?\not\Rightarrow\tau_2\rangle = E\langle\tau_1\Rightarrow ?\not\Rightarrow\tau_2\rangle[d']}$$

$\boxed{d \mapsto d'}$ 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

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. [12] 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 analogously to contextual modal type theory [13][3].

[3]Explore

Further, the use of dynamic types, casts, and the more general notion of holes will allow better handling of non-parametric 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.[4]

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. [12] 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.

[4]Talk more deeply about CMTT and how hole substitution is efficient in Hazel.

2.4 Requirement Analysis

2.5 Software Engineering Methodology

2.6 Legality

Chapter 3

TEMPORARY: Implementation Plan

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. [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] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] "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

Project Proposal

Description

This project will add some features to the Hazel language [14]. 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. [12]. 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 [15], error and dynamic program slicing [10], [11], 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 [14] 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 [16].

- 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.