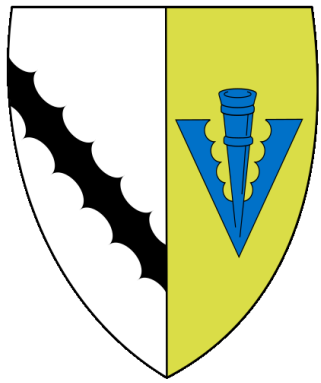
Max Carroll

Type Error Debugging in Hazel

Computer Science Tripos, Part II



Sidney Sussex College College
University of Cambridge
March 5, 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:	**5416 (errors:9)** [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.

ContentsblueUwasy#CONTEN

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 [**DebugTimeSelfReport**], with programmers spending a *highly skewed* proportion of their time identifying and resolving a small proportion of *difficult* bugs [**DebugSkew**].

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 [**StudentTypeErrorFixes**]. 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* [**TraceVisualisation**]

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 [**Hazel**], 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 [**HazelLivePaper**]. Further, I implement a *type error witness search procedure* based upon a similar idea implemented for a subset of OCaml by Seidel et al. [**SearchProc**]:

- **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 [**DebugNeeds**], *how* developers fix bugs [**HowFixBugs**], 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 [**ProgSlice**], though my definition of program slices matches more with functional program slices [**FunctionalProgExplain**], and the properties I explore are more similar to *dynamic program slicing* [**DynProgSlice**] but with type characteristics rather than evaluation characteristics, in a sense similar *type error slicing* [**ErrSlice**].

The *type witness search procedure* is based upon Seidel et al. [**SearchProc**], 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

Chapter 2 introduces the *type theories* (section 2.1.1) underpinning the *Hazel core calculus* (section 2.1.2), followed by the *Hazel implementation* (section 2.1.3).

Section 3.1 and section 3.2 formalise the ideas of *type slices* and *cast slices* in the Hazel core calculus, with properties proven[1] in section 3.3.

An implementation (section 3.4-3.5) of these has been created covering *most*[2] of the Hazel language, including a user interface.

A *type witness search procedure* was successfully implemented. This involved creating a *hole instantiation* and a simplified *hole substitution* method (section 3.7.2); there is currently no full hole substitution feature in Hazel despite it's presence in the core calculus (section 2.1.2). Additionally, a customisable instantiation method was implemented (section 3.8) controllable via a UI.

The slicing features and search procedure met all goals, **list eval goals briefly**, showing *effectiveness* (section 4.4) and being reasonably *performant* (section 4.5) over a corpus of *well-typed* and *ill-typed* Hazel programs respectively (section 4.3). Further, considered deviations from the slicing theories and strengths and weaknesses of the search procedure were evaluated in detail (section 4.6).

Finally, further directions and improvements have been presented along with discussion on the applicability of these features, slicing in particular, to real world debugging situations in chapter 5.

[1] **TODO**

[2] Except for type substitution.

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 [**PracticalFoundations**]. 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] [**TAPL**] 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, $\lambda x.e$ 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.

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 [**DynamicLangShift TIOBE**]. Of particular relevance to this project, execution traces are known to help provide insight to errors [**TraceVisualisation**], 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] [**DynamicTyping**]. 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, *semantic casts* [**SemanticCasts**] in *contract* systems

[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.

[**Contracts**]:

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

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* [**BasicCatTheory**] 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* [**GradualRefined**, **GradualFunctional**] 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* [**TAPL**] 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 $\blacktriangleright_{\rightarrow}$ 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 $\tau_1' \rightarrow \tau_2'$ 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 [**StandardMLTypeTheory**] by elaboration to an explicitly typed internal language XML [**CoreXML**]. 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:

$$\frac{\begin{array}{cc}\Gamma \vdash e_1 \rightsquigarrow d_1 : \tau_1 & \Gamma \vdash e_2 \rightsquigarrow d_2 : \tau_2' \\ \tau_1 \blacktriangleright_{\rightarrow} \tau_2 \rightarrow \tau & \tau_2 \sim \tau_2'\end{array}}{\Gamma \vdash e_1(e_2) : \tau \rightsquigarrow (d_1\langle \tau_1 \Rightarrow \tau_2 \rightarrow \tau \rangle)(d_2\langle \tau_2' \Rightarrow \tau_2 \rangle) : \tau}$$

If, e_1 elaborates to d_1 with type $\tau_1 \sim \tau_2 \rightarrow \tau$ 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 $\tau_2 \rightarrow \tau$ 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 [**Gradualizer**].

The runtime semantics of the internal expression is that of the *dynamic type system* discussed above (2.1.1). A cast is determined to succeed iff the types are *consistent*.

[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 $\tau_1 = \tau_2 \rightarrow \tau$ and the cast is redundant.

The *refined criteria* for gradual typing [**GradualRefined**] also provides an additional property for such systems to satisfy, the *gradual guarantee*, formalising the intuition that adding and removing annotations should *not* change the *behaviour* of the program except for catching errors either dynamically or statically.

