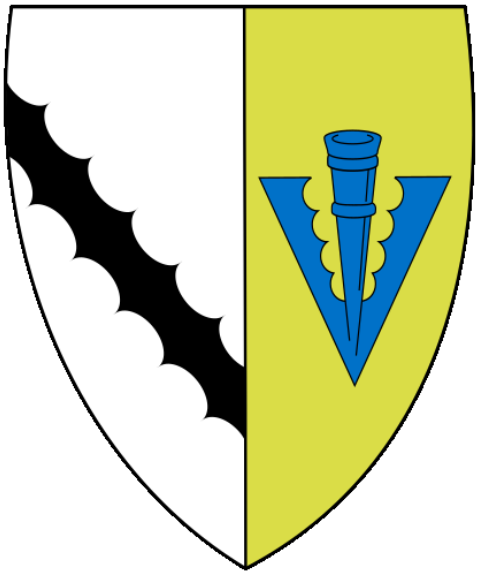
Max Carroll

Type Error Debugging in Hazel

Computer Science Tripos, Part II



Sidney Sussex College College
University of Cambridge
April 16, 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:	**22433** [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.

Special Difficulties

Any Special Difficulties encountered

Acknowledgements

Acknowledgements Here.

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

[2] *Calculation method here*

Contents

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 [18], with programmers spending a *highly skewed* proportion of their time identifying and resolving a small proportion of *difficult* bugs [11].

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 [20]. 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. Additionally, they don't generally specify any source code context which caused them. Instead, a dynamic type error is accompanied by an *evaluation trace*, which can be *more intuitive* [35] by demonstrating concretely why values are *not* consistent with their expected type as required by a *runtime cast*.

This project seeks to improve user understanding of type errors by localising static type errors *more completely*, and *combining* the benefits of static and dynamic type errors.

I consider three research problems and implement three features to solve them in the Hazel language [1]:[1]

1. Can we statically highlight code which explains *static type errors* more *completely*, including all code that *contributes* to the error?

   This would alleviate the issue of static errors being *incorrectly localised*, and help give a *greater context* to static type errors.

   Solution: I devise a *novel* method of **type slicing** including formal mathematical foundations built upon the formal *Hazel calculus* [16]. Additionally, it generalises to highlight all code relevant to typing *any* expressions (not just erroneous expressions).

2. Can we dynamically highlight source code which contributes to a *dynamic type error*?

[1] The answers may differ greatly for other languages. Hazel provides a good balance between complexity and usefulness.

Figure 1.1: A Static Type Error

This would provide missing source code context to understand how types involved in a dynamic type error originate from the source code.

Solution: I devise a *novel* method of **cast slicing**, also with formal mathematical foundations. Additionally, it generalises to highlight source code relevant to requiring any specific *runtime casts*.

3. Can we provide dynamic *evaluation traces* to explain *static type errors*?

   This would provide an *intuitive* concrete explanation for static type errors.

   Solution: I implement a **type error witness search procedure**, which discovers inputs (witnesses) to expressions which cause a *dynamic type error*. This is based on research by Seidel et al. [24] which devised a similar procedure for a subset of OCaml.

Hazel [1] is a functional locally inferred and gradually typed research language allowing the writing of *incomplete programs* under active development at the University of Michigan. Being gradually typed, allowing both static and dynamic code to coexist, this project successfully demonstrates the utility of these three features in improving understanding of *both* static and dynamic errors. Both, to a *greater extent* than in any existing languages, with the explanations being *useful* in debugging type errors, and is expected to reduce debugging times and aid in students understanding type systems.[2]

Example: *A basic well-formatted **graphical** walk-through for how a static type error could be diagnosed using the three features. Take simplest example from the evaluation.*

1.1 Related Work

There has been extensive research into attempting to understand *what* is needed [29], *how* developers fix bugs [13], 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 (**cite the directions**) but generally as a *teaching language* (**cite the explainthis paper**) 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 [73], though my definition of program slices matches more with functional program slices [34]. The properties I explore are more similar to *dynamic program slicing* [70] but with type characteristics rather than evaluation characteristics, in a sense similar *type error slicing* [55].

The *type witness search procedure* is based upon Seidel et al. [24], 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.

[2] Proving this would require a user study.

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 basic background on the *Hazel implementation* (section 2.1.3). Additionally, methods of non-deterministic programming are introduced (section 2.1.4).

Section 3.1 and section 3.2 formalise and prove[3] the ideas of *type slices* and *cast slices* in the Hazel core calculus.

An implementation (section 3.3-3.4) of these has been created covering *most*[4] of the Hazel language, including a user interface.

A *type witness search procedure* was successfully implemented (section 3.7). This involved created a non-deterministic evaluation method (section 3.6) abstracting search order and search logic.

The slicing features and search procedure met all goals (section 4.1), showing *effectiveness* (section 4.6) and being reasonably *performant* (section 4.5) over a corpus of *well-typed* and *ill-typed* Hazel programs respectively (section 4.4). Further, the features weaknesses were evaluation, with considered improvements implemented or devised (section 4.7). Additionally, a holistic evaluation was performed (**??**), considering the use of all the features, in combination with Hazel's existing error highlighting, for debugging both static and dynamic type errors.

Finally, further directions and improvements have been presented in chapter 5.

[3] TODO

[4] 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 [22]. For example:

- Exp e – e is an *expression*
- n : `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] [52, ch. 9] has a typing rule for lambda expression and application as follows:

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

Meaning, $\lambda x.e$ 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 [45, 6]. Of particular relevance to this project, execution traces are known to help provide insight to errors [35], 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 type*[2] [67]. 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 `?`.

[1] Simply typed lambda calculus.

[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.

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* [48] in *contract* systems [51]:

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

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

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

Gradual Type Systems

A *gradual type system* [26, 46] 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 `++ "str"`. 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:

[4] Wrapped in a cast between function types.
[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]:

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

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* [52, ch. 15] with a *top* type $\top$, but with symmetry instead of transitivity.

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

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

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

But, for evaluation to work the static type information needs to be encoded into casts to be used in the dynamic internal language, for which the evaluation semantics are defined. This is done via *elaboration*, similarly to Harper and Stone's approach to defining (globally inferred) Standard ML [58] by elaboration to an explicitly typed internal language XML [64]. The *elaboration judgement* $\Gamma \vdash e \rightsquigarrow e' : \tau$ read as: external expression e is elaborated to internal expression d with type τ under typing context Γ. For example we need to insert casts around function applications:

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

If, e_1 elaborates to d_1 with type $\tau_1 \sim \tau_2 \to \tau$ and e_2 elaborates to τ_2' with $\tau_2 \sim \tau_2$ then we place a cast[11] on the function d_1 to $\tau_2 \to \tau$ and on the argument d_2 to the function's expected

[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 $?$.

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

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

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 [21].

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*.

The *refined criteria* for gradual typing [26] 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* [27] takes on a more algorithmic definition of typing judgements, being more intuitive to implement. They also allow some amount of local type inference [59], allowing programmers to omit *type annotations*, instead type information. Global type inference systems [52, ch. 22] can be *difficult to implement*, often via constraint solving [7, ch. 10], and difficult or impossible to *balance* with complex language features, for example global inference in System F (2.1.1) is undecidable [61].

This is done in a similar way to annotating logic programming [74, p. 123], 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* [72]. 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:

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

[12] Highlighted in red as an error.

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

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

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

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

Contextual Modal Type Theory

Not *hugely* relevant really...

System F

Very brief explanation with less/no maths

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 [16] 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 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*.

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

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

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

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

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

External Language

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

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

One notable consequence of combining gradual and bidirectional typing is that the *subsumption rule* in bidirectional 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, $\blacktriangleright_\to$ 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) [43].[17]

The type assignment judgement $\Delta; \Gamma \vdash d : \tau$ means that d has type τ under typing and hole contexts Γ, Δ. The rules for holes take their types from the hole context and ensure that

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

[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 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 [54]. Additionally, ground types and a matching function are defined in Figs. A.8 & A.11, and typing of hole environments/substitution in Fig. A.7

Elaboration

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

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

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

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

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

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

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

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

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.

[18] With respect to the original typing context captured by the hole.
[19] Potentially leading to incorrect cast insertion.
[20] Between function types

- *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.

Dynamics

A small-step contextual dynamics [22, ch. 5] is defined on the internal expressions to define a *call-by-value*[21] **ENSURE THIS** evaluation order.

Like the *refined criteria* [26], 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)*

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.

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

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 [2] 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.

Discuss why the main branch was chosen over THI, hole substitution one etc. (main point being better documentation, sum types, though still very lacking

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?

Move most of this to implementation, give vague description/motivation and emphasise it's complexity

