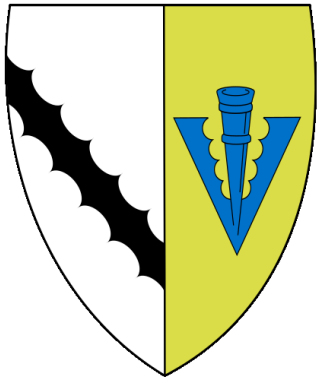
Max Carroll

Type Error Debugging in Hazel

Computer Science Tripos, Part II



Sidney Sussex College College
University of Cambridge
February 28, 2025

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:	**8012 (errors:4)** [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

1 Introduction **1**
- 1.1 Related Work . . . 3
- 1.2 Dissertation Outline . . . 3

2 Preparation **5**
- 2.1 Background Knowledge . . . 5
- 2.2 Starting Point . . . 23
- 2.3 Requirement Analysis . . . 23
- 2.4 Software Engineering Methodology . . . 23
- 2.5 Legality . . . 24

3 TEMPORARY: Implementation Plan **25**
- 3.1 Cast Slicing . . . 25
- 3.2 Search Procedure . . . 34
- 3.3 Type Error Witnesses Interaction in General . . . 41

4 Implementation **48**
- 4.1 Type Slicing Theory . . . 49
- 4.2 Cast Slicing Theory . . . 49
- 4.3 Proofs . . . 49
- 4.4 Type Slicing Implementation . . . 49
- 4.5 Cast Slicing Implementation . . . 50
- 4.6 Indeterminate Evaluation . . . 50
- 4.7 Evaluation Stepper . . . 51

4.8 Search Procedure 51

5 Evaluation **52**
5.1 Goals 52
5.2 Program Corpus Collection 52
5.3 Effectiveness Analysis 53
5.4 Critical Analysis 53
5.5 Performance Analysis 55

6 Conclusions **56**
6.1 Conclusion 56
6.2 Further Directions 56

Bibliography **58**

A Hazel Formal Semantics **67**
A.1 Syntax 67
A.2 Static Type System 68
A.3 Dynamics 71

B Another Appendix **75**

Index **77**

Project Proposal **78**
Description 78
Starting Point 80
Success Criteria 80
Work Plan 82
Resource Declaration 85

Chapter 1

Introduction

Software bugs are an inherent part of programming, often leading to unexpected behaviour and system failures. Debugging these errors is a *time-consuming process* taking between 20-60% of active work time [4], with programmers spending a *highly skewed* proportion of their time identifying and resolving a small proportion of bugs [2].

Type systems aim to alleviate some of this burden by classifying expressions and operations that are allowed to work on them. This may be done *statically* at compile time or *dynamically* during runtime. The expressions not conforming to the type system manifest themselves as *type errors*.

In static typing, blame for type errors are typically localised to a *single* location in the code. However, this localisation may be misleading, as the actual cause of the error might be rooted in a broader context, for example in OCaml 65% of type errors related to *multiple* locations [52]. This is a particularly prevalent issue in *type inferred* languages.

In dynamic typing, type errors are often missed as they only appear during runtime with specific inputs. However, a dynamic type error can be more intuitive due to it being accompanied by an *evaluation trace* demonstrating concretely

why values are *not* consistent with their expected type.

This project seeks to enhance the debugging experience in Hazel [21], a functional locally inferred and gradually typed research language under active development at the University of Michigan.

INTRODUCE HAZEL & IT'S VISION HERE WITH IT'S BASIC UNUSUAL FEATURES

I introduce two novel features to improve user comprehension of type errors, *type slicing* and *cast slicing*; additionally, mathematical foundations have been devised for these by building upon the Hazel Calculus [31]. Further, I implement a *type error witness search procedure* based upon a similar idea implemented for a subset of OCaml by Seidel et al. [43]:

- **Type slicing** highlights larger sections of code that contribute to an expression having a required type. Hence, a *static type error* location can be selected and the context enforcing the erroneous type revealed.

- **Cast slicing** propagates type slice information throughout evaluation, allowing the context of *runtime casts* to be examined. Hence, a *runtime type error* (that is, a *cast error*) can be selected to reveal the context enforcing it's expected type.

- The **type error witness search procedure** finds inputs to expressions that will cause a *cast error* upon evaluation, these accompanied with their execution trace are referred to as *type error witnesses*. Hence, dynamic type errors can be found automatically, and *concrete* type witnesses can be found for known static type errors.

These three features work well together to allow both static and dynamic type errors to be located and explained to a greater extent than in any existing languages. Arguably, these

explanations are intuitive, and should help reduce debugging times and aid in students understanding type systems.

For example, here is a walk-through for how a simple error could be diagnosed using these three features.

MAP function? show a static error version

1.1 Related Work

There has been extensive research into attempting to understand *what* is needed [26], *how* developers fix bugs [9], and a plethora of compiler *improvements* and *tools* **add citations here, primarily functional language tools**. This project builds upon this body of research in new ways focusing on the Hazel language, which is itself a research project being taken in various directions but generally as a *teaching language* for students.

To my knowledge the ideas of *type slicing* and **cast slicing** are novel. However, they do output *program slices* which were originally explored by Weiser [50], though my usage is more similar to *dynamic program slicing* [25] but with type characteristic rather than evaluation characteristic, in a sense similar *type error slicing* [47]. As Hazel is an expression-based language giving expression slices just the same as in Functional as Hazel is an expression-based language.

The *type witness search procedure* is based upon Seidel et al. [43], though there are significant differences in workings due to my use of Hazel and various extensions as compared to the subset of OCaml used by Seidel et al.

1.2 Dissertation Outline

Explain structure – what is covered in each chapter briefly.

Where should a worked example of these features be placed?

Chapter 2

Preparation

In this chapter I present the technical background knowledge for this project: an introduction to the type theory for understanding Hazel's core semantics, an overview of Hazel implementation, and notes on non-determinism (as the type witness search procedure is non-deterministic). Following this, I present my software engineering methodology.

2.1 Background Knowledge

2.1.1 Static Type Systems

A *type system* is a lightweight formal mathematical method which categorises values into *types* and expressions into types that evaluate to values of the same type. It is effectively a static *approximation* to the runtime behaviour of a language.

Syntax

Trivial, probably not needed? Cite BNF grammars etc. All Ib stuff. Maybe briefly show examples of Lambda calculus-like syntax?

Judgements & Inference Rules

A *judgement*, J, is an assertion about *expressions* in a language [18]. For example:

- Exp e – e is an *expression*
- $n : \texttt{int}$ – n has type `int`
- $e \Downarrow v$ – e evaluates to *value* v

While an *inference rule* is a collection of judgements $J, J_1, \ldots, J_n$:

$$\frac{J_1 \quad J_2 \quad \ldots \quad J_n}{J}$$

Representing the *rule* that if the *premises*, $J_1, \ldots, J_n$ are true then the conclusion, J, is true. When the collection of premises is empty, it is an *axiom* stating that the judgement is *always* true. Truth of a judgement J can be assessed by constructing a *derivation*, a tree of rules where it's leaves are axioms. It is then possible to define a judgement as the largest judgement that is *closed* under a collection of rules. This gives the result that a judgement J is true *if and only if* it has a derivation.