Bidirectional Type Systems

A *bidirectional type system* [**BidirectionalTypes**] takes on a more algorithmic definition of typing judgements, being more intuitive to implement. They also allow some amount of local type inference [**LocalInference**], allowing programmers to omit *type annotations*, instead type information. Global type inference systems [**TAPL**] can be *difficult to implement*, often via constraint solving [**ATTAPL**], and difficult or impossible to *balance* with complex language features, for example global inference in System F (2.1.1) is undecidable [**SystemFUndecidable**].

This is done in a similar way to annotating logic programming [**LogicProg**], by specifying the *mode* 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* [**ModeCorrectness**]. 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:

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

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 it is 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:

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

[12] Highlighted in red as an error.

Contextual Modal Type Theory

Not *hugely* relevant really...

System F

Very brief explanation with less/no maths

Recursive Types

Very brief explanation with less/no maths

2.1.2 The Hazel Calculus

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

It does this via adding *expression holes*, which can both be typed and have evaluation proceed around them seamlessly. This allows the evaluation around errors by placing them in holes.

The core calculus [**HazelLivePaper**] 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, but only particularly notable consequences are mentioned here. The intuition

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

should be clear from the previous gradual and bidirectional typing sections.[14]

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 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.

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

[15] This is required, as holes may later be substituted, with variables then receiving their values from the closure environment.

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 typing is naturally extended to allow subsuming any terms of *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 each hole *metavariables* u to it's *checked type* τ [16] and the type context Γ under which the hole was typed. Each metavariable context notated as $u :: \tau[\Gamma]$, notation borrowed from contextual modal type theory (CMTT) [**CMTT**].[17]

[16] Originally required when typing the external language expression. See Elaboration section.

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

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.

Full rules in Fig. A.6. Formally speaking these define categorical judgements [**ModalJudgements**]. 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.

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

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 $\sigma =$ $mathrmid(\Gamma)$, 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.

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 (in a *call-by-value* context). For example, they can be passed inside a (determinate) function: $(\lambda x.x)((\!|\,|\!)^u)$ can evaluate to $(\!|\,|\!)^u$.

Full rules are present in Fig. A.9.

[19] Potentially leading to incorrect cast insertion.

[20] Between function types

Dynamics

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

Like the *refined criteria* [**GradualRefined**], 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. $\texttt{int} \to \texttt{int} \blacktriangleright_{\text{ground}} \texttt{?} \to \texttt{?}$. This formalisation more closely represents common dynamically typed language implementations which only use generic type tags like *fun*, corresponding to the ground type $\texttt{?} \to \texttt{?}$. 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 in section 2.1.1.

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... *(TODO, explain evaluation contexts as will be relevant to the Stepper EV_MODE explanation)*

[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.

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

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 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**.

2.1.3 The Hazel Implementation

The Hazel implementation [**HazelCode**] 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.

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

[26] Having no side effects.

Language Features

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

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;
```

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

```
...
 | 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 `string`, 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`...

Each transition returns either a possible step `++ "str"`, 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.

[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.

- **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.

Each evaluation method module is transformed into an evaluator module by being passed into an OCaml *functor*. The resulting module produces a `transition` method that takes terms to evaluation results in an environment[30].

UI Architecture

Model View Update model.

2.1.4 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. [**SearchProc**] and the Hazel core lan-

[30] Mapping variable bindings.

guage [**HazelLivePaper**] 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

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!

3.1 Type Slicing Theory

I develop *type slicing* as a mechanism to aid programmers in understanding *why* a term has a given type via static means. Three slicing mechanism have been devised with differing characteristics, all of which associate terms with their typing derivation to produce a *program slice*.

The first two criteria attempt to give insight on the structure of the typing derivations, and hence how types are decided. While the third criterion gives a complete picture of the regions of code which, if changed, could cause a change in type of the whole expression.

Make some brief arguments into why the first two criteria are still useful.

3.1.1 Program Slices

A *program slice* ρ, is a pair $[\varsigma \mid \gamma]$, consisting of an *expression slice* and *typing context slice* respectively which are calculated based on some typing *criterion*[1] based on the typability of the slice ς under context γ.

Intuitively, an expression slice is a Hazel external expression highlighting the sub-terms of relevance to the *typing criterion*. For example if my criterion is to *omit terms which are typed as int*, then the following expressions highlights as:

$$(\lambda x : \texttt{Int}.\ \lambda y : \texttt{Bool}.\ x\ )(\ 1\ )$$

Formally, I represent this by specifying which sub-terms are omitted in the highlighted expression. So, Replace each omitted sub-term with a *gap*, notated $\square$. This is the same definition of a slice as presented in [**FunctionalProgExplain**].[2] i.e. representing the above highlighting we get slice:

$$(\lambda x : \texttt{Int}.\ \lambda y : \texttt{Bool}.\ \square)(\square)$$

Additionally, it is useful to omit variable names. For this I introduce *patterns* p for variable bindings:

$$p ::= \square \mid x$$

[1] One of the three slicing mechanisms.