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] [8], 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;
```

[26] Having no side effects.

[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 `req`, which the abstraction may choose to recursively evaluate, and bind a resulting value for use in calculating the next step.

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

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

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

- **Final Form Checker** – Returns whether a term is one of each of the final form. Using the evaluator abstraction with this means there is no need to maintain a separate syntactic value checker, instead it is derived directly from the evaluation transitions. Yet, it is still syntactic since the abstraction does not actually perform evaluation steps and continue evaluation, instead it just makes the step accessible[29] to the implementation.
- **Evaluator** – Maintains a stack machine and actually performs the reduction steps.
- **Stepper** – Returns a list of possible evaluation steps in terms of *evaluation contexts* under a non-deterministic evaluation method **Explain how `EvalCtx.t => EvalCtx.t` in `req_final` allows this**. The evaluation order can then be user-controlled.

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 Bounded Non-Determinism

The search procedure [24] is effectively a *dynamic type-directed* test generator, attempting to find dynamic type errors. In Hazel, dynamic type errors will manifest themselves as *cast errors*.

Type-directed generation of inputs and searching for one which manifests an error is a *non-deterministic* algorithm [75].

[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.
[30] Mapping variable bindings.

High Level

Non-determinism can be declaratively represented by two ideas:

- *Choice (`|||`):* Determines the search space, flipping a coin will return heads *or* tails.
- *Failure (`fail`):* The *empty* result, no solutions to the algorithm.

Suppose the result of the algorithm has type τ. These can be represented by operations:

$$|||_\tau : \tau \to \tau \to \tau$$

$$\texttt{fail}_\tau : \tau$$

Where `|||` should be *associative* and `fail` should be a *zero element*, forming a *monoid*:

$$x ||| (y ||| z) = (x ||| y) ||| z \qquad \texttt{fail} ||| x = x = x ||| \texttt{fail}$$

That is, the order of making binary choices does not matter, and there is no reason to choose failure.

There are many proposed ways to manage programs with choice:

Logic Programming: Languages like Prolog **(cite)** and Curry **(cite)** express non-determinism by directly implementing *choice* with non-deterministic evaluation. Prolog searches via backtracking, while Curry abstracts the search procedure.

There are ways to embed this within OCaml. Inspired by Curry, Kiselyov [14] created a tagless final style [32] domain specific language within OCaml. This fully abstracts the search procedure from the non-deterministic algorithm definition.

Continuations:

Effect Handlers: Effect handlers allow the description of effects and factors out the handling of those effects. Non-determinism can be represented by an effect consisting of the choice and fail operators [4, 28], while handlers can flexibly define the search procedure and accounting logic, e.g. storing solutions in a list.

In order to enumerate try multiple solutions, *multiple continuations* must be used. JSOO[31] does not (at time of writing) allow these.

Monadic: The non-determinism effect can be expressed as a monad. A monad is a parametric type $m(\alpha)$ equipped with two operations, `return` and `bind`, where `bind` is associative and `return` acts as an identity with respect to `bind`: A monad can then be extended to represent nondeterminism by adding a `choice` and `fail` operator satisfying the usual laws:

$$\texttt{choice} : \forall \alpha.\ m(\alpha) \to m(\alpha) \to m(\alpha)$$

$$\texttt{fail} : \forall \alpha.\ m(\alpha)$$

Where `bind` distributes over `choice`, and `fail` is a left-identity for `bind`.

$$\texttt{bind}(m_1 ||| m_2)(f) = \texttt{bind}(m_1)(f) ||| \texttt{bind}(m_2)(f)$$

[31] Javascript of OCaml. A dependency of Hazel.

A monad m, is a parameterised type with operations:

$$\texttt{bind} : \forall \alpha, \beta.\ m(\alpha) \to (a \to m(\beta)) \to m(beta)$$

$$\texttt{return} : \forall \alpha.\ a \to m(\alpha)$$

Satisfying the monad laws:

$$\texttt{bind}(\texttt{return}(x))(f) = f(x)$$

$$\texttt{bind}(m)(\texttt{return}) = m$$

$$\texttt{bind}(\texttt{bind}(m)(f))(g) = \texttt{bind}(m)(\texttt{fun}x \to \texttt{bind}(f(x))(g))$$

Figure 2.2: Monad Definition

$$\texttt{bind}(\texttt{fail})(f) = \texttt{fail}$$

In this context, bind takes a non-deterministic choice and a function which maps each *guess* in the state space to another choice, returning this choice of all the possible resulting choices after applying the function to the possible input choices. See how distributivity represents this interpretation. `fail` being the left identity of `bind` states that you cannot make any guesses from the no choice (`fail`). If a potential guess does not solve the problem, then just return `fail` (hence the name). For example:

Figure 2.3: Non-determinism Monad Example

While the `bind` and `return` operators are *not* required to represent non-determinism, they are a *familiar*[32] way to represent a large class of effects in general way.

Low Level

Which may abstract over different search/backtracking procedures. Writing these directly can be complex, unintuitive, and require much code, but is very flexible. **(cite evidence/reasoning)**:

Depth First Search: Generate options where choice is possible, and backtrack once each option has been fully tested.

Breadth First Search:

Iterative Deepening:

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

[32]For OCaml developers and functional programmers.

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. [24] and the Hazel core language [16] 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.

Git. Merging from dev frequently (many of which were very difficult, as my code touches all of type checking and much of dynamics; and some massive changes, i.e. UI update).

Talking with devs over slack.

Using existing unit tests & tests in web documentation to ensure typing is not broken.

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!

Motivate and pose questions for type slicing and cast slicing.

3.1 Type Slicing Theory

Might be worth deferring even more mathematical definitions to the appendices in favour shorter worded definitions in this section

Replace all occurrences of 'typing context' with 'typing assumptions' to avoid name clash with expression/program contexts.

I develop a novel method I term *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 *typing slices*.

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 contribute to the a term's type.

I would like to stress that the second and third criterion were *very challenging* to formalise, requiring extensive mathematical machinery: *context slices* (section 3.1.2), *type flows* (**ref appendix**), *checking context* (definition 3), *type-indexed slices* (section 3.1.3).

Clarify on and ensure terminology is consistent (typing vs checking/analysis vs synthesis)

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

PLACE ALL IMPORTANT DEFINITIONS INSIDE DEFINITION ENVIRONMENTS

3.1.1 Expression Typing slices

A *expression typing slice* ρ, is a pair ς^{γ}, consisting of an *expression slice* ς and *typing context slice* γ 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 [34].[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:

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

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:

$$\upsilon ::= \square \mid \texttt{?} \mid b \mid \upsilon \to \upsilon$$

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:

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

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 upper bounds of precision chains.

An expression slice ς *of* e is a slice such that $e \sqsubseteq e$.

[1] One of the three slicing mechanisms.

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

Typing 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, $\emptyset$. These are notated γ and if $\gamma \subseteq \Gamma$ then γ is a slice *of* Γ.

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

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

expression typing 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 $[\square, \emptyset]$. Similarly, an element ρ in P_e^Γ can referred to as a expression typing slice *of* e under Γ.

3.1.2 Context Typing Slices

add pattern context.

Formally, an *expression context* $\mathcal{C}$ is an expressions with *exactly one* sub-term marked as $\bigcirc$:[3]

$$\mathcal{C} ::= \bigcirc \mid \lambda x : \tau.\ \mathcal{C} \mid \lambda x.\ \mathcal{C} \mid \mathcal{C}(e) \mid e(\mathcal{C}) \mid \mathcal{C} : \tau$$

Where $\mathcal{C}\{e\}$ substitutes expression e for the mark $\bigcirc$ in $\mathcal{C}$, the result of this is necessarily an expression. Additionally, contexts are composable: substituting a context into a context, $\mathcal{C}_1\{\mathcal{C}_2\}$ produces another valid context, notate this by $\mathcal{C}_1 \circ \mathcal{C}_2$[4].

Similarly to expressions, contexts can be extended to *context slices* by allowing slices within:

$$c ::= \bigcirc \mid \lambda p : \upsilon.c \mid c(\varsigma) \mid \varsigma(c) \mid c : \upsilon$$

However, the precision relation $\sqsubseteq$ is defined differently, requiring that the mark $\bigcirc$ must remain in the same position in the context structurally speaking. For example $\bigcirc(\square) \sqsubseteq \bigcirc(1)$, but $\bigcirc \not\sqsubseteq \bigcirc(1)$. This can be concisely defined by *extensionality* (**cite**):

Definition 1 (Context Precision) *If c' and c are context slices, then $c' \sqsubseteq c$ if and only if, for all expressions e, that $c'\{e\} \sqsubseteq c\{e\}$.*

Again, we refer to a context slice c of $\mathcal{C}$ as one satisfying that $c' \sqsubseteq \mathcal{C}$.

We also get that filling contexts preserves the precision relations both on expression slices *and* context slices:

Conjecture 1 (Context Filling Preserves Precision) *For expression slice ς and context slice c. Then if we have slices $\varsigma' \sqsubseteq \varsigma$, $c' \sqsubseteq c$ then also $c'\{\varsigma'\} \sqsubseteq c\{\varsigma\}$.*

Therefore, context slices c can be though of as monotone function (**cite**)...

Show that composition is also preserved over precision, i.e. it is a functor.

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

[4] Context can alternatively be though of as functions from expressions to expressions.

Make some references to category theory, i.e. category of slices with morphisms being context slices. Or that slices form categories on expression e and contexts are a functor between expression typing slice categories. i.e. contexts are monotonic functions

rewrite The accompanying notion of a *typing assumption slice* can be extended to *functions on typing assumptions*. This function can represent what typing assumptions need to be **added**, or can be *removed* when placing e in it's context slice.

Functions can be composed and a precision partial order can also be defined via extensionality:

Definition 2 (Function Precision) *If f' and f are functions, then $f' \sqsubseteq f$ if and only if, for all typing contexts Γ, that $f'(\Gamma) \subseteq f(\Gamma)$.*

Again, such functions are monotone **(find the name of this mathematical property... sorta an extended monotonicity)**:

Conjecture 2 (Function Application Preserves Precision) *For typing context Γ and function f. Then if we have slices $\Gamma' \subseteq \Gamma$, $f' \sqsubseteq f$ then also $f'(\Gamma') \sqsubseteq f(\Gamma)$.*

This pair of context slice and function f will be referred to as a *context slice* and notated ρ^f, should f be the identity it may be omitted in shorthand.

Extend composition to program contexts and substitution of expression typing slices. Again as slices are a superset of expressions, then this extends to expression etc.

When

3.1.3 Type-Indexed Slices

Do type-indexed expressions slices actually need contexts on the sub-slices? Are these every not claculable from the rule they are being used in

Have context i.e. contexts with a _ and let the syntax write it directly as a *type-indexed context slice*, i.e. a type-indexed slice, but with the slice ρs replaced with ρs. Then allow easy syntax to create from a type indexed slice i.e.

$$(\rho\{? \mid \rho\})(\bigcirc) = \rho\{? \mid \rho(\bigcirc)\}$$

Reverse composition order for this?? Also, syntax for going from indexed context to indexed expression i.e. {}

For *criteria 3 and cast slicing*, there is a need to decompose slices to find sub-slices which contribute to specific portions of a compound type. For example, which part of the expression typing slice was related to the argument type of a function specifically.

Give EXAMPLE

A *type(-indexed) slice* s consists of: a expression typing slice, a context slice, and a *type index* i. This index is either an *atomic* type label or is *compound*, consisting of type slices conforming to the structure of types:

$$i ::= ? \mid b \mid s \to s$$

$$s ::= \rho\{i \mid \rho\}$$

The type that a type slice s *represents* is the slice retaining only it's type labels. This will be notated by $[\![s]\!]$, defined inductively:

$$[\![p\{? \mid \rho\}]\!] = ? \quad [\![p\{b \mid \rho\}]\!] = b$$

$$[\![p\{s_1 \to s_2 \mid \rho\}]\!] = [\![s_1]\!] \to [\![s_2]\!]$$

This same notation will also be used to extract the type of a type index i, similarly defined.

A term e in some context $\mathcal{C}$ will be associated with a *type slice* with the meaning that ρ contains a expression typing slice for typing e and p contains a context slice for typing e in context $\mathcal{C}$ according to some criterion. Typically the context slice would correspond to type analysis and the expression typing slice would correspond to type synthesis.

The compound type slices must satisfy the crucial property that the sub-terms are sub-*slices* of ρ. That is:

$$p\{p_1\{i_1 \mid \rho_1\} \to p_2\{i_2 \mid \rho_2\} \mid \rho\} \implies p_1\{\rho_1\} \sqsubseteq \rho \text{ and } p_2\{\rho_2\} \sqsubseteq \rho$$

The precision relation can be extended to slices pointwise upon the expression typing slice and context slice for atomic types a (i.e. $a ::= ? \mid b$):

$$p'\{a \mid \rho'\} \sqsubseteq p\{a \mid \rho\} \iff p' \sqsubseteq p \text{ and } \rho' \sqsubseteq \rho$$

And recursively for compound slices:[5]

$$p'\{s_1' \to s_2' \mid \rho'\} \sqsubseteq p\{s_1 \to s_2 \mid \rho\} \iff p' \sqsubseteq p,\ \rho' \sqsubseteq \rho,\ s_1' \sqsubseteq s_1,\ \text{and } s_2' \sqsubseteq s_2$$

Composition of expression typing slices to the outer context is possible, $(p' \circ p)\{i \mid \rho\}$, and is notated shorthand as $p'\{s\}$ for $s = p\{i \mid \rho\}$.

Additionally, if p is empty, $\bigcirc^{\emptyset}$, a type slice may be notated without it: $i \mid \rho$. Equally, when ρ is empty, $\square^{\emptyset}$, then notate $p'\{i\}$. This means that if both p, ρ are both empty then we have i which is syntactically identical to types τ, so we can trivially treat types as *empty type slices*.

Then function matching $\blacktriangleright_{\to}$ can be extended to slices in multiple different ways with different uses depending on the context, see the appendix **ref**.

3.1.4 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 $\Gamma \vdash e \Rightarrow \tau$:[6]

[5] Note, function arguments are *covariant* for this.

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

Conjecture 3 (Uniqueness) *If ρ and ρ' are minimum synthesis slices for $\Gamma \vdash e \Rightarrow \tau$, then $\rho = \rho'$.*

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.

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

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 4 (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 $\rho \sqsubseteq \rho'$.*

Extension to Type-Indexed Slices

This mechanism can be trivially extended to type-indexed slices, where types being synthesised can be replaced by slices directly, i.e. *slice synthesis*. See the appendix **ref AND FIX**.