Properties on expressions can be proved using *rule induction*, if a property is *preserved* by every rule for a judgement, and true for it's axioms, then the property holds whenever the judgement is derivable.

A *hypothetical judgement* is a judgement written as:

$$J_1, \ldots, J_n \vdash J$$

is true if J is derivable when additionally assuming each J_i are axioms. Often written $\Gamma \vdash J$ and read J *holds under context* Γ. Hypothetical judgements can be similarly defined inductively via *rules*.

Defining a Type System

A typical type system can be expressed by defining the following hypothetical judgement form $\Gamma \vdash e : \tau$ read as *the expression e has type τ under typing context Γ* and referred as a *typing judgement*. Here, $e : \tau$ means that expression e has type τ. A *typing context*, Γ, is a list of types for variables $x_1 : \tau_1, \ldots, x_n : \tau_2$. For example the SLTC[1] [38, ch. 9] has a typing rule for lambda expression and application as follows:

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\ e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1(e_2) : \tau_2}$$

Meaning, $x.e$ has type $\tau_1 \to \tau_2$ if e has type τ_2 under the extended context additionally assuming that x has type τ_1. And, $e_1(e_2)$ has type τ_2 if e_1 is a function of type $\tau_1 \to \tau_2$ and it's argument e_2 has type τ_1.

2.1.2 Product & Labelled Sum Types

Briefly demonstrate. Link to TAPL *Variants* and products

Dynamic Type Systems

Dynamic Typing has purported strengths allowing rapid development and flexibility, evidenced by their popularity [33, 46]. Of particular relevance to this project, execution traces are known to help provide insight to errors [11], yet statically typed languages remove the ability to execute programs with type errors, whereas dynamically typed languages do not.

A *dynamically typed system* can be implemented and represented semantically by use of dynamic *type tags* and a *dynamic*

[1] Simply typed lambda calculus.

type[2] [1]. Then, runtime values can have their type checked at runtime and *cast* between types. This suggests a way to encode dynamic typing via *first-class*[3] cast expressions which maintain and enforce runtime type constraints alongside a dynamic type written `?`.

Cast expressions can be represented in the syntax of expression by $e\langle\tau_1 \Rightarrow \tau_2\rangle$ for expression e and types τ_1, τ_2, encoding that e has type τ_1 and is cast to new type τ_2. An intuitive way to think about these is to consider two classes of casts:

- *Injections* – Casts *to* the dynamic type $e\langle\tau \Rightarrow ?\rangle$. These are effectively equivalent to type tags, they say that e has type τ but that it should be treat dynamically.
- *Projections* – Casts *from* the dynamic type $e\langle ? \Rightarrow \tau\rangle$. These are type requirements, for example the add operator could require inputs to be of type `int`, and such a projection would force any dynamic value input to be cast to `int`.

Then when *injections* meet *projections* meet, $v\langle\tau_1 \Rightarrow ? \Rightarrow \tau_2\rangle$, representing an attempt to perform a cast $\langle\tau_1 \Rightarrow \tau_2\rangle$ on v. We check the cast is valid and perform if so:

$$\frac{\tau_1 \text{ is castable to } \tau_2}{v\langle\tau_1 \Rightarrow ? \Rightarrow \tau_2\rangle \mapsto v'} \qquad \frac{\tau_1 \text{ is not castable to } \tau_2}{v\langle\tau_1 \Rightarrow ? \Rightarrow \tau_2\rangle \mapsto v\langle\tau_1 \Rightarrow ? \not\Rightarrow \tau_2\rangle}$$

Compound type casts can be broken down during evaluation upon usage of such constructs. For example, applying v to a *wrapped*[4] functions could decompose the cast to separately cast the applied argument and then the result. Inspired by,

[2] Not necessarily needed for implementation, but is useful when reasoning about dynamic types within a formal type system or when considering types within a *static* context.

[3] Directly represented in the language syntax as expressions.

[4] Wrapped in a cast between function types.

semantic casts [15] in *contract* systems [14]:

$$(f\langle\tau_1 \to \tau_2 \Rightarrow \tau_1' \to \tau_2'\rangle)(v) \mapsto (f(v\langle\tau_1' \Rightarrow \tau_1\rangle)\langle\tau_2 \Rightarrow \tau_2'\rangle)$$

Or if f has the dynamic type:

$$(f\langle ? \Rightarrow \tau_1' \to \tau_2'\rangle)(v) \mapsto (f(v\langle\tau_1' \Rightarrow ?\rangle)\langle ? \Rightarrow \tau_2'\rangle)$$

Then direction of the casts reflects the *contravariance* [37, ch. 2] of functions[5] in their argument. See that the cast $\langle\tau_1' \Rightarrow \tau_1\rangle$ on the argument is *reversed* with respect to the original cast on f. This makes sense as we must first cast the applied input to match the actual input type of the function f.

Hence, casts around functions (type information) will be moved to the actual arguments at runtime, meeting with casts casts on the argument, resulting in a cast error or a successful casts.

Gradual Type Systems

A *gradual type system* [45, 44] combines static and dynamic typing. Terms may be annotated as dynamic, marking regions of code omitted from type-checking but still *interoperable* with static code. For example, the following type checks:

```
let x : ? = 10;  // Dynamically typed
x ++ "str"       // Statically typed
```

Where `++` is string concatenation expecting inputs to be `string`. But would then cause a runtime *cast error* when attempting to calculate `10 ++ "str"`.

It does this by representing casts as expressed previously. The language is split into two parts:

[5] A bifunctor.

- The *external language* – where static type checking is performed which allows annotating expressions with the dynamic type.
- The *internal language* – where evaluation and runtime type checking is performed via cast expressions.[6] The example above would reduce to a *cast error*[7]:

$$10\langle \texttt{int} \Rightarrow ? \not\Rightarrow \texttt{string} \rangle \texttt{ ++ "str"}$$

For type checking, a *consistency* relation $\tau_1 \sim \tau_2$ is introduced meaning *types* τ_1, τ_2 *are consistent*. This is a weakening of the type equality requirements in normal static type checking, allowing consistent types to be used additionally.