[2] With their 'holes' equating with my 'gaps'. Different terminology used to distinguish with Hazel's holes

This gives the following extended syntax of expression slices, ς, extending fig. 2.1:

$$\varsigma ::= \square \mid c \mid x \mid \lambda p : \upsilon.\ \varsigma \mid \lambda x.\ \varsigma \mid \varsigma(\varsigma) \mid (\!|\,|\!)^u \mid (\!|\varsigma|\!)^u \mid \varsigma : \upsilon$$

Where υ are types, similarly with potential omitted sub-term gaps:

$$\upsilon ::= \square \mid ?b \mid \upsilon \to \upsilon$$

These slices are then allowed to be *typed* by representing gaps $\square$ by holes of fresh metavariables $(\!|\,|\!)^u$ in *expressions*, fresh variables in *patterns*, and the dynamic type in *types*, see **(fig APPENDIX)**. From here-on consider $\square$ as interchangeable with a hole $(\!|\,|\!)^u$ of fresh metavariable u or the dynamic type.

We then have a *precision* relation on expression slices, $\varsigma_1 \sqsubseteq \varsigma_2$ meaning ς_1 is less or equally precise than ς_2, that is ς_1 matches ς_2 structurally except that some subterms may be gaps, see **ref appendix**. For example, see this precision chain:

$$\square \sqsubseteq \square + \square \sqsubseteq 1 + \square \sqsubseteq 1 + 2$$

We have that $\sqsubseteq$ is a partial order (**cite**), that is, satisfies relexivity, antisymmetry, and transitivity. Respectively:

$$\frac{}{\varsigma \sqsubseteq \varsigma} \qquad \frac{\varsigma_1 \sqsubseteq \varsigma_2 \qquad \varsigma_2 \sqsubseteq \varsigma_1}{\varsigma_1 = \varsigma_2} \qquad \frac{\varsigma_1 \sqsubseteq \varsigma_2 \qquad \varsigma_2 \sqsubseteq \varsigma_3}{\varsigma_1 \sqsubseteq \varsigma_3}$$

We also have a *bottom* (least) element, $\square \sqsubseteq \varsigma$ (for all ς). This relation is trivially extended to include complete expressions e which satisfy that: if $e \sqsubseteq \varsigma$ then $e = \varsigma$, i.e. complete terms are always *unique* upper bounds of precision chains.

Context slices are simply a typing context Γ, which is used to represent the notion of *relevant typing assumption*. Typing contexts are just functions mapping variables to types notated $x : \tau$ (see section 2.1.1). Functions are sets, so they also have a partial order of subset inclusion, $\subseteq$. Again, we have a bottom element, $\emptyset$.

The precision relation and subset inclusion can be extended pointwise to give a partial order, $\sqsubseteq$, on program slices:

$$[\varsigma_1 \mid \gamma_1] \sqsubseteq [\varsigma_2 \mid \gamma_2] \iff \varsigma_1 \sqsubseteq \varsigma_2 \text{ and } \gamma_1 \subseteq \gamma_2$$

Program slices will often be grouped and indexed upon expressions and typing contexts, P_e^Γ which contains all slices $\rho \sqsubseteq (e, \Gamma)$. So, the set P_e^Γ forms a lattice (**cite**) with unique least upper bound $[e, \Gamma]$ and greatest lower bound $[\square, \emptyset]$.

3.1.2 Criterion 1: Synthesis Slices

For *synthesis type slices* we consider an expression synthesising a type τ under some context Γ:

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

And consider the slices in P_e^Γ and attempt to find the minimum slice $\rho = [\varsigma \mid \gamma]$ constraining that ρ also synthesises the same type τ under the restricted context γ:

$$\gamma \vdash \varsigma \Rightarrow \tau$$

Where minimality requires that no other (strictly) less precise slice satisfies the criterion. That is: for any slice $\rho' = [\varsigma' \mid \gamma']$, if $\gamma' \vdash \varsigma' \Rightarrow \tau$ and $\rho' \sqsubseteq \rho$, then $\rho' = \rho$.

GIVE CONCRETE EXAMPLE HERE, use highlighting

I conjecture that, under the Hazel type system, there exists a unique minimum slice for each $\Gamma \vdash e \Rightarrow \tau$:[3]

Conjecture 1 (Uniqueness) *If ρ and ρ' are minimum synthesis slices for $\Gamma \vdash e \Rightarrow \tau$, then $\rho = \rho'$.*

[3] Would follow from uniqueness of typing derivations in Hazel.

These slices can be found by omitting portions of the program which are *type checked*. If, $\Gamma \vdash e \Leftarrow \tau$, then by use of the subsumption rule we also have that $\Gamma \vdash \square \Leftarrow \tau$:

$$\text{Subsumption}\frac{\Gamma \vdash \square \Rightarrow ? \qquad \tau \sim ?}{\Gamma \vdash e \Leftarrow \tau}$$

As the dynamic type is consistent with any type: $? \sim \tau$.

Then, to find the *minimum synthesis slice*, we can mimic the Hazel type synthesis rules (see fig. A.2), replacing uses of type analysis with gaps. Creating a judgement $\Gamma \vdash e \Rightarrow \tau \dashv \rho$ meaning: *e that synthesises type τ under context Γ produces minimum synthesis slice ρ.*

To demonstrate, the expression slice of a variable x can only be either x, requiring the use of $x : \tau$ from the context:

$$\text{SVar}\frac{x : \tau \in \Gamma \qquad \tau \neq ?}{\Gamma \vdash x \Rightarrow \tau \dashv [x \mid x : \tau]}$$

But if $x : ?$, then the (empty) slice $[\square, \emptyset]$ also synthesises $?$, so instead use this.