The only interesting case is the fact that slices for argument types to functions can now be accessed, so must be represented:

3.1.5 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 objective is to show *why* a term is required to be checking against a type.

The useful notion to represent these are *context slices*. It is the terms immediately *around* the sub-term where the type checking is enforced:

For example, when checking this annotated term:

$$(\lambda x.(\!|)^u) : \texttt{Bool} \to \texttt{Int}$$

The *inner hole term* $(\!|)^u$ (underlined from now on) is required to be consistent with `Int` due to the annotation. The context slice will then be:

$$(\lambda\, x\, .\, \underline{(\!|)^u}\,) : \texttt{Bool} \to \texttt{Int}$$

Intuitively, this means that the *contextual* reason for $(\!|)^u$ to be required to be an `Int` is: that it is within a function which the annotation enforced to check against `Bool` $\to$ `Int`, of which only the return type (`Int`) is relevant.

In other words, if the slice was type synthesised, then the hole term would still be *required* to check against `Int`.

For this criterion we consider only the smallest context that resulted in a term being type checked, for example in the following term:

$$\underline{1} : \texttt{Int} : \texttt{?} : \texttt{Bool}$$

The context that enforced 1 to be checked against an `Int` was *only* the inner annotation. We refer to this as the *checking context*. The term when inside it's checking context will always synthesis a type.[7]

These definitions below are verbose and it might be better to just leave it intuitive for this, with definition deferred to appendix.

To represent this, we use a notion of a types *flowing* from other types inside a derivation, i.e. if a type is decomposed, then it's parts *flow* from the complete type. This can be formalised by creating a *type flow* rules **(Ref to appendix)**.[8] Under this flow, every checked type τ' in a derivation has an *origin* synthesis rule producing *first* type which τ' flows from.

Maybe give example?

Then, the *checking context* we want to consider is that of the conclusion of the rule containing the source of the type:

Definition 3 (Checking Context) *For a derivation $\Gamma \vdash e \Rightarrow \tau$ containing a sub-derivation $\Gamma'' \vdash e'' \Leftarrow \tau''$ and where the origin of τ'' is another sub-derivation $\Gamma' \vdash e' \Rightarrow \tau'$. Then either:*

- *e'' is a sub-term of e': the checking context of e'' is $\mathcal{C}$ such that $e' = \mathcal{C}\{e''\}$.*
- *Otherwise, $\Gamma'' \vdash e'' \Rightarrow \tau''$ must be a premise in another rule whose conclusion is a synthesis $\Gamma''' \vdash e''' \Rightarrow \tau'''$ where e' is a sub-term of e'''. The checking context is $\mathcal{C}$ such that $e''' = \mathcal{C}\{e''\}$*

[7] If it analysed against a type, then this would have it's own checking context. We merge these contexts.

[8] This type flow is closely related to mode correctness.

It is clear that this must then be the smallest context containing derivations of both the *checked expression* and it's *origin*. This checking context can be defined more intuitively using rules (**see appendix**) and proven to coincide with the definition above.

An *analysis slice* is a slice of a checked term's *checking context*:

Definition 4 (Analysis Slice) *For a derivation* $\Gamma \vdash e \Rightarrow \tau$*, containing a sub-derivation* $\Gamma' \vdash e' \Leftarrow \tau'$ *with checking context* $\mathcal{C}$*. Then we have that* $\Gamma \vdash \mathcal{C}\{e'\} \Rightarrow \tau''$*.*

An analysis slice of e' *is a program context slice* c^f *such that:*

- c *is a slice of* $\mathcal{C}$*, that is,* $c \sqsubseteq \mathcal{C}$*.*
- f *is a slice of* $\Gamma' \mapsto \Gamma$*.* ***Formalise this better...***
- $c\{e\}$ *synthesises a type consistent*[9] *with* τ'' *under typing context* $f(\Gamma')$ *and still contains the sub-derivation checking* e' *against* τ' *in checking context* c*:*

$$f(\Gamma') \vdash c\{e'\} \Rightarrow \tau''', \quad \tau''' \sim \tau'', \quad \Gamma' \vdash e' \Leftarrow \tau'$$

The *minimum analysis slice* is just the minimum under the precision ordering $\sqsubseteq$. And it must be unique:

Conjecture 5 (Uniqueness) *If* ρ *and* ρ' *are minimum analysis slices for sub-terms* e' *of* e *where* $\Gamma \vdash e \Leftarrow \tau$*, then* $\rho = \rho'$*.*

This can again be represented by a judgement read as, e *which type checks against* τ *in checking context* $\mathcal{C}$ *has analysis slice* $\boldsymbol{\rho}$:

$$\Gamma; \mathcal{C} \vdash e \Leftarrow \tau \dashv \boldsymbol{\rho}$$

There are two situations which enforce *checking contexts*, annotations:

$$\frac{\Gamma \vdash e \Leftarrow \tau}{\Gamma; \bigcirc : \tau \vdash e \Leftarrow \tau \dashv \bigcirc : \tau}$$

And applications, for which we need a slice of the application for the *argument* type of the function, which has previously been devised for *type-indexed synthesis slices* (**ref**):

$$\frac{\Gamma \vdash e_1 \dashv\!\!\Rightarrow \boldsymbol{\rho}\{c_1^{f_1}\{i_1 \mid \varsigma_1^{\gamma_1}\} \to s_2 \mid \rho\} \qquad \Gamma \vdash e_2 \Leftarrow [\![i_1]\!]}{\Gamma; e_1(\bigcirc) \vdash e_2 \Leftarrow [\![i_1]\!] \dashv (c_1\{\varsigma_1\})(\bigcirc)^{f_1}}$$

That is, if e_1 synthesises some type $\tau_1 \to \tau_2$ (i.e. $[\![i_1]\!] \to [\![s_2]\!]$), then the slice for τ_1 in the context as a slice e_1 is $c_1\{\varsigma_1\}$.

Finally, type analysis rules pass the context down (i.e. sub-terms have the same checking context) and also records that the context now needs to remove the $x : \tau_1$ assumption:

$$\frac{\Gamma; \mathcal{C} \vdash \lambda x.\ e \Leftarrow \tau_1 \to \tau_2 \dashv \boldsymbol{\rho}}{\Gamma, x : \tau_1; \mathcal{C} \circ (\lambda x.\ \bigcirc) \vdash e \Leftarrow \tau_2 \dashv \mathrm{ret}(\boldsymbol{\rho}) \circ (\lambda x.\ \bigcirc)^{(-)\setminus x:\tau_1}}$$

Todo, case for if x is not needed...

Where $\mathrm{ret}(\boldsymbol{\rho})$ takes the part of the slice that was required in synthesising τ_2, defined in **appendix**. This direct definition is quite complex; checking against *type-indexed slices* representing the synthesis of $\tau_1 \to \tau_2$ is *much easier*.

This definition does indeed find the minimum analysis slice:

[9] e' within the sliced context might now synthesise a *less precise* type.

Conjecture 6 (Correctness) *If* $\Gamma \vdash e \Rightarrow \tau$ *with sub-derivation* $\Gamma \vdash e' \Leftarrow \tau'$ *in checking context* $\mathcal{C}$ *then we also have that* $\Gamma; \mathcal{C} \vdash e' \Leftarrow \tau' \dashv \mathcal{p}$ *and:*

- p *is an analysis slice for* e'*.*
- *For any* $\mathcal{p}' = [c' \mid \gamma'] \sqsubseteq [\mathcal{C} \mid \Gamma]$ *such that* $\mathcal{p}'$ *is an analysis slice of* e' *then* $\mathcal{p} \sqsubseteq \mathcal{p}'$*.*

Extension to Type-Indexed Slices

As mentioned previously, using *type-indexed synthesis slices* calculated via *criterion 1* as input makes analysis slices much easier to calculate.

Additionally, the rules in this form end up being more closely tied to the Hazel typing rules and hence easier to formalise.

See appendix

3.1.6 Criterion 3: Contribution Slices

This criterion aims to highlight all regions of code which *contribute* to the given type (either synthesised or analysed). Where *contribute* means that if the sub-term changed it's type, then the overall term would[10] also, or would become ill-typed. Importantly, we also consider *contexts* for expression which are analysed, again considering any component terms which having their type changed would result in an error when trying to analyse against the type. For example in the following term:

$$(\lambda f : \texttt{Int} \to \texttt{?}.\ f(1))(\lambda x : \texttt{Int}.\ x)$$

The terms which *contribute* to the *second*[11] lambda term checking successfully against $\texttt{Int} \to \texttt{?}$ is everything *except* the x term, while context slice is just the annotation (and required structural constructs). Highlighting related to synthesis will be a darker shade:

$$(\lambda\, f : \texttt{Int} \to \texttt{?}.\ f(1)\,)\ \underline{(\lambda\, x : \texttt{Int}.\ x\,)}$$

Notice that, in any typing derivation the only sub-terms which *can* have their type changed without causing the rule to no longer apply are those which use type *consistency*.[12] The only such rule is *subsumption*. Further, the only time a term could be changed to *any* type and still remain valid is when it is checked for consistency with the dynamic type $\texttt{?}$.

Therefore, this criterion just *omits all dynamically annotated regions* of the program.[13] And, the contextual part of the slices are just *analysis slices*.

Definition 5 (Contribution Slices) *For* $\Gamma \vdash e \Rightarrow \tau$ *containing sub-derivation* $\Gamma' \vdash e' \Leftarrow \tau'$ *with checking context* $\mathcal{C}$*.*

A contribution slice of e' *is an analysis slice for* e' *in* $\mathcal{C}$ *paired with an expression typing slice* ς^{γ} *such that:*

- ς *is a slice of* e'*, that* $\varsigma \sqsubseteq e'$*.*

[10] No matter which type it was changed to specifically.
[11] Underlined below from now on.
[12] Note that this logic does *not* extend to globally inferred languages.
[13] But does not omit the dynamic annotations themselves.

- *Under restricted typing context γ, that ς checks against any τ_2' at least as precise as τ':*[14]

$$\forall \tau_2'.\ \tau' \sqsubseteq \tau_2' \implies \gamma \vdash e' \Leftarrow \tau_2'$$

A contribution slice for a sub-term e'' involved in sub-derivation $\Gamma'' \vdash e'' \Rightarrow \tau''$ where $e'' \neq e'$ is an expression typing slice $\varsigma''^{\gamma''}$ which also synthesises τ'' under γ'', that $\gamma'' \vdash \varsigma'' \Rightarrow \tau''$. Further, any sub-term of e'' which has a contribution slice of the above variety, is replaced inside ς by that corresponding expression typing slice.

This is most naturally calculated using *type-indexed slices* exactly replicating the Hazel typing rules:

Think more about type indexed context slices. Think how to reconstruct

$$\text{SConst}\ \frac{}{\Gamma \vdash c \dashv_{\Rightarrow} b \mid c} \qquad \text{SVar}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x \dashv_{\Rightarrow} \text{map}(x^{\{x:\tau\}}, \tau)}$$

$$\text{SFun}\ \frac{\begin{array}{cc} s_1 = (\square : \bigcirc) \circ \text{annot}(\tau_1) & \Gamma, x : \tau_1 \vdash e \dashv_{\Rightarrow} s_2 \\ s_2 = i_2 \mid \varsigma^{\gamma} & x \blacktriangleright_{\gamma} p \end{array}}{\Gamma \vdash \lambda x : \tau_1.\ e \dashv_{\Rightarrow} (\lambda \bigcirc.\ \square) \circ s_1 \to (\lambda p : \tau_1.\ \bigcirc) \circ s_2 \mid (\lambda p : \tau_1.\ \varsigma)^{\gamma \setminus x:\tau_1}}$$

TODO: fix SFun annotations for the argument slice, they need to remove unused requirements?

$$\text{SApp}\ \frac{\begin{array}{c} \Gamma \vdash e_1 \dashv_{\Rightarrow} s_1 \qquad s_1 \blacktriangleright_{\to} s_2 \to s \\ \Gamma \vdash e_2 \Leftarrow s_2(\bigcirc) \dashv s_2' \end{array}}{\Gamma \vdash e_1(e_2) \dashv_{\Rightarrow} \text{app}(s_2')}$$

$$\text{SEHole}\ \frac{}{\Gamma \vdash (\!|\,|\!)^u \dashv_{\Rightarrow} ? \mid (\!|\,|\!)^u} \qquad \text{SNEHole}\ \frac{\Gamma \vdash e \dashv_{\Rightarrow} p\{i \mid \varsigma^{\gamma}\}}{\Gamma \vdash (\!|e|\!)^u \dashv_{\Rightarrow} ? \mid (\!|\varsigma|\!)^{u\gamma}}$$

$$\text{SAsc}\ \frac{c = \bigcirc : \text{annot}(\tau) \qquad \Gamma \vdash e \Leftarrow \mathfrak{s} \dashv s}{\Gamma \vdash e : \tau \Rightarrow \text{app}(s)}$$

$$\text{AFun}\ \frac{\begin{array}{c} \mathfrak{s} \blacktriangleright_{\to} \mathfrak{s}_1 \to \mathfrak{s}_2 \\ \Gamma, x : [\![\mathfrak{s}_1]\!] \vdash e \Leftarrow \mathfrak{s}_2 \circ (\lambda x.\ \bigcirc) \dashv s_2 \end{array}}{\Gamma \vdash \lambda x.e \Leftarrow \mathfrak{s} \dashv \mathfrak{s}_1\{\lambda \square.\square\} \to s_2}$$

$$\text{ASubsume}\ \frac{\begin{array}{c} \Gamma \vdash e \dashv_{\Rightarrow} s \\ [\![s]\!] \sim [\![\mathfrak{s}]\!] \end{array}}{\Gamma \vdash e \Leftarrow \mathfrak{s} \dashv \bigsqcup \text{static}([\![\mathfrak{s}]\!], s)}$$

Figure 3.1: Contribution Slices

Where $\text{map}(\rho, \tau)$ creates a type-indexed slice of type τ with the slice context $\bigcirc$ and expression typing slice ρ tagged on all components of τ. Annot is as defined in criterion 1 extended to type-indexed slices. $x \blacktriangleright_{\gamma} = p$ matches p with x if $x \in \text{dom}(\gamma)$, otherwise returns $\square$. $\blacktriangleright_{\to}$ is extended to slices as follows:

$$p\{s_1 \to s_2 \mid \rho\} \blacktriangleright_{\to} p \circ s_1 \to p \circ s_2$$

[14] Essentially, sub-terms that check against ? also synthesise ?. Defined this way to include the case of unannotated lambdas (which do not synthesise).

$$p\{? \mid \rho\} \blacktriangleright_{\to} p\{? \mid \rho\} \to p\{? \mid \rho\}$$

static replaces a (sub)slice term if it is in the position of a ? on the input. Reconstructing the term is just taking the join down one level **(PROVE This)**.

3.1.7 Join Types

The Hazel core calculus is very primitive, only consisting of *base types*, *annotations*, and *functions*. Extensions to gradual types [12], and Hazel [9][15]: *if* expressions, *pattern matching*, *sum types* etc. all require[16] *join types*.

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.

Hence, type flow needs to be extended to allow these. Equally, slices themselves can be joined if they have common contexts (though there is a decision whether to include both branches of a join).

3.1.8 Type-Indexed Slicing Context

It seems natural to extend the typing context to a type-indexed slicing context. Then, when accessing typing assumptions via variable references, the source information about the derivation for this type is revealed. But, the problem is, there is no context propagated, these slices are slices of the orignal term; propagating all this would be a big pain.

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 expression typing 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.

Mention and compare with blame tracking

3.2.1 Indexing Slices by Types

3.2.2 Elaboration

All casts inserted come from type checking, so can be sliced

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

[16] Or are easiest formulated with.

3.2.3 Dynamics

Ground type casts will be added, but their addition is purely technical and we can treat their reasoning to just be extracting the relevant portion of the original non-ground type.

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 criteria 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 Type Slicing Implementation

Here I detail how the theories above were adapted to produce an implementation for Hazel.

3.3.1 Hazel Terms

Hazel represents its abstract syntax tree (AST) **(cite)** in a standard way by creating a mutually recursive algebraic data-type (**cite**).

Terms are classified into similar groups as described in the calculus (see section 2.1.2), though combining external and internal expressions, and adding *patterns*:

- **Expressions**: The primary encompassing term including: constructors[17], holes, operators, variables, let bindings, functions & closures, type functions, type aliases, pattern match expressions, casts, explicit fix[18] expression.
- **Types**: Unknown type[19], base types, function types, product types, record types, sum types, type variables, universal types, recursive types.
- **Patterns**: Destructuring constructs for bindings, used in functions, match statements, and let bindings, including: Holes, variables (to bind values to), wildcard, constructors, annotations (enforcing type requirements on bound variables).
- **Closure Environments**: a closure mapping variables to their assigned values in it's syntactic context.

For example fig. 3.2 shows a let binding expression whose binding is a tuple pattern (in blue) binding two variables `x`, `y` annotated with a type (in purple).

[17] The basic language constructs: integers, lists, labelled tuples etc. Also, user-defined constructors via sum types.

[18] Fixed point combinator for recursion.

[19] Either dynamic type, or a type hole.