Consistency must satisfy a few properties: that the dynamic type is consistent with every type, $\tau \sim ?$ for all types τ, that $\sim$ is reflexive and symmetric, and two concrete types[8] are consistent iff they are equal[9]. A typical definition would be like:

$$\frac{}{\tau \sim ?} \qquad \frac{}{\tau \sim \tau} \qquad \frac{\tau_1 \sim \tau_2}{\tau_2 \sim \tau_1} \qquad \frac{\tau_1 \sim \tau_1' \qquad \tau_2 \sim \tau_2}{\tau_1 \to \tau_2 \sim \tau_1' \to \tau_2'}$$

This is very similar to the notion of *subtyping* [38, ch. 15] with a *top* type $\top$, but with symmetry instead of transitivity.

Then typing rules can be written to use consistency instead of equality. For example, application typing:

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2' \qquad \tau_1 \blacktriangleright_{\to} \tau_2 \to \tau \qquad \tau_2 \sim \tau_2'}{\Gamma \vdash e_1(e_2) : \tau_2'}$$

[6] i.e. the proposed *dynamic type system* above.

[7] Cast errors now represented with a strike-through and in red. From here-on they are considered as first-class constructs.

[8] No sub-parts are dynamic.

[9] e.g. $\tau_1 \to \tau_2 \sim \tau_1' \to \tau_2'$ iff $\tau_1 = \tau_1'$ and $\tau_2 = \tau_2'$ when $\tau_1, \tau_1', \tau_2, \tau_2'$ don't contain $?$.

Where $\blacktriangleright_{\to}$ is a pattern matching function to extract the argument and return types from a function type.[10] Intuitively, $e_1(e_2)$ has type τ_2' if e_1 has type $\tau_1' \to \tau_2'$ or $?$ and e_2 has type τ_1 which is consistent with τ_1' and hence is assumed that it can be passed into the function.

But, for evaluation to work the static type information needs to be encoded into casts to be used in the dynamic internal language, for which the evaluation semantics are defined. This is done via *elaboration*, similarly to Harper and Stone's approach to defining (globally inferred) Standard ML [20] by elaboration to an explicitly typed internal language XML [19]. The *elaboration judgement* $\Gamma \vdash e \rightsquigarrow e' : \tau$ read as: external expression e is elaborated to internal expression d with type τ under typing context Γ. For example we need to insert casts around function applications:

$$\frac{\Gamma \vdash e_1 \rightsquigarrow d_1 : \tau_1 \quad \Gamma \vdash e_2 \rightsquigarrow d_2 : \tau_2' \qquad \tau_1 \blacktriangleright_{\to} \tau_2 \to \tau \quad \tau_2 \sim \tau_2'}{\Gamma \vdash e_1(e_2) : \tau \rightsquigarrow (d_1\langle \tau_1 \Rightarrow \tau_2 \to \tau \rangle)(d_2\langle \tau_2' \Rightarrow \tau_2 \rangle) : \tau}$$

If, e_1 elaborates to d_1 with type $\tau_1 \sim \tau_2 \to \tau$ and e_2 elaborates to τ_2' with $\tau_2 \sim \tau_2$ then we place a cast[11] on the function d_1 to $\tau_2 \to \tau$ and on the argument d_2 to the function's expected argument type τ_2 to perform runtime type checking of arguments. Intuitively, casts must be inserted whenever type consistency is used, though the casts to insert are non-trivial [7].

The runtime semantics of the internal expression is that of the *dynamic type system* discussed above (2.1.2). A cast is determined to succeed iff the types are *consistent*.

The *refined criteria* for gradual typing [45] also provides an additional property for such systems to satisfy, the *gradual*

[10] This makes explicit the implicit pattern matching used normally.

[11] This cast is required, as if $\tau_1 = ?$ then we need a cast to realise that it is even a function. Otherwise $\tau_1 = \tau_2 \to \tau$ and the cast is redundant.

guarantee, formalising the intuition that adding and removing annotations should *not* change the *behaviour* of the program except for catching errors either dynamically and statically.

Bidirectional Type Systems

A *bidirectional type system* [12] takes on a more algorithmic definition of typing judgements, being more intuitive to implement. They also allow some amount of local type inference [39], allowing programmers to omit *type annotations*, instead type information. Global type inference systems [38, ch. 22] can be *difficult to implement*, often via constraint solving [36, ch. 10], and difficult or impossible to *balance* with complex language features, for example global inference in System F (2.1.2) is undecidable [51].

This is done in a similar way to annotating logic program by specifying the *mode* [49, p. 123] of the type parameter in a typing judgement, distinguishing when it is an *input* (type checking) and when it is an *output* (type synthesis).

We express this with two judgements:

$$\Gamma \vdash e \Rightarrow \tau$$

Read as: *e synthesises a type* τ *under typing context* Γ. Type τ is an *output*.

$$\Gamma \vdash e \Leftarrow \tau$$

Read as: e analyses against a type τ under typing context Γ. Type τ is an *input*

When designing such a system care must be taken to ensure *mode correctness* [5]. Mode correctness ensures that input-output dataflow is consistent such that an input never needs to be *guessed*. For example the following function application rule is *not* mode correct:

$$\frac{\Gamma \vdash e_1 \Leftarrow \tau_1 \to \tau_2 \qquad \Gamma \vdash e \Leftarrow \tau_1}{\Gamma \vdash e_1(e_2) \Leftarrow \tau_2}$$

We try to *check* e_2 with input τ_1[12] which is *not known* from either an *output* of any premise nor from the *input* to the conclusion, τ_2. On the other hand, the following *is* mode correct:

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 \Leftarrow \tau_1}{\Gamma \vdash e_1(e_2) \Leftarrow \tau_2}$$

Where τ_1 is now known, being *synthesised* from the premise $\Gamma \vdash e_1 \Rightarrow \tau_1 \to \tau_2$. As before, τ_2 is known as an input in the conclusion $\Gamma \vdash e_1(e_2) \Leftarrow \tau_2$.

Such languages will typically have three obvious rules. First, we should have that variables can synthesise their type, after all it is accessible from the typing context Γ:

$$\text{Var}\frac{x : \tau \in \Gamma}{\Gamma \vdash x \Rightarrow \tau}$$

And annotated terms can synthesise their type by just looking at the annotation $e : \tau$ and checking the annotation is valid:

$$\text{Annot}\frac{\Gamma \vdash e \Leftarrow \tau}{\Gamma \vdash e : \tau \Rightarrow \tau}$$

Finally, when we check against a type that we can synthesise a type for, variables for example. It would make sense to be able to *check* e against this same type τ; we can synthesise it, so must be able to check it. This leads to the subsumption rule:

$$\text{Subsumption}\frac{\Gamma \vdash e \Rightarrow \tau' \qquad \tau = \tau'}{\Gamma \vdash e \Leftarrow \tau}$$

Contextual Modal Type Theory

Not *hugely* relevant really...

System F

Very brief explanation with less/no maths

[12] Highlighted in red as an error.