For functions, we can recursively find the slice of the function body (which synthesises it's type in the original rules, hence having a minimum synthesis slice) and place inside a function. If the assumption $x : \tau_1$ was *required* in synthesising that type, then this name must be present in the expression slice and the context slice no longer requires this assumption to type check the sliced function:

$$\text{SFun}\frac{\Gamma, x : \tau_1 \vdash e \Rightarrow \tau_2 \dashv [\varsigma \mid \gamma, x : \tau_1]}{\Gamma \vdash \lambda x : \tau_1.\ e \Rightarrow \tau_1 \to \tau_2 \dashv [\lambda x : \tau_1.\ \varsigma \mid \gamma]}$$

Otherwise, if γ does not use variable x then this binding may be omitted:

$$\text{SFunConst}\frac{\Gamma, x : \tau_1 \vdash e \Rightarrow \tau_2 \dashv [\varsigma \mid \gamma] \qquad x \notin \text{dom}(\gamma)}{\Gamma \vdash \lambda x : \tau_1.\ e \Rightarrow \tau_1 \to \tau_2 \dashv [\lambda \square : \tau_1.\ \varsigma \mid \gamma]}$$

For function applications we can simply omit the argument, while the slice for the function can be obtained as it synthesises it's type.

$$\text{SApp}\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \dashv [\varsigma_1 \mid \gamma_1]}{\Gamma \vdash e_1(e_2) \Rightarrow \tau[\varsigma_1(\square) \mid \gamma_1]}$$

The remaining rules are in fig. B.1.

It is *expected* **(Proof TODO)** that these rules do indeed always produce a slice for any expression which synthesises a type, and that this slice is minimal:

Conjecture 2 (Correctness) *If* $\Gamma \vdash e \Rightarrow \tau$ *then:*

- $\Gamma \vdash e \Rightarrow \tau \dashv \rho$ *where* $\rho = [\varsigma \mid \gamma]$ *with* $\gamma \vdash \varsigma \Rightarrow \tau$.
- *For any* $\rho' = [\varsigma' \mid \gamma'] \sqsubseteq [e \mid \Gamma]$ *such that* $\gamma' \vdash \varsigma' \Rightarrow \tau$ *then* $\rho \sqsubseteq \rho'$.

Maybe push context slices into it's own section here?

3.1.3 Criterion 2: Analysis Slices

A similar idea can be devised for analysis slices. Essentially, we do the opposite of *criterion 1* and omit sub-terms where *synthesis* was used. The intention is to show how a analysing type is deconstructed based on the structure of the term being analysed.

The useful notion to represent this is the slice of an expression's *syntactic context*, it is the terms immediately around it which determine how the checked type is deconstructed:

For example, when checking this term against `Bool` $\to$ `Int`:

$$(\lambda x.(\!|\,|\!)^u)$$

The slice context of the inner hole term $(\!|\,|\!)^u$, which is required to be consistent with `Int`, would be the following:

$$(\lambda\, x\, .\, (\!|\,|\!)^u\,)$$

Intuitively, this means that the *contextual* reason for $(\!|\,|\!)^u$ to be required to be an `Int` is that it was within a function (and the context was initially checked against `Bool` $\to$ `Int` and deconstructed to retrieve the return type `Int`.

Formally, an *expression slice context* $\mathcal{C}$ is a term with at most *one* sub-term marked as $\bigcirc$:[4]

$$\mathcal{C} ::= \bigcirc \mid \square \mid c \mid x \mid \lambda x : \tau.\, \mathcal{C} \mid \lambda x.\, \mathcal{C} \mid \mathcal{C}(\varsigma) \mid \varsigma(\mathcal{C}) \mid (\!|\,|\!)^u \mid (\!|\mathcal{C}|\!)^u \mid \mathcal{C} : \tau$$

CONTEXT slices don't need gaps in this case, maybe remove?

Where $\mathcal{C}\{\varsigma\}$ substitutes expression slice ς for the mark $\bigcirc$ in $\mathcal{C}$ if it exists, the result of this is necessarily an expression slice. Additionally, contexts are composable: substituting a context into a context, $\mathcal{C}_1\{\mathcal{C}_2\}$ produces another valid context, notate this by $\mathcal{C}_1 \circ \mathcal{C}_2$.

Then, for a term e which type checks against type τ under context Γ, the *minimum analysis slice* of sub-terms e' of e checking against typed τ' under typing context Γ' inside slice context $\mathcal{C}$, that is $e = \mathcal{C}\{e'\}$ and $\Gamma' \vdash e' \Leftarrow \tau'$, is a *minimum program slice* $[\varsigma, \gamma]$ such that ς checks against τ' under γ, that $\gamma \vdash \varsigma \Leftarrow \tau'$ and ς within context $\mathcal{C}$ also checks against τ i.e. $\Gamma \vdash \mathcal{C}\{\varsigma\} \Leftarrow \tau$.[5] Written, given that $\Gamma \vdash \mathcal{C}\{e'\} \Leftarrow \tau$:

$$\Gamma'; \mathcal{C} \vdash e' \Leftarrow \tau' \dashv [\varsigma \mid \gamma]$$

This judgement can be defined syntactically, similarly to *synthesis slices*. While the Hazel core calculus only has one analysis rule (for unannotated lambdas), there are several other valid analysis rules[6] that are omitted to preserve determinacy of typing. Using these is useful to demonstrate this criterion.