```
let (x, y) : (Int, Bool) = (1, true) in x
```

Figure 3.2: Let binding a tuple with a type annotation.

Every term (and sub-term) is annotated with an *identifier* (ID, `Id.t`) in the AST which refers back to the syntax structure tree. In a sense, this is the equivalent of *code locations*, but Hazel is edited via a structure editor.

3.3.2 Type Slice Data-Type

I detail here *how* and *why* I implement type slices for Hazel.

Expression slices as ASTs

Actually the below might be inaccurate, check the structure of the typing system to see if we would ever need to re traverse an AST?

Either way, there is opportunity to speak about persistence with how type slices are combined and decomposed and how they overlap. etc.

Directly storing expression slices directly as ASTs is both *space and time inefficient*. The AST type is a persistent data structures [62, ch. 2], meaning that any update to the tree will retain both the old and updated tree. All nodes on the path to the updated sub-tree must be copied; for example, fig. 3.3 shows how a node G (in blue) can be sliced off from the tree while the old tree (in black) and the new tree (in red) both persist and share some unmodified nodes structurally. Expression slices do exactly this slicing operation extensively, so would require significant copying.

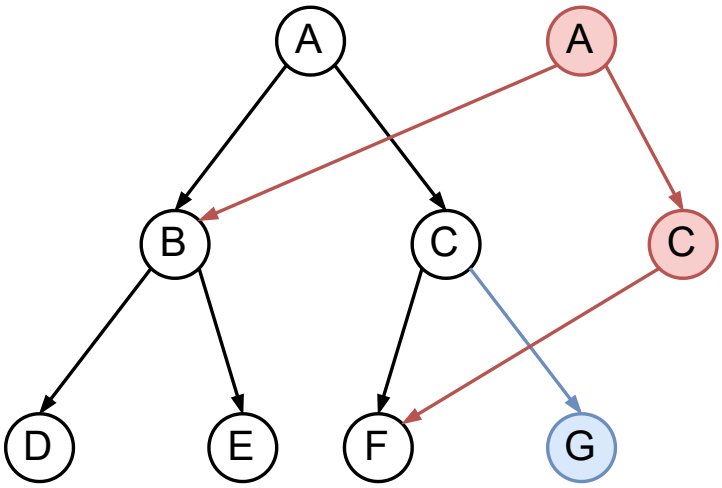


Figure 3.3: Persistent Tree

Maybe talk about a destructive version (always retain the original tree but from the perspective of various cursors

There are ways to partially avoid this, for example the *Zipper* data structure represents trees by from the perspective of a *cursor* node rather than the *root*. The part of the tree above the cursor is stored as an *upside-down*. This allows the cursor to be

We can derive an analogous *one-hole context* type for the AST by *differentiating* it's type [53, 42].

Zippers only useful if we need access to the parent node which cannot just be done by passing down info.

However, we still have to copy nodes when shifting the cursor; during type-checking all nodes will be visited by the cursor so we would get at least a *doubling* in space used. Additionally, converting a zipper back into a tree would take linear time[20] and still require copying the path to the cursor as before.[21]

Produce Tikz Diagram Here.

Figure 3.4: Tree Zipper

However, the biggest issue is that expressions slices have *multiple* gaps, so cannot be represented by a *one*-hole context. Further, the slices vary in their number of holes, so generalising one-hole contexts to two-hole contexts etc. is not an option as each would require it's own distinct type. There are extensions to zippers allowing multiple holes which would work, *multi-zippers* [5] for example, but has large constant overhead and would be very complicated to implement for such an extensive AST.

Unstructured Code Slices

With this in mind, given that the structure of expression slices does not actually matter for highlighting[22], I represent slices indirectly by these IDs in an *unstructured* list, referred to now as a *code slice* (`code_slice` type).

Additionally, this has the side effect of allowing more *granular* control over slices, as they now need not conform with the structure of expressions which is taken advantage of to reduce slice size, **(ref to section discussing this + evaluation)**.

Equally, the typing assumption slices are only required for formal type checking, however I maintain these is code slices to allow for different UI styling of slices originating from the use of the typing context (bold border).

Type-Indexed Slices

Cast slicing and contribution slices required *type-indexed* slices. I therefore tag type constructors with slices recursively, i.e.:

However, this did not model the structure of type slices particularly well. Slices are generally incrementally constructed. Synthesis slices build upon slices of sub-terms and analysis slices , demonstrated in fig. 3.6.

The original data-type works well for synthesis slices as the list data-type is persistent and sub-slices are never modified. But not for analysis slices, which require an id to be added to

[20] In the depth of the cursor.

[21] Still advantageous as it delays the copying only for when the slice is actually *used* in a way that requires this conversion. UI for slices would still work directly on the zipper structure without copying.

[22] Only matters for type checking slices, which always succeeds by design.