Recursive Types

Very brief explanation with less/no maths

2.1.3 The Hazel Calculus

Hazel is a language that allows the writing of incomplete programs, evaluating them, and evaluating around dynamic errors.[13]

The core calculus [31] is a gradually and bidirectionally typed lambda calculus. Therefore it has a locally inferred bidirectional *external language* with the dynamic type ? elaborated to an explicitly typed *internal language* including cast expressions.

The full semantics are documented in the Hazel Formal Semantics appendix A, but only rules relevant to addition of *holes* are discussed in this section. The combination of gradual and bidirectional typing system is itself non-trivial and only particularly notable changes are mentioned here. The intuition should be clear from the previous gradual and bidirectional typing sections.[14]

Hazel's primary addition is the addition of *expression holes*, which can both be typed and have evaluation proceed around them seamlessly. This also allows the evaluation of ill-typed expressions by placing them in holes.

Syntax

The syntax, in Fig. 2.1, consists of *types* τ including the dynamic type ?, *external expressions* e including (optional) annotations, *internal expressions* d including cast expressions. The

[13] Among other features, like an structure editor with syntactically meaningless states, and various learning aids.

[14] The difficulties combining gradual and bidirectional typing are largely orthogonal to adding holes.

external language is *bidirectionally typed*, and therefore is a locally inferred surface syntax for the language, and is statically elaborated to (explicitly typed) *internal expressions*.

Notating $(\!|\,|\!)^u$ or $(\!|e|\!)^u$ for empty and non-empty holes respectively, where u is the *metavariable* or name for a hole. Internal expression holes, $(\!|\,|\!)^u_\sigma$ or $(\!|e|\!)^u_\sigma$, also maintain an environment σ mapping variables x to internal expressions d. These internal holes act as *closures*, recording which variables have been substituted during evaluation.[15]

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

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

We have a bidirectionally static semantics for the *external language*, giving the bidirectional typing judgements: $\Gamma \vdash e \Rightarrow \tau$ and $\Gamma \vdash e \Leftarrow \tau$. Holes synthesise the *dynamic type*, a natural choice made possible by the use of gradual types:

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

One notable consequence of combining gradual and bidirectional typing is that the *subsumption rule* in bidirectional typ-

[15] This is required, as the term inside the hole may contain one of these variables

ing is naturally extended to allow subsuming any *consistent* types:

$$\text{ASubsume}\ \frac{\Gamma \vdash e \Rightarrow \tau' \qquad \tau \sim \tau'}{\Gamma \vdash e \Leftarrow \tau}$$

Of course e should type check against τ if it can synthesise a consistent type as the goal of type consistency is that we may type check terms as if they were of the consistent type.

The remaining rules are detailed in Fig. A.2, with *consistency* relation $\sim$ in Fig. A.3 and (fun) type matching relation, $\blacktriangleright_{\rightarrow}$ in Fig. A.4.

Internal Language

The internal language is non-bidirectionally typed and requires an extra *hole context* Δ mapping hole *metavariables* u to their checked types[16] and the type context under which notated $u :: \tau[\Gamma]$, notation borrowed from contextual modal type theory (CMTT) [29].[17]

The type assignment judgement $\Delta; \Gamma \vdash d : \tau$ means that d has type τ under typing and hole contexts Γ, Δ. The rules for holes take their types from the hole context and ensure that the hole environment substitutions σ are well-typed[18]:

$$\text{TAEHole}\ \frac{u :: \tau[\Gamma'] \in \Delta \qquad \Delta; \Gamma \vdash \sigma : \Gamma'}{\Delta; \Gamma \vdash (\!|\,|\!)^{u}_{\sigma} : \tau}$$

Hazel is proven to preserve typing; a well-typed external expression will elaborate to a well-typed internal expression which is consistent to the external type. Hence, there is no need for an algorithmic definition of the internal language typing.

[16] As originally required by the external language

[17] Hole contexts corresponding to *modal contexts*, hole names with *metavariables*, and holes with *metavariable closures* (on environments σ).

[18] With respect to the original typing context for the hole.

Full rules in Fig. A.6. Formally speaking these define categorical judgements [35]. Additionally, ground types and a matching function are defined in Figs. A.8 & A.11, and typing of hole environments/substitution in Fig. A.7

Elaboration

Cast insertion requires an elaboration to the *internal language*, so must output an additional context for holes Δ. The judgements are notated:

$$\Gamma \vdash e \Rightarrow \tau \rightsquigarrow d \dashv \Delta$$

external expression e which synthesises type τ under type context Γ is elaborated to internal expression d producing hole context Δ.

$$\Gamma \vdash e \Leftarrow \tau \rightsquigarrow d : \tau' \dashv \Delta$$

external expression e which type checks against type τ under type context Γ is elaborated to internal expression d of consistent type τ' producing hole context Δ.

The elaboration judgements for holes must add the hole to the output hole context. And they will elaborate to holes with the default empty environment $\sigma = (\Gamma)$, i.e. no substitutions.

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

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

When elaborating a type *checked* hole, this checked type is used. Typing them instead as ? would imply type information being lost[19].

The remaining elaboration rules are stated in Fig. A.5.

[19] Potentially leading to incorrect cast insertion.

Final Forms

The primary addition of Hazel is the addition of a new kind of *final forms* and *values*. This is what allows evaluation to proceed around holes and errors. There are three types of final forms:

- *Values* – Constants and functions.
- *Boxed Values* – Values wrapped in *injection* casts, or *function*[20] casts.
- Indeterminate Final Forms – Terms containing holes that cannot be directly evaluated, e.g. holes or function applications where the function is indeterminate, e.g. $(\!|\,|\!)^u(1)$.

Importantly, *any* final form *can* be treated as a value and, for example, passed inside a (determinate) function, e.g. $(\lambda x.x)((\!|\,|\!)^u)$ can evaluate to $(\!|\,|\!)^u$.

maybe a good place to put an example here

Full rules are present in Fig. A.9.

Dynamics

A small-step contextual dynamics [18, ch. 5] is defined on the internal expressions to define a *call-by-value*[21] **ENSURE THIS** evaluation order.

Like the *refined criteria* [45], Hazel presents a rather different cast semantics designed around *ground types*, that is *base types*[22] and least specific[23] compound types associated via a ground matching relation mapping compound types to their corresponding ground type, e.g. $\texttt{int} \to \texttt{int} \blacktriangleright_{\text{ground}} \texttt{?} \to \texttt{?}$.

[20] Between function types

[21] Values in this sense are *final forms*.

[22] Like `int` or `bool`.

[23] In the sense that `?` is more general than any concrete type.

This formalisation more closely represents common dynamically typed language implementations which only use generic type tags like *fun*, corresponding to the ground type $? \to ?$. However, the idea of type consistency checking when *injections* meet *projections* remains the same.[24]

The cast calculus is more complex as discussed previously, due to being based around *ground types* . However, the fundamental logic is similar to the dynamic type system described previously **ref backwards**.