[4] The two separate syntax definition for application allow a *mark* to be in either the left or right expression, but *not both*.

[5] Note that the context $\mathcal{C}$ of e' has no part omitted when type checking within the context.

[6] Flipping synthesis and analysis where *mode correctness* is preserved.

For example, for the atomic terms: constants, variables, holes etc. the empty context trivially satisfies the criterion:

$$\text{AConst}\frac{\Gamma \vdash c \Rightarrow b' \qquad b \sim b'}{\Gamma; \mathcal{C} \vdash c \Leftarrow b \dashv [\bigcirc \mid \emptyset]}$$

And for unannotated functions:

$$\text{AFun}\frac{\Gamma, x : \tau_1; \mathcal{C} \circ (\lambda x.\ \bigcirc) \vdash e \Leftarrow \tau_2 \dashv [\varsigma \mid \gamma, x : \tau_1]}{\Gamma; \mathcal{C} \vdash \lambda x.\ e \Leftarrow \tau_1 \to \tau_2 \dashv [\lambda x.\ \varsigma \mid \gamma]}$$

$$\text{AFun}\frac{\Gamma, x : \tau_1; \mathcal{C} \circ (\lambda x.\ \bigcirc) \vdash e \Leftarrow \tau_2 \dashv [\varsigma \mid \gamma] \qquad x \notin \text{dom}(\gamma)}{\Gamma; \mathcal{C} \vdash \lambda x.e \Leftarrow \tau_1 \to \tau_2 \dashv [\lambda \square.\ \varsigma \mid \gamma]}$$

As it turns out, variables are never added to γ so the first version of the rule above cannot happen. The rule is stated as this may not necessarily be the case in general and for consistency with *criterion 3*. Finally, function applications:

$$\text{AApp}\frac{\tau \neq ? \qquad \Gamma \vdash e_1 \Rightarrow \tau_1 \qquad \tau_1 \blacktriangleright_{\to} \tau_2 \to \tau' \qquad \tau \sim \tau' \qquad \Gamma; \mathcal{C} \circ (e_1(\bigcirc)) \vdash e_2 \Leftarrow \tau_2 \dashv [\varsigma \mid \gamma]}{\Gamma; \mathcal{C} \vdash e_1(e_2) \Leftarrow \tau \dashv [\square(\varsigma) \mid \gamma]}$$

As with *synthesis slices*, if the checked type is ? then an empty slice should be returned. Intuitively, the checked type is ?, and cannot be further decomposed, so there is not much use in slicing further structure. This has the second purpose of omitting all regions of code that are *dynamically typed*.

I believe, this formulation does indeed find the minimum such slice in the context:

Conjecture 3 (Correctness) *For $\Gamma \vdash e \Leftarrow \tau$ and any context $\mathcal{C}$ and term e' such that $e = \mathcal{C}\{e'\}$. If $\Gamma'; \mathcal{C} \vdash e' \Leftarrow \tau' \dashv \rho$ for $\rho = [\varsigma \mid \gamma]$:*

- *ρ is an analysis slice: $\gamma \vdash \varsigma \Leftarrow \tau'$ and $\Gamma \vdash \mathcal{C}\{\varsigma\} \Leftarrow \tau$*

- *ρ is minimal: if also $\rho' = [\varsigma' \mid \gamma']$ and $\gamma \vdash \varsigma \Leftarrow \tau'$ and $\Gamma \vdash \mathcal{C}\{\varsigma\} \Leftarrow \tau$, then $\rho \sqsubseteq \rho'$.*

Consider if manipulating the context $\mathcal{C}$ is useful, for example if the context includes an annotation, we could find also the *minimum context* such that .

3.1.4 Criterion 3: *good_name_here*

Combines the previous two criteria. The main difficulty is how to combine the two slices during subsumption, kinda extends the previous.

Talk about annotations as analysis slices?

The objective of this criterion is to determine which parts of an expression required to be typed checked. Therefore, it omits the sub-terms which could be changed to a different type, but still have the whole expression check against the same type.

Give example here with highlighting.

Formally, this criterion considers expressions e type checked against a type τ under typing context Γ:

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

And finds the minimum slice of e in this context such that

3.1.5 Join Types

The Hazel core calculus is very primitive, only consisting of *base types*, *annotations*, and *functions*. Extensions to gradual types [**GradualJoins**], and Hazel [**MarkedLocalisation**][7]: *if* expressions, *pattern matching*, *sum types* etc. all require[8] *join types*.

[7] Here as the *meet* of the opposite of my *precision* order.

[8] Or are easiest formulated with.