```
type typslice_typ_term =
  | Unknown
  | Arrow(slice_t, slice_t)  // Function type
  | ... // Type constructors
and typslice_term = (typslice_typ_term, code_slice)
and typslice_t = IdTagged.t(slice_term)
```

Figure 3.5: Initial Type Slice Data-Type

all sub-slices. Hence, analysis slices had *quadratic* space complexity in the depth of the type (**Get numbers in evaluation to show it being worse if possible**).

```
let f : Int -> Int = fun x -> x in f(0)
```

Synthesis slice for f(0) is just constructed incrementally from the slices for 0 and f.
Analysis slice for 0, f and all it's sub-slices all depend upon the annotation and binding etc.

Figure 3.6: Slicing is Incremental

Incremental Slices

Therefore, I explicitly represent slices incrementally, with two modes, for synthesis slice parts (termed *incremental slice tags*) and analysis slice parts (termed *global slice tags*).

I now tag each type constructor with incremental or global slices representing:

- Incremental Slice Tag: The slice of an expression is it's sub-slices adding it's incremental slice. But the sub-slices do not depend on the incremental slice.
- Global Slice Tag: All the sub-slices of this term depend on this slice.

So instead of tagging an id in an analysis slice to all subslices, I tag it only to the constructor as a *global slice*, and *lazily* tag it to sub-slices upon usage (during type destructuring).

We get the following type:

```
type typslice_typ_term =
  | Unknown
  | Arrow(slice_t, slice_t)  // Function type
  | ... // Type constructors
and typslice_term =
  | SliceGlobal(slice_t, code_slice)
  | SliceIncr(slice_t, code_slice)
and typslice_t = IdTagged.t(typslice_term)
```

Figure 3.7: Incremental Slice Data-Type

A key change when using this model is that when deconstructing types via function matching, list matching, etc. used throughout type checking and unboxing, the global slice must be pushed inside the resulting deconstructed types. This ensures that no part of the analysis slice context is lost **(compare to function matching in the type-indexed slice theory)**.

Usability & Efficiency

The type was further changed and many utility functions were added to enforce invariants and to ease development.

Empty Slices Many type constructors have no associated incremental slice part (**example**), especially during evaluation. Equally, when type slicing is turned off. A `TypSlice(typslice_typ_term)` constructor is added to represent this case.

Fully Empty Slices For the case when type slicing is turned off we also know that *all* sub-slices are empty, and we get an isomorphism between slices and the original *type*. For convenience in integrating with existing code, a fourth slice type `Typ(typ_term)` is added allowing a trivial and *efficient*[23] injection from types to typeslices by tagging with `Typ`. Note that this means the empty slices `TypSlice` no longer needs to include atomic types.[24]

Slice Tag Duplicates The previous formulation allowed structures a type constructor with multiple global slices. For example:

The possibility of any permutation of slices makes programming awkward when conceptually all we need is *at most one* global and incremental slice tag. I enforce this invariant by refining the type to the actual type used in the final implementation:[25] **TODO: Refactor code to use empty type-slices**

```
...
and typslice_empty_term = [
  | `Typ(typ_term)
  | `TypSlice(typslice_typ_term)
]
and typslice_incr_term = [
  | `Typ(typ_term)
  | `TypSlice(typslice_typ_term)
  | `SliceIncr(typslice_typ_term, code_slice)
]
and typslice_term = [
  | `Typ(typ_term)
  | `TypSlice(typslice_typ_term)
  | `SliceIncr(typslice_empty_term)
  | `SliceGlobal(typslice_incr_term, code_slice)
]
and typslice_t = IdTagged.t(typslice_term)
...
```

Polymorphic Variants [30, ch. 7.4], notated `[ | ... ]` are used here for convenience in incrementally writing functions, explained for an example *apply* function below. These are

[23] Conversion to the empty slice type would instead take linear time.
[24] These being equivalent to types, as no sub-slices exist.
[25] Modulo some insignificant refactorings.

variants which exhibit *row polymorphism* [63] [7, ch. 10.8] where these variants are related by a *structural subtyping* relation [69] where polymorphic variants of the same *structure*, with constructors of the same name and types, are subtypes.

We have that, `typslice_empty_term` is a subtype of `typslice_incr_term` which is a subtype of `typslice_term`. All other type constructors are either co-variant or contra-variant [65, ch. 2] with respect to the subtyping relation. For example, id tagging is covariant[26], so an `IdTagged.t(typslice_incr_term)` is a subtype of `IdTagged.t(typslice_incr_term) = typslice_t`.

Equally, a function of type `typslice_incr_term` → `typslice_incr_term` is a subtype of `typslice_empty_term` → `typslice_term`, being contravariant in it's argument and covariant in it's result. This function subtyping property significantly reduces work in defining functions on this type as seen below.

Utility Functions Functions on slices often do not concern the slices, but only the structure of it's underlying type, for example in unboxing[27] (**ref to sec**). In which case it is more convenient to just write a `typ_term` → α and `typslice_term` → α function directly on the possible empty slice types.

An `apply` function is provided to apply this onto the slice term. See how the bottom two branches can both be passed into the `apply` function even though they have different types (but are both subtypes of `typslice_term`).