Evaluation proceeds by *capture avoiding variable substitution* $[d'/x]d$ (substitute d' for x in d). Additionally, substitutions are recorded in each hole's environment σ by substituting all occurences of x for d in each σ **Add figure for this**.

The instruction transitions are in Fig. A.10 and the contextual dynamics defining a small-step semantics in A.12. **SWAP DYNAMICS BACK TO DETERMINISTIC**

A contextual dynamics is defined via an Evaluation context

Hole Substitutions

Holes are indexed by *metavariables* u, and can hence also be substituted. Hole substitution is a *meta* action $[\![d/u]\!]d'$ meaning substituting each hole named u for expression d in some term d' with the holes environment. Importantly, the substitutions d can contain variables, whose values are found by checking the holes *environment*, effectively making a *delayed substitution*. See the following rule:

$$[\![d/u]\!](\!|)^u_\sigma = [[\![d/u]\!]\sigma]d$$

When substituting a matching hole u, we replace it with d and *apply substitutions from the environment* σ *of* u to d.[25] This

[24] With projections/injections now being to/fro *ground types*.

[25] After first substituting any occurrences of u in the environment σ

corresponds to *contextual substitution* in CMTT. The remaining rules can be found in A.13

This can be thought of as a *fill-and-resume* functionality, allowing incomplete program parts to be filled *during evaluation* rather than only before evaluation.

As Hazel is a *pure language*[26] and as holes act as closures, then performing hole substitution is *commutative* with respect to evaluation. That is, filling incomplete parts of a program *before* evaluation gives the same result as filling *after* evaluation then resuming evaluation. Formalised in **ref theorems**.

2.1.4 The Hazel Implementation

The Hazel implementation [22] is written primarily in ReasonML and OCaml with approx. 65,000 lines of code. It implements the Hazel core calculus along with many additional features. Relevant features and important abstractions are discussed here.

Language Features

- **Lists** –
- **Tuples** –
- **Labelled Sums** –
- **Type Aliases** –
- **Pattern Matching** –
- **Explicit Polymorphism** – System F style
- **Recursive Types** –

[26] Having no side effects.

Monadic Evaluator

This is extremely hard to explain concisely!! or at all... Ask on Slack?

The transition semantics are defined on an intricate *monadic* **is this is actually a monad...?** evaluator which is discussed in depth in the Implementation section **REF**. It is equipped with custom `let.` and `and.` binding operators[27] [**OCamlManual**], and a `otherwise` and `req_final` function. Allowing transition rules to be simply written (simplified):

```
...
| Seq(d1, d2) =>
        let. _ = otherwise(d1 => Seq(d1, d2))
        and. d1' =
          req_final(req(state, env), d1 => Seq1(d1, d2), d1);
        Step({expr: d2, state});
...
| Int(i) =>
      let. _ = otherwise(env, Int(i));
      Value;
...
 | EmptyHole =>
      let. _ = otherwise(env, EmptyHole);
      Indet;
...
```

Representing rules by a `let. _ = otherwise(env, r)` determining how to rewrap an expression if it is unevaluable. A term may be unevaluable if it requires some subterms to be *final*, but that this is not the case.

The `req_final(req(state, env), _, d)` function will pass a reference the recursive evaluation abstraction `req`, which the abstraction may choose to recursively evaluate, and bind a resulting value for use in calculating the next step.

Explain the middle EvalCtx arg to `req_final`...

[27] Which allow a convenient for writing code with binding functions.

Each transition returns either a possible step `Step({expr})`, or states that the term is indeterminate `Indet`, or a value `Value`.[28]

The results that these 'evaluate' to are abstract, they do not necessarily have to be terms, as demonstrated by the following implementations:

- **Final Form Checker** – Returns whether a term is one of each of the final form. Using the evaluator abstraction with this means there is no need to maintain a separate syntactic value checker, instead it is derived directly from the evaluation transitions. Yet, it is still syntactic since the abstraction does not actually perform evaluation steps and continue evaluation, instead it just makes the step accessible[29] to the implementation.

- **Evaluator** – Maintains a stack machine and actually performs the reduction steps.

- **Stepper** – Returns a list of possible evaluation steps in terms of *evaluation contexts* under a non-deterministic evaluation method **Explain how** `EvalCtx.t => EvalCtx.t` **in** `req_final` **allows this**. The evaluation order can then be user-controlled.

UI Architecture

Model View Update model.

[28] Or a constructor, discussed more in the Implementation section.

[29] And the final form checker will just classify such an expression immediately as non-final.

2.1.5 Non-Determinism

2.2 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 for search procedures. Primarily, the OCaml search procedure for ill-typed witnesses Seidel et al. [43] and the Hazel core language [31] were researched over the preceding summer.

Tools and Source Code

My only experience in OCaml was from the Part IA Foundations of Computer Science course. The Hazel source code had not been inspected in any detail until after starting the project.

2.3 Requirement Analysis

2.4 Software Engineering Methodology

Do the theory first.

2.5 Legality

MIT licence for Hazel.

Chapter 3

TEMPORARY: Implementation Plan

3.1 Cast Slicing

3.1.1 Theoretical Foundations and Context

Context: Why are casts elaborated

See The Gradualizer [7] 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 [19]. 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 [47, 17].

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:

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

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

$\boxed{\Gamma \vdash e \Leftarrow \tau \rightsquigarrow d : \tau' \dashv \Delta \parallel \varsigma'}$ 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 [6] 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 [34].

See constraint free error slicing [42].

See the gradualizer to understand how casts work [7].

See Blame tracking to see how common type location transfer is handled [48].

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 [6] for intuition and [48] for intuition on polarity affecting blame.

[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

Consider deeply type constructor polarities [38, 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 [30, 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 aggregat-

ing 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 [13] might be useful.

3.2 Search Procedure

Look into Zippers [23], 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 [16], Gradual dynamic type inference [28], Seidel et al. [43], 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.

Generating of values to substitute will match the required type as annotated on the hole. These should be the most gen-

[11] Or the evaluation environment will track these.

eral 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' [3][12] or just could be random as in QuickCheck [8], SmallCheck [41]. 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 [41]. For example, even if a function has a cast requiring integer arguments, the hole instantiation should be delayed, as passing a hole is 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).

[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.

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 [40].

A *semi-persistent* data structure [10] 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 [8])

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 [53]) 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.*

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.

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

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 [24, 32] 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. [43].

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

There are 3 modes of use that I can envisage:

1. Searching in a selected expression:
    - Select an expression (via cursor)
    - Start search procedure on this expression, instantiating lazily all holes in the expression.
    - Evaluate as far as possible/up until a limit to collect all cast errors. Or alternatively stop at the first.
2. Searching for witnesses of a specific, statically caught, type error.
    - 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. Searching in a selected expression, around holes:

   Same interaction as (1), but evaluates around holes using a multi-zipper structure. This approach has the following benefits:

   - Avoids evaluating regions without holes (unless necessary).
   - Finds errors in (some) dead code.
   - Is a more principled way to attempt to find instantiations of *every* hole involved. Each trace around each hole is evaluated to similar depths.