A join of two types $\tau_1 \sqcup \tau_2$ (if one exists) is the least precise (most general) type that more precise than both τ_1, τ_2: that $\tau_1 \sqsubseteq \tau_1 \sqcup \tau_2$ and $\tau_1 \sqsubseteq \tau_1 \sqcup \tau_2$. Therefore, the join is therefore is consistent with both τ_1, τ_2. Type consistency can be reformulated in terms of joins: τ_1, τ_2 are consistent *if and only if* they have a join. This is the *order-theoretic* **(cite)** join with respect to the precision partial order on types. For example, the type of an *if statement* would be the join of the types of it's branches.

These add an additional way to generate a *new type* other than by synthesis or from annotations. *add some machinery to demonstrate how a program slice could be constructed by taking the 'deepest' branch in the join and working out if it was the left/right branch etc.*

3.2 Cast Slicing Theory

Fairly trivial, just treat slices as types and decompose accordingly. The whole reason of indexing by type was to allow this.

The idea of it being a minimal program slice producing the same cast doesn't really work here due to dynamics. Explore the maths of this. Either way, it is a useful construct in practice. Exploring this in more detail, looking at *dynamic program slicing* could be a good future direction.

3.2.1 Indexing Program Slices by Types

3.2.2 Elaboration

3.2.3 Dynamics

3.2.4 Cast Dependence

This in combination with the indexed slices could have some nice mathematical properties.

Though, these would need to retrieve information about parts of slices that were lost when decomposing slices. i.e. when a slice $\tau_1 \to \tau_2$ extracts the argument type τ_2, the slicing criterions lose track of the original lambda binding. A way to reinsert these bindings such that we get a minimal term which *Evaluates to the same cast* e.g. But this will have lots of technicalities with correctly tracking the restricted contexts γ' for closures (i.e. functions returning functions which have been applied once). **TALK ABOUT THIS IN THE FURTHER DIRECTIONS SECTION.**

3.3 Proofs

Do if there is time. Prove that analysis slice contexts actually make sense, that they maintain a valid term that is *still* minimal.

3.4 Type Slicing Implementation

3.4.1 Type Slice Data-Type

Detail initial implementation (just tagging existing types). Compare with final implementation, quantitative numbers for im-

prove could be obtained but would require quite a bit of coding work... Polymorphic Variants :))

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).

Mention that full slices as in the theory is very inefficient.

Integration with Existing Type Data-Type

Use of 'Typ. Explain how this allows it to easily be disabled and saves space (maybe).

Synthesis & Analysis Slices

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

Mapping Functions

Type Slice Joins

Just unions of the lists. Double check that we can't have elements from more than one branch highlighted.[9]

3.4.2 Static Type Checking

Detail the statics and Info types. Explain the distinction between Self.re and Mode.re, which links very nicely with the synthesis and analysis slice distinctions.

[9] Type variables might make this possible??

3.4.3 Elaboration

Make casts use type slices, insertion is mostly the same. Just need to ensure that the slices are preserved (i.e. types are threaded through without being replaced)

3.4.4 User Interface

Click on analysis or synthesis slices from context inspector

3.5 Cast Slicing Implementation

3.5.1 Cast Transitions

Cast transition and unboxing logic

3.5.2 User Interface

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

3.6 EV_MODE Evaluation Abstraction

Very complex!!!

3.7 Indeterminate Evaluation

3.7.1 Futures Data-Type

Lazy lists, often infinite

3.7.2 Hole Instantiation

Small Hole hypothesis, quick check

Choosing which Hole to Instantiate

Use EV_Mode to select next hole to instantiate

Synthesising Terms for Types

Difficulties instantiating strings...

Substituting Holes

Detail that this was an unexpected extra task, and is therefore not exactly the same as hole substitution as detailed in Preparation (i.e. no metavars or contexts annotated on holes, but it is enough for the search procedure to work)

3.7.3 Cast Laziness

Ref the original cast slicing paper, which is not lazy apparently? Laziness sort of breaks the idea that runtime errors evaluate to cast errors. There can be compound values of the wrong type being cast, but the error will only be found upon accessing parts of the compound type.

Making casts eager is a major change to the actual transitions.

Eager casts also catch 'spurious' errors (see Evaluation).

3.7.4 User Interface

3.8 Evaluation Stepper

3.8.1 Evaluation Contexts

3.8.2 Customisable Hole Instantiation

3.8.3 User Interface

3.9 Search Procedure

3.9.1 Detecting Relevant Cast Errors

i.e. failed cast at head of term

3.9.2 Filtering Indeterminate Evaluation

Done via EV_MODE similarly to finding which hole to instantiate.

3.9.3 Iterative Deepening

Required after evaluating that infinite loops break the thing

3.9.4 User Interface

Chapter 4

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.

4.1 Goals

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

4.2 Hypotheses

Various hypotheses for results, mentioned below.

4.3 Program Corpus Collection

I need a corpus of programs with type errors and a corpus of programs with annotations (which may be well-typed). Currently have neither!! Think I might need help with this...

Both the above could be transpiled from OCaml, ill-typed ones by just ignoring all types (though this adds some bias to the search procedure, by ignoring partially annotated examples or annotated but ill-typed situations, but it *should* be roughly ok). But sum types etc. may be harder to transpile.