```
let rec apply = (f_typ, f_slc, s) =>
  switch (s) {
  | `Typ(ty) => f_typ(ty)
  | `TypSlice(slc) => f_slc(slc)
  | `SliceIncr(s, _) => apply(f_typ, f_slc, s)
  | `SliceGlobal(s, _) => apply(f_typ, f_slc, s)
  }
```

Similarly, mapping function which map a `typ_term` → `typ_term` and similar functions onto slices are provided while maintaining the slice tags. Another particularly useful one is a mapping function that maps `typ_term` → `typslice_term` etc. and merges the slices around the input term and the output slice of the mapped function. Other utility functions include wrapping functions, unpacking functions, matching functions etc.

Type Slice Joins

Type joins are extensively used in the Hazel implementation for *branching statements*. The type of a branching statement is the *least specific* type which is still *at least as specific* as all the branches. This corresponds to the *lattice join* of the types of the branches with respect to the precision relation.

Previously in section 3.1.7, I stated how slices could be joined. To implement this, first any contextual code slices required to place the branches within a common context are wrapped onto the branch slices. Then the slices are joined, unioning the incremental code slices of branches.

[26] It is just a labelled pair type.

[27] e.g. checking if a slice is a *list* type.

For basic synthesis and analysis slices, I decide to take the code slices for each atomic type in the joined type from only *one* branch. This retains a complete explanation of *why* the joined type is synthesised/checked, but does *not* constitute a valid contribution slice.

This slicing is not as easy as just taking the slice of a single branch, as the most specific sub-parts of the joined type may come from differing branches. For example in fig. 3.8, the *then* branch has a type `Int` → (`?`, `Int`) and the *else* branch has type `Int` → (`Int`, `?`) meaning the joined type is `Int` → (`Int`, `Int`). We can omit one[28] of the redundant annotations on the argument but still must retain a slice of the 0 term in both branches to get the `(Int, Int)` slice.

if ? then fun x : Int -¿ (?, 0) else fun x : Int -¿ (0, ?)

Figure 3.8: Type Slice Join

The unstructured nature of code slices also allows the type constructors to select only *one* branch to take the slice from. The given figure would *not* highlight the function constructor in the *then* branch. However, this is not be a valid expression slice in the theoretic sense and could be confusing for the user **(discuss in evaluation, missing info)**.

Contribution Slices

To create a *contribution slice* (definition and correctness reasoning in section 3.1.6) we will need to combine analysis and synthesis slices on the same term that:

- Matches compound type constructor slice tags are combined.
- Atomic *dynamic* type parts of the *analysis slice* are retained in preference to the synthesis slice part.
- Other atomic type constructors have slices combined.

We get a type slice whose type is equivalent to the analysis slice, but includes relevant parts of the synthesis slice.

Use same example here as in theory

Figure 3.9: Contribution Slice

Weak Head Normalisation

Describe where this is used and how slices do this. This subsection could be elided.

3.3.3 Static Type Checking

The Hazel implementation is *bidirectionally typed*. During type checking, the typing *mode* in which to check the term is specified with `Mode.t` type: *synthesising*[29] (`Syn`) or *analysing* (`Ana(Typ.t)`).

[28] The figure, and my implementation, omits the left branches.

[29] Additionally split into synthesising functions and type functions, but this detail is elided here.

The type checker associates each term with a type information object `Info.t`, stored in a map by term id with efficient access. The *type information* stores the following info:

- The term itself and it's ancestors.
- **Mode**: Typing expectations enforced by the context.
- **Self**: Information derived independent from the *mode*, e.g. synthesised type, or type errors arising from inconsistent branch types, syntax errors.
- **Typing context and co-context**. A co-context
- Status: Is it an error, what type of error? For example, inconsistency between type expectations and synthesised/actual type.
- **Type**: The *actual* type of the expression after *accounting for errors*. Errors are placed in holes, so synthesise the dynamic type.
- Constraints: Patterns also store constraints to determine redundant branches and inexhaustive match statements, in the sense of [31, ch. 13][30]. Hazel-specific details in [10].

As suggested by my slicing theory, I augment the four bolded fields to refer to type slices, returning type slices from synthesis and analysing directly against type slices. This results in wide-changing code, but I only detail the general workings here.

Self

The *self* data-structure now returns *types slices* instead of *types*. Every expression construct which can be synthesised has a corresponding function in `Self.re` to construct the slice from it's sub-derivations (slices of synthesised types of sub-expressions). Hence, the synthesis slicing behaviour for each type of expression can be easily configured uniformly via editing these functions.

For example, the slice of a pattern matching statement, given slices of all it's branches (`tys`) is the join of it's branches wrapped in an incremental slice consisting of the ids of the match statement itself. Otherwise if the branches are inconsistent it returns a failure tagging the branch slices with ids of the branches:

This stage also included factoring out some expectation-independent code from the type checking function which had been missed by others.

Mode

The *mode* now analyses against *type slices* instead of *types*. Again, each construct which could deconstruct an analysing type has a corresponding function in `Mode.re` which outputs the mode(s) to check the inner expressions. The inner analysis slices are tagged with a *global*[31] slice tag describing *why* the slice was deconstructed. As mentioned before **(ref)**, deconstructing types retains the contextual (global) parts of the analysis slice.

For example, I can deconstruct a list slice with a list matching function `matched_list`. Using this, the mode to check a term inside a *list literal* is the matched inner list slice wrapped

[30] Only present in the *first* edition.

[31] Being part of the analysis slice, relevant to all sub-slices.

```
let of_match =
    (ids: list(Id.t), ctx: Ctx.t, tys: list(TypSlice.t),
    c_ids: list(Id.t)): t =>
  switch (
    TypSlice.join_all(
      ~empty=`Typ(Unknown(Internal)) |> TypSlice.fresh,
      ctx,
      tys,
    )
  ) {
  | None => NoJoin(Id, add_source(c_ids, tys))
  | Some(ty) => Just(ty |> TypSlice.(wrap_incr(slice_of_ids(ids))))
  };
```

Figure 3.10: Match Statement `Self.t`