   And downsides:

   - Implementation difficulty.
   - Higher constant factor in evaluation.
   - Unusual evaluation order, making reasoning more difficult. This could be remedied by creating a system to replay the evaluation in a more standard ordering than the search procedure used.

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).

3.3.5 Actual Plan of Action

- Implement hole substitution.
- Implement lazy hole instantiation: First when inside casts. Then also upon reaching operation when no wrapped in a cast. **This is nondeterministic. Represent with lazy lists and enumerations of all instances of a type (see smallcheck).**
- Implement a evaluation viewer for this evaluation semantics, similar to the stepper.
- Explaining holes: Clicking each hole instantiation gives a reason why it was instantiated. Importantly, it will also link back to the ids of the original hole in the code AND make it clear why this term is a hole (i.e. due to being an error)[23].
- Search for witnesses of statically caught type errors. For the three cases: Inconsistent types using an analysis context; Inconsistent branches, which will produce a list of evaluation traces *(note: no cast errors here)*[24]; Inconsistent with arrow type.
- UI: Click on static type errors to obtain this.
- Implement a multi-zipper data structure. [32]
- Add a 'composition' function that given a cursor on a hole in the zipper, can compose the hole with it's context to find the smallest reduction involving it.

[23] Error holes which involve instantiated free variables may be somewhat confusing (they will be instantiated to a different type to the value within the hole)

[24] Consider how to link these errors back to code

- Implement the multi-zipper search procedure evaluation using this.

Chapter 4

Implementation

This project was conducted in two major phases:

First, I constructed a core mathematical theory for *type slicing* and *cast slicing* formalising what these ideas actually were and considered the changes to the system presented by Seidel et al. for the *type error witnesses search procedure* to work in Hazel.

Then, I implemented the theories, making it suitable for implementation and extending it to the majority of the Hazel language. Further, suitable deviations from the theory were made upon critical evaluation and are detailed throughout.

Annotate the above with the relevant section links!

4.1 Type Slicing Theory

4.1.1 Program Slices

4.1.2 Synthesis Type Slices

4.1.3 Analysis Type Slices

Detail initial plan, then describe how this is nice and easy maths but not very useful

4.2 Cast Slicing Theory

4.3 Proofs

4.4 Type Slicing Implementation

4.4.1 Type Slice Data-Type

Detail initial implementation (just tagging existing types).

Code Slices

id based. Has ctx used but not actually required as I decide to directly store typslices in context (explain how this differs from the theory)

Integration with Existing Type Data-Type

Synthesis & Analysis Slices

Incremental/Global. Detail the choice to not annotate many analysis slices

Mapping Functions

Type Slice Joins

4.4.2 Static Type Checking

4.4.3 Elaboration

4.4.4 User Interface

4.5 Cast Slicing Implementation

4.5.1 Cast Transitions

4.5.2 User Interface

Mainly talk about the Model-view architecture and passing the cursor into the evaluator view to allow

4.6 Indeterminate Evaluation

4.6.1 Futures Data-Type

4.6.2 Hole Instantiation

Small Hole hypothesis, quick check

Choosing which Hole to Instantiate

Synthesising Terms for Types

Substituting Holes

Detail that this was an unexpected extra task, and is therefore not exactly the same as hole substitution as detailed in Prepa-

ration (i.e. no metavars or contexts annotated on holes, but it is enough for the search procedure to work)

4.6.3 Cast Laziness

Ref the original cast slicing paper.

4.6.4 User Interface

4.7 Evaluation Stepper

4.7.1 Customisable Hole Instantiation

4.7.2 User Interface

4.8 Search Procedure

4.8.1 Detecting Relevant Cast Errors

i.e. failed cast at head of term

4.8.2 Filtering Indeterminate Evaluation

4.8.3 Iterative Deepening

Required after evaluating that infinite loops break the thing

4.8.4 User Interface

Chapter 5

Evaluation

Should I put proposed implementation plans and improvements here or in implementation??

Evaluation Here.

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

5.1 Goals

i.e. Project Proposal. But make it with more clarity, i.e. 'Most Type Errors Admit Witnesses' Completeness etc. most of Hazel...

5.2 Program Corpus Collection

5.2.1 Methodology

5.2.2 Alternatives

OCaml -¿ Hazel transpiler

5.3 Effectiveness Analysis

5.3.1 Search Procedure

Witness Coverage

Describe reasons for failure in next section

Code Coverage

Trace Size

Slice Size

Does this correlate with trace size!

5.3.2 Type Slicing

Correctness

?

Code Slice Size

Compare with theory adhering slices and 'simplified' slices.

5.4 Critical Analysis

5.4.1 Analysis Type Slices

Which constructs are ignored? Why, do they have the biggest impact on size?

5.4.2 Categorising Programs Lacking Type Error Witnesses

Non-Termination

The original procedure would get stuck on programs that loop forever. (To) Fix with iterative deepening

Repeated Instantiations

A hole being instantiated to a hole. Does this ever happen??

Dead Code

Search proc cannot reach, and failed cast detection would never find one there even if it existed (i.e. statically found).

Dynamically Safe Code

Code that is safe to run in all situations, but still exhibits a static type error.

Needle in a Haystack

Very specific input required from multiple hole instantiations. Combinatorial explosion makes this very hard to find (solve with coverage directed search, but this requires SMT solvers at least).

5.4.3 Non-Local Errors

Useful when type correct code written but used in the wrong way, i.e. write the wrong map function with @ still has a valid type. Especially prevalent with global inference.

5.4.4 Bidirectional Type Error Localisation

It is generally good. Find some cases where bidirectional type error localisation is wrong.

5.4.5 Improving Hole Instantiation

To improve code coverage. i.e. Strings and Floats are annoying, SMT solvers could be used...

5.4.6 Combinatorial Explosion

5.5 Performance Analysis

5.5.1 Search Procedure

Time

Space

5.5.2 Slices

Space

Compare with using using ‘Typ (turned off). Compare with old implementation (if possible)

Chapter 6

Conclusions

6.1 Conclusion

Conclusions Here.

6.2 Further Directions

Further directions here, referencing unsatisfactory results from Evaluation. Also various extensions.

6.2.1 Extension to Full Hazel Language

6.2.2 Slicing

Proofs

User Studies

Effectiveness gauge

6.2.3 Search Procedure

Jump Trace Compression

Symbolic Execution & SMT Solvers

Ad-Hoc Polymophism

Not in Hazel, but the search procedure can't deal well with it

Formal Semantics & Proofs

Was an uncompleted extension goal

6.2.4 Let Polymorphism & Global Inference

Constraint Slicing

Gradual Type Inference

Miyazaki

Localisation

Bidirectional typing localisation is good, but global inference is bad. The search procedure could improve localisation and also give more meaning to localisations.