4.3.1 Methodology

4.3.2 Alternatives

OCaml $\rightarrow$ Hazel transpiler

4.4 Effectiveness Analysis

4.4.1 Search Procedure

Witness Coverage

Describe reasons for failure in next section

Code Coverage

Were some branches not taken? 'Solvable' via symbolic execution

Trace Size

Larger trace sizes are harder to comprehend? Cite...

Cast Slice Size

Common complaints on error slices are large slice sizes. cite... Does this correlate with trace size?

4.4.2 Type Slicing

Correctness

?

Code Slice Size

Compare large theory slices with the 'simplified' slices.

4.5 Performance Analysis

4.5.1 Search Procedure

Time

Space

Lazy list stuff in particular...

4.5.2 Slices

Time

Very slice when used continuously... This might be due to bad design or bugs?

Space

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

Cast slices should be small as they only use sub-parts of terms and mostly exist at the leaves, or are incremental parts (i.e. to [?]). Compare the slice size of *casts* vs *type* slices.

4.6 Critical Analysis

4.6.1 Slicing

Which constructs are ignored in simplified slices? Why? Do they have the biggest impact on size?

Discuss the usability aspects of the 4 different type slicing ideas. Discuss

4.6.2 Cast Laziness

Annotations will create casts, but in many cases these annotations will never actually be used. i.e. on a product where only the 2nd element is ever used, a cast error on the 1st element is still caught. Eager casts catch these errors, lazy casts do NOT.

4.6.3 Cast Errors during Evaluation

Cast errors may appear during evaluation, but subsequently ignored as they may be discarded upon further evaluation. i.e. we get to a safe result, even though a cast error appeared on the way. It is debatable if this should count as an error. Should be easy to implement a version which catches this and do some tests?

4.6.4 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).

4.6.5 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.

4.6.6 Bidirectional Type Error Localisation

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

4.6.7 Improving Hole Instantiation

To improve code coverage. i.e. Strings and Floats are annoying, SMT solvers could be used to help explore branches... Equality solvers in particular would be very useful and quite easy to implement, in particular for strings which are generally used via equality as opposed to lists which are used incrementally via pattern matching.

4.6.8 Combinatorial Explosion

State space gets very large as more holes are instantiated from other holes, i.e. $[] \to [?] \to [?, ?] \to [?, ?, ?]$...

Chapter 5

Conclusions

5.1 Conclusion

Conclusions Here.

5.2 Further Directions

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

5.2.1 Extension to Full Hazel Language

Type functions, recursive types, deferrals

5.2.2 Cast Slicing

Cast slicing here doesn't extend the nice properties I have with type slicing: that a program slice will synthesise/analyse the same type. It might be possible to have a system that does?

The implementation differs by storing slices directly in the context, making analysis slices.

Also, the fact that cast slices are manipulated in evaluation is completely opaque to the user *unless* they step through manually. Ways to better represent this information exist (cast dependency graphs etc.)

Proofs

PRimarily about the search procedure & casts.

Also about creating a *stronger* link between static type errors and runtime errors, what Hazel considers as a runtime error is somewhat unintuitive, see the points made in evaluation. A formal property of what exactly a type error witness *is* which actually matches with the implementation would be nice.

User Studies

Effectiveness gauge in the real world

5.2.3 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

5.2.4 Let Polymorphism & Global Inference

Errors with global inference errors are often more subtle, where trace visualisation really shines.

Constraint Slicing

To allow slices to work with constraint solvers. One error slicing paper already does this.

Gradual Type Inference

Miyazaki has a good paper on this. Seems like it could work well in Hazel, though at the loss of parametricity.

Localisation

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

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

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

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.

$$
\begin{array}{lll}
[\![d/u]\!]c & = c & \\
[\![d/u]\!]x & = x & \\
[\![d/u]\!]\lambda x : \tau.d' & = \lambda x : \tau.[\![d/u]\!]d' & \\
[\![d/u]\!]d_1(d_2) & = ([\![d/u]\!]d_1)([\![d/u]\!]d_2) & \\
[\![d/u]\!](\!|)^u_\sigma & = [[\![d/u]\!]\sigma]d & \\
[\![d/u]\!](\!|)^v_\sigma & = (\!|)^b_{[\![d/u]\!]\sigma} & \text{if } u \neq v \\
[\![d/u]\!](\!|d'|\!)^u_\sigma & = [[\![d/u]\!]\sigma]d & \\
[\![d/u]\!](\!|d'|\!)^v_\sigma & = (\!|[\![d/u]\!]d'|\!)^b_{[\![d/u]\!]\sigma} & \text{if } u \neq v \\
[\![d/u]\!]d'\langle\tau \Rightarrow \tau'\rangle & = ([\![d/u]\!]d')\langle\tau \Rightarrow \tau'\rangle & \\
[\![d/u]\!]d'\langle\tau \Rightarrow ? \nRightarrow \tau'\rangle & = ([\![d/u]\!]d')\langle\tau \Rightarrow ? \nRightarrow \tau'\rangle &
\end{array}
$$

$\boxed{[\![d/u]\!]\sigma = \sigma'}$ 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