```
let of_list = (ids: list(Id.t), ctx: Ctx.t, mode: t): t =>
  switch (mode) {
  | Syn
  | SynFun
  | SynTypFun => Syn
  | Ana(ty) =>
    Ana(TypSlice.(matched_list(ctx, ty)
    |> wrap_global(slice_of_ids(ids))))
  };
```

Figure 3.11: List Literal `Mode.t`

(globally) in the ids of list literal itself (`ids`) which enforced this matching. We use this mode to check each element in the list literal.

Typing (Co-)Context

The typing context and co-contexts are modified to use type slices, given that we now always have a slice to accompany a type in any situation.

Section 3.1.8 discusses one major consequence of allowing this. Slices for variables may now include the slice binding it's type.

This *deviates* from the theoretical notion of an expression slice: the structural context in which the variable is used is untracked when passing through the context. But, it is easy to implement using *unstructured* code slices and is a useful addition conceptually (**discuss in evaluation**).

Type

The *type* field is extended to a *type slice*. This has special behaviour for contribution slices and errors.

Basic Slices If there are no errors, just use the analysed type slice, or if in synthesis mode use the synthesis slice.

Contribution Slices: IMPL TODO When synthesis and analysis slices exist, they can be combined here. Section 3.3.2 defines and explains this combination.

Error Slices: IMPL TODO Type inconsistency error checking can be extended to type slices via using type slice joins. The analysed and synthesised type slices are inconsistent if and only if a join exists. We can show both conflicting slices in this situation to explain the error **(Implement)**. Additionally, syntax errors could have a slicing mechanism implemented here.[32]

3.3.4 Sum Types

Sum types are the hardest to work with, describe general design and difficulties in all the previous parts.

3.3.5 User Interface

Click on analysis or synthesis slices from context inspector

Show figures.

Implement UI from cursor inspector. Implement disabling of slicing.

3.4 Cast Slicing Implementation

To implement cast slicing, I replace casts between *types* by casts between *type slices*. The required type-indexed nature of type slices is already implemented, allowing these casts the be decomposed.

3.4.1 Elaboration

Cast insertion recursively traverses the unelaborated term, inserting casts to the term's statically determined type as stored in the `Info` data-structure and from the type as can be determined directly from the term.

For example, for list literals we can recursively elaborate the list's terms and join their static slices into `inner_type`. Then, intuitively we would know during dynamics from these elaborated sub-terms that the list has a list type of `List(inner_type)`:

Maybe a more diagrammatic option here

Figure 3.12: List Literal Elaboration

We can therefore construct the source type slices in these casts directly form the term during this traversal. **(TODO impl for atomic types like ints)**

Ensuring that all the type slice information from the `Info` map is retained and/or reconstructed during elaboration was a meticulous and error-prone process.

[32]Not within the scope of this project. Empty slices are given instead.

list example

Figure 3.13: Ground Matching List

3.4.2 Cast Transitions

Section 2.1.2 gave an intuitive overview of how casts are treated at runtime. Type-indexed slices allows cast slices to be decomposed in exactly the same way.

However, as Hazel only checks consistency between casts between *ground types* (fig. A.8), there are two rules where new[33] casts are *inserted* ITGround, ITExpand (fig. A.10). The new types are both created via a *ground matching* relation (fig. A.11) which takes the topmost compound constructor.

As we already store type slices incrementally, the part of the slice which corresponds *only* to the outer type constructor is just the outer slice tag.

3.4.3 Unboxing

When a final form (section 2.1.2) has a type, Hazel often needs to extract parts according to this type during evaluation. But due to casts and holes, this is not trivial [10].

For example, if a term is a final form of type list, then it could be either:

- A list literal.
- A list with casts wrapped around it.
- A list cons with indeterminate tail, e.g. `1::2::?`.

Additionally, when the input is not a list at all, it can return `DoesNotMatch`. Hence, allowing dynamic errors to be caught and also for use in pattern matching.

To allow for the varying outputs to unboxing depending on different patterns to match by, GADTs are used **(cite and explain)**.

Various helper functions for unpacking type slices into it's sub-slices to significantly simplify pattern matching.[34] A uniform function for this could be implemented with a GADT, in a similar way to unboxing, but is not required.

Hazel Unboxing Bug

While writing the search procedure I found an unboxing *bug* which would always *indeterminately match* a cons with indeterminate tail with *any* list literal pattern (of *any* length), even when it is known that it could never match. For example a list cons `1::2::?` represents lists with length ≥ 2, but even when matching a list literal of length 0 or 1 it would indeterminately match rather than explicitly *not* match.

Pattern matching checks if each pattern matches the scrutinee with the following behaviour, starting from the first branch:

- *Branch matches?* Execute the branch.
- *Branch does not match?* Try the next branch.

[33] As opposed to being derived from decomposition.

[34] These differ from matching functions, which also match the dynamic type to functions, lists etc.

- *Branch indeterminately matches?* Hazel cannot assume the branch doesn't match so cannot move on and must safely stop evaluation here classifying the entire match as an indeterminate term.

Figure 3.14 demonstrates a concrete example which would get stuck in Hazel, but does *not* need to.

See PR example

Figure 3.14: Pattern Matching Bug

I reported and fixed this bug and added additional tests to ensure the bug never reappears. A PR was merged into the dev branch **(pending)**.

3.4.4 User Interface

Mainly talk about the Model-view architecture and passing the cursor into the evaluator view to allow clicking on casts in evaluation result/stepper.

Difficulties in ensuring id tags for slices are not subtly aliased during elaboration and evaluation. This caused problems in selecting slices with UI. Look through commits to give example

3.5 EV_MODE Evaluation Abstraction

This section describes in detail the evaluator abstraction present in hazel, which allows ...

Very complex!!! The reader could skip this section, it is purely technical.

3.6 Indeterminate Evaluation

Dynamic type errors may only occur when an expression is given *specific* inputs. However, a dynamic error is accompanied by an evaluation trace, which is often a *useful debugging aid* [35]. When debugging static type errors, traces leading to a corresponding dynamic error are normally *unavailable*.[35] This section concerns the building blocks leading up to a search procedure that finds inputs which lead to dynamic errors *automatically*.

Seidel et al. [24] provides an algorithm for this in OCaml and provides evidence for the usefulness of traces via a *user study*. This algorithm *lazily* narrows hole terms non-deterministically to a *least specific* value based on it's expected type in the context it was used. For example, if a hole is used within the `(+)` operator, it is non-deterministically instantiated to an integer. **(give better example with lists)**

In Hazel, we already have a notion of hole terms and can already run program with static errors, with runtime type information being maintained by runtime casts. This section introduces *indeterminate evaluation* as a natural analogue to Seidel's idea: to *lazily* narrow holes during evaluation to *least specific* values by exploiting the runtime type information available within Hazel's runtime casts. My implementation extends Seidel's to consider more classes of

[35] As an ill-typed program will not run.

expressions[36] and differs mechanically in many ways due to language differences and fundamental design differences.[37] Notably, indeterminate evaluation is a *generic* evaluation method, not specifically relating only to searching for cast errors, which is covered in section 3.7.

This section covers the following, answering each question:

3.6.1 How should we resolve the non-determinism in instantiating holes *fairly*? How can the search order be abstracted? Unlike Seidel's approach, my implementation is *fair*: exhaustively considering all possibilities[38], and allows search methods that avoid non-termination[39].

3.6.2 Describes an algorithm for indeterminate evaluation. Is every possibility explored fairly?

?? How can *per-solution* state be maintained irrespective of evaluation order?

3.6.4 Discusses hole instantiation and substitution. What does lazy instantiation actually entail, when exactly should a hole be instantiated? Which hole[40] should be instantiated in order to continue evaluation to make progress? How should holes be substituted with their narrowed values; the same hole may exist in multiple locations within the expression?

3.6.5 How to use the evaluation abstraction (**??**) to perform just one evaluation step?

3.6.6 Finally, I consider Hazel-specific problems. Once we know which hole to instantiate, how can we get it's *expected type*? Hazel's lazy treatment of pushing casts into compound data types means not all such holes will be wrapped directly in casts. Additionally, the case of pattern matching is difficult, allowing holes to be *non-uniformly* cast to *differing types*. How can holes be instantiated in these situations?

As always, a UI is implemented in section 3.6.7.

3.6.1 Resolving Non-determinism

To model infinite non-determinism I decide to use a *monadic* high-level representation, and represent infinite solutions with sequences from Jane Street's `Base.Sequence`. I choose this due to it's *uniform*[41] and familiar workings for other developers in the codebase.

This sections describes an abstract DSL which abstracts the notion of non-determinism and the concrete search ordering. It's *module type* of combinators is `Nondeterminism.Search`. Section 3.7.4 discusses the actual implementations of this interface, giving three different searching procedures.

Alternatives

Section 2.1.4 considered multiple ways to represent non-determinism. The other options were:

[36] Notably, sum types.
[37] Differences, and Hazel-specific challenges are noted throughout. **(ENSURE THIS!)**
[38] For countable types.
[39] For example, if one particular instantiation leads to an infinite loop in code.
[40] There may be multiple.
[41] Many other types use the monadic return/bind interface. Lists and Options for example.

Direct implementation: This would not allow for easily abstracting the search order and would obfuscate the workings of the indeterminate evaluation and instantiation algorithms.

Effect handlers: Multiple continuation effect handlers were not supported by JSOO[42] **(cite)**,

Continuations: Directly writing continuations is difficult and generally unfamiliar to OCaml developers **(cite)**.

Optimised DSL : Introducing a formal DSL including optimisations, such as the proposed Tagless-Final DSL [32, 14], is very complex.

Monadic Non-determinism

To recap, in a monadic model of non-determinism we have three operations on monads `m('a)`:

- `return : 'a => m('a)`, injects a single solution into it's non-deterministic monadic representation.
- `bind : m('a) => ('a => m('b)) => m('b)`, corresponding to a conjunction of choice. Given possibilities from `m('a)`, each leading to more possibilities in `m('b)`: `bind` conjoins all the non-deterministic possibilities from `m('b)`.
- `fail : 'a => m('a)`, representing no solution, no choices are possible.
- `choice : m('a) => m('a) => m('a)`, a disjunction of choice: either solutions are possible. Represented infix by `<||>`.

These satisfy the before mentioned laws (section 2.1.4). And we can define a host of other useful constructs from these:

- `map : m('a) => ('a => 'b) => m('b)`. Represented infix by `>>|`. Where `m >>| f = m >>= (x =` Mapping represents performing a transformation on all possible solutions.
- `join : m(m('a)) => m('a)`, where `join(m) = m >>= (x => x)`. **(todo, give intuition for join)**.
- `once : m('a) => option('a)`, extracts any one solution, if one exists. This can be used to efficiently model *don't care* non-determinism, where if one possibility fails, then we know all others fail also.
- `run : m('a) => Sequence.t('a)` produces a lazy list of all solutions.
- `guard : bool => m(unit)`, represents success, defined by `code(false) = fail` and `code(true) = return(())`. When chaining binds, if any guard bind fails, then the whole computation fails.
- `ifte : m('a) => ('a => m('b)) => m('b)`. This corresponds to Mercury's interpretation of an if-then-else construct [3]. Where `ifte(c)(thn)(els)` executes the `els` only if the conditional `c` fails initially. Otherwise it binds the results of the in the `thn` branch. Importantly, if the conditional fails on backtracking, the `els` branch is *not* computed. Hence, it is put to good use for explaining failure of the conditional.

[42] An important dependency of Hazel.

Abstracting Search Order: Tree Model

Typical stream-based models of non-determinism [71] only admit the possibility of depth-first search (DFS). Streams provide no way of remembering choice points and backtracking before finishing a computation.

Stream model here, see [60]? Use arrows to indicate DFS.

Figure 3.15: Streams require Depth-First Search

Instead, a model of non-determinism using *trees* allows for nested choices. These trees can be traversed in various orders, such as breadth-first search (BFS)[60].

Tree of choices, use arrows to indicate breadth first search.

Figure 3.16: Tree of Choices with Breadth-First Search

To retain the possibility of failure, forests (lists of trees) can be used, with the empty list being failure.

Use same example as BFS above (pairs/choose function, i.e. instantiating tuples).

Figure 3.17: Forest Defined Using `wrap`

A combinator, `wrap`, allows for the programmer to control directly what constitutes a *fork* in the tree as suggested by Spivey [47]. Previous models would uniformly take each use of the choice operator as a fork, permitting less control. In essence, this allows for some notion of *cost*[43] for each solution to be directly encoded in the algorithm.

Implementations for DFS, BFS, and bounded DFS are given in section 3.7.4. In each case, the stream of solutions produced by `run` will return in the corresponding search order.

Fairness in Infinite Choice & Conjunction

Previously

Fair Choice: Consider the choice `m <||> n`, if `m` has infinite solutions. When using *append* to represent choice for sequences **cite**, then `m <||> n = m`[44]. No solutions of `n` will be encountered upon running through this choice. But append is a perfectly valid definition of choice as according to the specified laws.

To ensure each possible solution is returned in finite time, it is useful to introduce fair choice. A fair choice computation will always return *every* possible answer. For the append example above, any form of *interleaving* can represent fair choice. **Cite fair choice, e.g. kiselyov?**

Fair choice is denoted by `<|>`. Figure fig. 3.18, shows how two infinite choice trees for non-negative and negative integers respectively would be appended vs interleaved fairly.

[43] The depth of the node in the tree.

[44] `m` never ends to start appending `n`.

Figure 3.18: Unfair vs Fair Choice

1 -¿ (1, 1), (1, 2), ... 2 -¿ (2, 1), (2, 2), ...

Figure 3.19: Unfair vs Fair Conjunction

Fair Conjunction: Similarly, conjunction is not fair. For a computation `(m <||> n) >>= f`, if `m >>= f` is infinite, then only these might be returned. For example, binding sequences is typically defined via concatenating and mapping: `m >>= f = concat(m >>| f)`.

The use of unfair choice (`append`) leads to unfairness in bind. A fair conjunction would use fair choice instead.

Fair conjunction is denoted by `>>-`. Figure 3.19 demonstrates how *pairs* of natural numbers can be generated fairly using *fair conjunction*.

These fair combinators satisfies the original non-determinism and monad laws up to the ordering of results. **Spivey says `>>-` is not associative? Not sure I agree?**

Recursive Functions

As OCaml is strict, to allow for easier I define a shorthand lazy application function `apply(f, x)` defined by `return(x) >>= f`. This is equivalent to regular application up to laziness. Represented infix by `|>-`. This allows the writing of recursive functions directly resulting in infinite choices without OCaml's strictness leading to infinite recursion.

TODO: Do I need a fixed point operator? it seems hard to get this working for BFS!

3.6.2 A Non-Deterministic Evaluation Algorithm

In Hazel, for any term d, we have four situations, for which indeterminate evaluation performs the following rules:

- d is a final value.

  Return the value as a possible result.

- d is an indeterminate final term.

  Some of these terms contain holes.[45] Non-deterministically instantiate these holes lazily and *fairly* try indeterminately evaluating each resulting term.

- d is not final. Therefore a (deterministic) step is possible.

  Step to the new term d'. Indeterminately evaluate d'.

- There was an *evaluator exception*.

  Fail.

Instantiation can be represented by a *non-deterministic* function `instantiate`, discussed in detail in section 3.6.4:

`Instantiation.instantiate : Exp.t => m(Exp.t)`[46]

[45] Not all, indeterminate terms also arise from static or dynamic errors.

[46] Details & types relating to state and environment threaded through evaluation are omitted here for clarity. Therefore, these signatures differ from the code.

Classifying the term d, into values, indeterminate terms, and expressions with a possible step is done by a *deterministic* function `take_step`, discussed in section 3.6.5:

```
OneStepEvaluator.take_step : Exp.t => TryStep.t
```

Recursively performing the search is wrapped up as a new level in the search tree. This means that the tree never has an unbounded branching factor, as would occur by not wrapping evaluation steps if the evaluation went into an infinite loop.

To allow for multiple searching methods, the algorithm is placed within a *functor* **(cite)**, which takes a searching method, `S : Search`, with the previously described signature (section 3.6.1). Using `S`, the algorithm is defined in fig. 3.20.