Bibliography

[1] M. Abadi et al. “Dynamic typing in a statically-typed language”. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’89. Austin, Texas, USA: Association for Computing Machinery, 1989, pp. 213–227. ISBN: 0897912942. DOI: 10.1145/75277.75296. URL: https://doi.org/10.1145/75277.75296.

[2] Abdulaziz Alaboudi and Thomas D. LaToza. *An Exploratory Study of Debugging Episodes*. 2021. arXiv: 2105.02162 [cs.SE]. URL: https://arxiv.org/abs/2105.02162.

[3] Alexandr Andoni et al. “Evaluating the ”Small Scope Hypothesis””. In: (Oct. 2002).

[4] Moritz Beller et al. “On the dichotomy of debugging behavior among programmers”. In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE ’18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 572–583. ISBN: 9781450356381. DOI: 10.1145/3180155.3180175. URL: https://doi.org/10.1145/3180155.3180175.

[5] Maurice Bruynooghe. “Adding redundancy to obtain more reliable and more readable prolog programs”. In: *CW Reports* (1982), pp. 5–5.

[6] Matteo Cimini and Jeremy G. Siek. “Automatically generating the dynamic semantics of gradually typed languages”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. Vol. 7. POPL ’17. ACM, Jan. 2017, pp. 789–803. DOI: 10.1145/3009837.3009863. URL: http://dx.doi.org/10.1145/3009837.3009863.

[7] Matteo Cimini and Jeremy 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*. POPL ’16. ACM, Jan. 2016, pp. 443–455. DOI: 10.1145/2837614.2837632. URL: http://dx.doi.org/10.1145/2837614.2837632.

[8] Koen Claessen and John Hughes. “QuickCheck: a lightweight tool for random testing of Haskell programs”. In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. ICFP ’00. New York, NY, USA: Association for Computing Machinery, 2000, pp. 268–279. ISBN: 1581132026. DOI: 10.1145/351240.351266. URL: https://doi.org/10.1145/351240.351266.

[9] Zack Coker et al. “A Qualitative Study on Framework Debugging”. In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2019, pp. 568–579. DOI: 10.1109/ICSME.2019.00091.

[10] Sylvain Conchon and Jean-Christophe 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. URL: http://dx.doi.org/10.1007/978-3-540-78739-6_25.

[11] Bas Cornelissen, Andy Zaidman, and Arie van Deursen. "A Controlled Experiment for Program Comprehension through Trace Visualization". In: *IEEE Transactions on Software Engineering* 37.3 (2011), pp. 341–355. DOI: 10.1109/TSE.2010.47.

[12] Jana Dunfield and Neelakantan R. Krishnaswami. "Complete and easy bidirectional typechecking for higher-rank polymorphism". In: *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*. ICFP'13. ACM, Sept. 2013. DOI: 10.1145/2500365.2500582. URL: http://dx.doi.org/10.1145/2500365.2500582.

[13] Jean-Christophe Filliâtre and Sylvain Conchon. "Type-safe modular hash-consing". In: *Proceedings of the 2006 workshop on ML*. ICFP06. ACM, Sept. 2006. DOI: 10.1145/1159876.1159880. URL: http://dx.doi.org/10.1145/1159876.1159880.

[14] Robert Bruce Findler and Matthias Felleisen. "Contracts for higher-order functions". In: *SIGPLAN Not.* 37.9 (Sept. 2002), pp. 48–59. ISSN: 0362-1340. DOI: 10.1145/583852.581484. URL: https://doi.org/10.1145/583852.581484.

[15] Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. "Semantic casts: Contracts and structural subtyping in a nominal world". In: *European Conference on Object-Oriented Programming*. Springer. 2004, pp. 365–389.

[16] Ronald Garcia and Matteo Cimini. "Principal Type Schemes for Gradual Programs". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '15. Mumbai, India: Association for Computing Machinery, 2015, pp. 303–315. ISBN: 9781450333009. DOI: 10.1145/2676726.2676992. URL: https://doi.org/10.1145/2676726.2676992.

[17] Christian Haack and J.B. Wells. “Type error slicing in implicitly typed higher-order languages”. In: *Science of Computer Programming* 50.1–3 (Mar. 2004), pp. 189–224. ISSN: 0167-6423. DOI: 10.1016/j.scico.2004.01.004. URL: http://dx.doi.org/10.1016/j.scico.2004.01.004.

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

[19] Robert Harper and John C. Mitchell. “On the type structure of standard ML”. In: *ACM Transactions on Programming Languages and Systems* 15.2 (Apr. 1993), pp. 211–252. ISSN: 1558-4593. DOI: 10.1145/169701.169696. URL: http://dx.doi.org/10.1145/169701.169696.

[20] Robert Harper and Christopher 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. URL: http://dx.doi.org/10.7551/mitpress/5641.003.0019.

[21] *Hazel Project Website*. URL: https://hazel.org/ (visited on 02/28/2025).

[22] *Hazel Source Code*. URL: https://github.com/hazelgrove/hazel (visited on 02/28/2025).

[23] GÉRARD HUET. “The Zipper”. In: *Journal of Functional Programming* 7.5 (Sept. 1997), pp. 549–554. ISSN: 1469-7653. DOI: 10.1017/s0956796897002864. URL: http://dx.doi.org/10.1017/S0956796897002864.

[24] Oleg Kiselyov. *Generic Zippers*. URL: https://okmij.org/ftp/continuations/zipper.html.

[25] Bogdan Korel and Janusz Laski. "Dynamic program slicing". In: *Information Processing Letters* 29.3 (Oct. 1988), pp. 155–163. ISSN: 0020-0190. DOI: 10.1016/0020-0190(88)90054-3. URL: http://dx.doi.org/10.1016/0020-0190(88)90054-3.

[26] Lucas Layman et al. "Debugging Revisited: Toward Understanding the Debugging Needs of Contemporary Software Developers". In: *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. 2013, pp. 383–392. DOI: 10.1109/ESEM.2013.43.

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

[28] Yusuke Miyazaki, Taro Sekiyama, and Atsushi Igarashi. "Dynamic type inference for gradual Hindley–Milner typing". In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: 10.1145/3290331. URL: https://doi.org/10.1145/3290331.

[29] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. "Contextual modal type theory". In: *ACM Transactions on Computational Logic* 9.3 (June 2008), pp. 1–49. ISSN: 1557-945X. DOI: 10.1145/1352582.1352591. URL: http://dx.doi.org/10.1145/1352582.1352591.

[30] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, Apr. 1998. ISBN: 9780511530104. DOI: 10.1017/cbo9780511530104. URL: http://dx.doi.org/10.1017/CBO9780511530104.