Slicing Theory

B.1 Precision Relations

B.1.1 Patterns

B.1.2 Types

B.1.3 Expressions

B.2 Program Slice Typing

B.3 Criterion 1: Synthesis Slices

$\boxed{\Gamma \vdash e \Rightarrow \tau \dashv \rho}$ e synthesising type τ under context Γ produces minimum synthesis slice ρ

$$\text{SConst}\frac{}{\Gamma \vdash c \Rightarrow b \dashv [c \mid \emptyset]} \qquad \text{SVar}\frac{x : \tau \in \Gamma}{\Gamma \vdash x \Rightarrow \tau \dashv [x \mid x : \tau]} \qquad \text{SVar?}\frac{x : ? \in \Gamma}{\Gamma \vdash x \Rightarrow \tau \dashv [\square \mid \emptyset]}$$

$$\text{SFun}\frac{\Gamma, x : \tau_1 \vdash e \Rightarrow \tau_2 \dashv [\varsigma \mid \gamma, x : \tau_1]}{\Gamma \vdash \lambda x : \tau_1.\ e \Rightarrow \tau_1 \to \tau_2 \dashv [\lambda x : \tau_1.\ \varsigma \mid \gamma]}$$

$$\text{SFunConst}\frac{\Gamma, x : \tau_1 \vdash e \Rightarrow \tau_2 \dashv [\varsigma \mid \gamma] \qquad x \notin \mathrm{dom}(\gamma)}{\Gamma \vdash \lambda x : \tau_1.\ e \Rightarrow \tau_1 \to \tau_2 \dashv [\lambda \square : \tau_1.\ \varsigma \mid \gamma]}$$

$$\text{SApp}\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \dashv [\varsigma_1 \mid \gamma_1]}{\Gamma \vdash e_1(e_2) \Rightarrow \tau[\varsigma_1(\square) \mid \gamma_1]} \qquad \text{SEHole}\frac{}{\Gamma \vdash (\!|\,|\!)^u \Rightarrow ? \dashv [\square \mid \emptyset]}$$

$$\text{SNEHole}\frac{\Gamma \vdash e \Rightarrow \tau \dashv}{\Gamma \vdash (\!|e|\!)^u \Rightarrow ? [\square, \emptyset]} \qquad \text{SAsc}\frac{\Gamma \vdash e \Leftarrow \tau}{\Gamma \vdash e : \tau \Rightarrow \tau \dashv [\square : \tau \mid \emptyset]}$$

Figure B.1: *Minimum synthesis slice* calculation

$\boxed{\Gamma \vdash e \dashv\!\Rightarrow \upsilon[\rho]}$ e synthesising type υ $\blacktriangleright_{type}$ τ under context Γ produces minimum type-indexed synthesis slice(s) $\upsilon[\rho]$

$$\textsc{SConst}\ \frac{}{\Gamma \vdash c \dashv\!\Rightarrow b[c \mid \emptyset]} \qquad \textsc{SVar}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x \dashv\!\Rightarrow \tau[x \mid x : \tau]}$$

$$\textsc{SFun}\ \frac{\Gamma, x : \tau_1 \vdash e \dashv\!\Rightarrow \upsilon_2[\varsigma \mid \gamma, x : \tau_1]}{\Gamma \vdash \lambda x : \tau_1.\ e \dashv\!\Rightarrow (\tau_1[\square : \tau_1 \mid \emptyset] \to \upsilon_2[e \mid \gamma])[\lambda x : \tau_1.\ \varsigma \mid \gamma]}$$

$$\textsc{SFunConst}\ \frac{\Gamma, x : \tau_1 \vdash e \dashv\!\Rightarrow \upsilon_2[\varsigma \mid \gamma] \qquad x \notin \mathrm{dom}(\gamma)}{\Gamma \vdash \lambda x : \tau_1.\ e \dashv\!\Rightarrow (\tau_1[\square : \tau_1 \mid \emptyset] \to \upsilon_2[e \mid \gamma])[\lambda \square : \tau_1.\ \varsigma \mid \gamma]}$$

$$\textsc{SApp}\ \frac{\Gamma \vdash e_1 \dashv\!\Rightarrow \upsilon_1[\varsigma_1 \mid \gamma_1]}{\Gamma \vdash e_1(e_2) \dashv\!\Rightarrow \upsilon[\varsigma_1(\square) \mid \gamma_1]} \qquad \textsc{SEHole}\ \frac{}{\Gamma \vdash (\!|)^u \dashv\!\Rightarrow ?[\square \mid \emptyset]}$$

$$\textsc{SNEHole}\ \frac{}{\Gamma \vdash (\!|e|\!)^u \dashv\!\Rightarrow ?[\square \mid \emptyset]} \qquad \textsc{SAsc}\ \frac{}{\Gamma \vdash e : \tau \Rightarrow \tau[\square : \tau \mid \emptyset]}$$

Figure B.2: *Minimum synthesis slice* calculation retaining subslices indexed on types

B.4 Criterion 2: Analysis Slices

B.5 Criterion 3: ...

B.6 Elaboration

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

Index

Proposal

Description

This project will add some features to the Hazel language [**Hazel**]. 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. [**SearchProc**]. 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 [**Blame**], error and dynamic program slicing [**ErrSlice**, **DynProgSlice**], 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 [**Hazel**] 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 [**HazelLivePaper**].
- **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 [**HazelCode**].
- 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.