[31] Cyrus Omar et al. "Live functional programming with typed holes". In: *Proceedings of the ACM on Programming Languages* 3.POPL (Jan. 2019), pp. 1–32. ISSN: 2475-1421. DOI: 10.1145/3290327. URL: http://dx.doi.org/10.1145/3290327.

[32] Pavel Panchekha. *Multi Zippers*. URL: https://pavpanchekha.com/blog/zippers/multi-zippers.html#org1556357.

[33] Linda Dailey Paulson. "Developers shift to dynamic programming languages". In: *Computer* 40.2 (2007), pp. 12–15. DOI: 10.1109/MC.2007.53.

[34] Roly Perera et al. "Functional programs that explain their work". In: *ACM SIGPLAN Notices* 47.9 (Sept. 2012), pp. 365–376. ISSN: 1558-1160. DOI: 10.1145/2398856.2364579. URL: http://dx.doi.org/10.1145/2398856.2364579.

[35] FRANK PFENNING and ROWAN DAVIES. "A judgmental reconstruction of modal logic". In: *Mathematical Structures in Computer Science* 11.04 (July 2001). ISSN: 1469-8072. DOI: 10.1017/s0960129501003322. URL: http://dx.doi.org/10.1017/S0960129501003322.

[36] Benjamin C Pierce. *Advanced topics in types and programming languages*. MIT press, 2024.

[37] Benjamin C Pierce. *Basic category theory for computer scientists*. MIT press, 1991.

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

[39] Benjamin C. Pierce and David N. Turner. "Local type inference". In: *ACM Transactions on Programming Languages and Systems* 22.1 (Jan. 2000), pp. 1–44. ISSN: 1558-4593. DOI: 10.1145/345099.345100. URL: http://dx.doi.org/10.1145/345099.345100.

[40] Maciej Pirog and Jeremy Gibbons. "A Functional Derivation of the Warren Abstract Machine". In: (2011). Submitted for publication. URL: http://www.comlab.ox.ac.uk/jeremy.gibbons/publications/wam.pdf.

[41] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. "Smallcheck and lazy smallcheck: automatic exhaustive testing for small values". In: *Proceedings of the First ACM SIGPLAN Symposium on Haskell*. Haskell '08. Victoria, BC, Canada: Association for Computing Machinery, 2008, pp. 37–48. ISBN: 9781605580647. DOI: 10.1145/1411286.1411292. URL: https://doi.org/10.1145/1411286.1411292.

[42] Thomas 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. URL: http://dx.doi.org/10.1007/978-3-642-32037-8_1.

[43] Eric L. Seidel, Ranjit Jhala, and Westley 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*. ICFP'16. ACM, Sept. 2016, pp. 228–242. DOI: 10.1145/2951913.2951915. URL: http://dx.doi.org/10.1145/2951913.2951915.

[44] Jeremy G. Siek and Walid Taha. "Gradual Typing for Functional Languages". In: *Scheme and Functional Programming Workshop*. 2006, pp. 81–92.

[45] Jeremy G. Siek et al. "Refined Criteria for Gradual Typing". In: *Summit on Advances in Programming Languages*. 2015. URL: https://api.semanticscholar.org/CorpusID:15383644.

[46] TIOBE Software. *TIOBE Programming Community Index*. [Online; accessed 27-February-2025]. 2025. URL: https://www.tiobe.com/tiobe-index/.

[47] F. Tip and T. B. Dinesh. "A slicing-based approach for locating type errors". In: *ACM Transactions on Software Engineering and Methodology* 10.1 (Jan. 2001), pp. 5–55. ISSN: 1557-7392. DOI: 10.1145/366378.366379. URL: http://dx.doi.org/10.1145/366378.366379.

[48] Philip Wadler and Robert Bruce 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. URL: http://dx.doi.org/10.1007/978-3-642-00590-9_1.

[49] David HD Warren. "Applied logic: its use and implementation as a programming tool". In: (1978).

[50] Mark David Weiser. "Program Slices: Formal, Psychological, And Practical Investigations Of An Automatic Program Abstraction Method." In: (1979). DOI: 10.7302/11363. URL: http://deepblue.lib.umich.edu/handle/2027.42/180974.

[51] J.B. Wells. "Typability and type checking in System F are equivalent and undecidable". In: *Annals of Pure and Applied Logic* 98.1 (1999), pp. 111–156. ISSN: 0168-0072. DOI: https://doi.org/10.1016/S0168-0072(98)00047-5. URL: https://www.sciencedirect.com/science/article/pii/S0168007298000475.

[52] Baijun Wu and Sheng Chen. "How type errors were fixed and what students did?" In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017). DOI: 10.1145/3133929. URL: https://doi.org/10.1145/3133929.

[53] Ningning Xie and Daan Leijen. "Effect handlers in Haskell, evidently". In: *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell*. Haskell 2020. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 95–108. ISBN: 9781450380508. DOI: 10.

1145/3406088.3409022. URL: https://doi.org/10.1145/3406088.3409022.

Appendix A

Hazel Formal Semantics

This is the complete formal semantics for the Hazel *core calculus*. It is *gradually typed* so consists of both an *external language* and *internal language*.

ADD THE USEFUL THEOREMS AND REF THROUGHOUT TEXT. Add brief notes pointing out unusual features.

A.1 Syntax

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

Figure A.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.

A.2 Static Type System

A.2.1 External Language

$\boxed{\Gamma \vdash e \Rightarrow \tau}$ 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 A.2: Bidirectional typing judgements for *external expressions*

$\boxed{\tau_1 \sim \tau_2}$ 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 A.3: Type consistency

$\boxed{\tau \blacktriangleright_{\to} \tau_1 \to \tau_2}$ 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 A.4: Type Matching

A.2.2 Elaboration

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

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

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

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

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

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

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

$\boxed{\Gamma \vdash e \Leftarrow \tau \rightsquigarrow d : \tau' \dashv \Delta}$ 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 A.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 A.7: Identity substitution and substitution typing

A.2.3 Internal Language

$\boxed{\Delta; \Gamma \vdash d : \tau}$ 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 A.6: Type assignment judgement for *internal expressions*

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

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

Figure A.8: Ground types

A.3 Dynamics

A.3.1 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

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

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

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

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

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

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

Figure A.9: Final forms

A.3.2 Instructions

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

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

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

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

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

Figure A.10: Instruction transitions

A.3.3 Contextual Dynamics

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

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

Figure A.11: Ground type matching

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 A.12: Contextual dynamics of the internal language

A.3.4 Hole Substitution

$\boxed{[\![d/u]\!]d' = d''}$ 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 A.13: Hole substitution

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 syntax, 14

External language type system, 15

Hazel, 14

Project Proposal

Description

This project will add some features to the Hazel language [21]. 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. [43]. 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 [48], error and dynamic program slicing [47, 25], 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 [21] 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 [31].
- **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 [22].
- 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.