```
module Make = (S: Search) => {
  module Instantiation = Instantiation.Make(S);
  open S;
  open S.Infix;

  let rec values = (d: DHExp.t) : S.t(DHExp.t) => {
    let step = OneStepEvaluator.take_step(d);
    switch (step) {
    | BoxedValue => return(d)
    | Indet =>
      d |>- wrap(Instantiation.instantiate
        >>- values);
    | Step(d') => wrap(d' |>- values);
    | exception (EvaluatorError.Exception(_)) => fail
    };
  };
};
```

Figure 3.20: Indeterminate Evaluation to Values

3.6.3 Threading Evaluation State

In order to track statistics for use in the evaluation, state must be threaded through indeterminate evaluation. State tracked includes, on a *per-solution* basis:[47]

- Trace length: number of steps performed to get to this solution.
- Number of instantiations performed to get to this solution.

So instead of threading only the solutions, `DHExp.t`, I also thread the state, `(DHExp.t, IndetEvaluatorState.t)`, updating according depending on the branch taken.

Additionally, Hazel evaluates expressions in an global environment (`env : Environment.t`) which must also be passed down, but does not change after performing a step.

[47] Hazel also uses the state to track other interesting features: *Probes* and *Tests*. These are complex, making use of mutability, but are outside the scope of this project. However, my code still maintains this state correctly **TODO**.

One example with a cast error. One example with one holes. An example with multiple holes. An example where the holes don't matter.

Figure 3.21: Where Does Indeterminateness Originate?

3.6.4 Hole Instantiation & Substitution

There are three key problems to solve in order to instantiate terms.

Choosing which Hole to Instantiate

An indeterminate term may contain *multiple* holes or even *no* holes. Which hole needs to be instantiated in order to *make progress*?

A term is concluded to be indeterminate if some part of it is indeterminate, as according to the rules in 2.1.2. If those corresponding sub-parts were instead a value, then the whole term would be a value. Recursively applying this logic, there will be a collection of terms where indeterminateness *originates*. These will either be holes or erroneous terms. If they are holes, then instantiating these may will either allow evaluation to continue or directly turn the overall term into a concrete value. Figure 3.21 demonstrates some examples of the possible situations.

Hazel's transition semantics specifies these rules, and careful use of the evaluation abstraction allows propagating the terms where indeterminateness originate (those which output the rule `Indet` in `transition`). To simplify, I consider and instantiate only the *first* hole where indeterminateness originates at a time. Should multiple need to be instantiated to progress, it will simply instantiate one hole, then fail to evaluate, then instantiate another, then try evaluate again, etc.

Synthesising Terms for Types

Suppose we know which hole to instantiate and to which type (section 3.6.6). How do we refine these holes *fairly* and lazily, to their *least specific* value that allows evaluation to continue?

Some types are *atomic* and must be instantiated to (possibly infinite set) of concrete types; again, we must be careful to ensure that the branching factor of the search tree remains finite:

Booleans: Simple binary choice: `return(true) <||> return(false)`.

Integers: Integers are countable, so can be enumerated, for example, by constructing the tree in fig. 3.22, which wraps each integer and it's negative in each level of the tree. Of course, OCaml integers are not arbitrary precision, so not every mathematical integer is represented here; in the future, Hazel might use arbitrary precision integers.

```
let rec ints_from = n => return(n) <||> wrap(n + 1 |>- ints_from)
let nats = ints_from(0)
let negs = ints_from(1) >>| n => -n
let ints = nats <|> wrap(negs)
```

SHOW TREE HERE

Figure 3.22: Enumerating Integers

Strings: A string is either *empty* or is a string with a first character from a finite set. We can recursively wrap all strings, prefixed by each character. See fig. 3.23:

```
let chars = // Every single letter string considered
let rec strings = () => return("")
  <||> wrap(chars >>= chr =>
            (() |>- strings) >>- str =>
            chr ++ str))
```

SHOW TREE HERE, also check impl

Figure 3.23: Enumerating Strings

Floats: Every float can be represented by a (or multiple) rational number(s). Therefore, generating every rational number will then performing float division, will generate every float. Again, this does not actually generate every rational due to OCaml's fixed-precision integers, and hence will not generate every float. See section 4.7.5 for further discussion.

Other types are *inductive*, these can be represented indirectly by lazily instantiating only their *outermost* constructor:

Lists: Any list can be directly represented as either the empty list, [ ] or a cons ? :: ?. We do not need to generate every list directly.

Sum Types: Enumerate each of the sum's constructors with their least specific value.

Functions: A function can be represented with it's least specific value by λ?. ?. However, to allow functions to be used it is useful to instantiate the pattern to the wildcard pattern '_'. The function may then be applied to any value. This therefore only generates *constant* functions, which is all that is required to find cast errors. But, extensions to program synthesis (section 5.1.6) would require more sophisticated function instantiation.

One might state that atomic types could be represented inductively. For example, every string of prefix `s` could be represented by `s ++ |\dyn|`. However, Hazel does not currently treat these inductively; there is no way to destructure such a string, without first fully evaluating to a concrete value. The evaluation (section 4.7.5), discusses the implications of this in detail.

Maintaining Correct Casts

Every hole instantiated must have originally been of the dynamic type. Therefore, that hole's context would have *expected* it to be of the dynamic type. In order to maintain correct casts, from *actual* type to *expected* type, we must cast the every instantiation back to the dynamic type.

Substituting Holes

Hole's can be evaluated, and may also be duplicated, or be added to closures, before they are required to be instantiated. How can we consistently substitute these holes for their instantiations?

Figure 3.24 shows how a hole $?_1$ can duplicated during evaluation and also bound within an closure.

Bbbb

Figure 3.24: Duplicated Holes

Hole substitution was described as part of the Hazel. Unexpectedly, the main Hazel branch did not yet implement it. Full implementation of *metavariables* and *delayed closures* is complex. Therefore, I instead use the existing term ids to represent metavariables, and ensure these ids are propagated correctly throughout statics, elaboration, and evaluation.[48]

By matching these ids, we can substitute terms for holes consistently throughout the entire expressions. This *included* substituting the holes eagerly inside closures, followed by evaluating the closure bindings to values, a required invariant.

3.6.5 One Step Evaluator

The step evaluator uses the evaluation abstraction to create an *evaluation context* from a term. **REF** Decomposing it into an object containing:

- An evaluation context, with a *mark* inserted in place of the sub-term which is evaluated according to the transition rules.
- The environment under which the sub-term is evaluated.
- The kind of rule that was used (e.g. substitution, addition, etc.).

Maintaining this information allows the indeterminate evaluation method to treat different steps differently. For example a search procedure might wish to extract only the steps which were substitutions, in order to produce a compressed execution trace.

3.6.6 Determining the Types for Holes

If we know which hole to instantiate, how do we know which type to instantiate it to?

For efficiency, my implementation both determines *which hole*, and it's *type information* during the same pass.

Directly from Casts

Most of the time, a hole is directly surrounded by a cast, whose type information can be used to perform an instantiation.

[48] A huge portion of the code-base required checks.

An example step leading to a specific eval obj.

Figure 3.25: `EvalObj.t` Example

Cast Laziness

However, it is *not* always the case that a hole which needs to be instantiated is surrounded by a cast. For efficiency reasons, Hazel treats casts over compound data-types lazily, for example, casts around a tuple will only by pushed inside upon usage of a component of the tuple.

Therefore, in order to actually instantiate these into values directly, casts must be pushed inside compound data *eagerly*. **TODO IMPL**

However, when detecting cast errors, it is possible to maintain lazy casts by extending Hazel's cast failure checking. Section 3.7.2 discussed this in more detail.

Pattern Matching

Does a hole even have only one possible type?

The introduction of (dynamic) pattern matching actually allows terms to be matched against *non-uniform* types if the scrutinee is dynamic. In section 3.6.6, if `term` is an integer 0 then it returns 0, but if it's the string "one" then it returns 1. Hence instantiating `code` to either an int or string might allow progress.

let term : ? = in case term — x::xs =¿ x — 0 =¿ 0 — "one" =¿ 1 end

Hazel implements this by placing casts on the *branches*, rather than the scrutinee. Therefore, we can collect each of these possible types from the casts, and wrap them around the scrutinee, continuing evaluation accordingly.

Extended Match Expression Instantiation

An interesting extension was attempted to improve code coverage of the instantiations **ref branch**. Given patterns are known, it is possible to instantiate holes according to not only the types, but also the *structure* of each pattern. This allows the instantiation to prioritise searching along each branch.

Hence, we could try instantiating the match scrutinee with least specific versions which match the patterns on each branch, e.g. `?::?` for `x::xs`. i.e.

To implement this, **(TODO)** the scrutinee needs to be matched against a pattern *and* instantiated at the same time. Crucially, instead of just giving up during indeterminate matches, the indeterminate part can be instantiated until it matches.

Figure 3.28: Combining Matching and Instantiation

However, this is not always enough to actually *allow* destructuring using that branch in a match statement. The possibility that *more specific* patterns could be present above the current

Show how indeterminate matches cause a list to try increasing lengths by repeatedly instantiating it's tail...

Figure 3.26: Instantiating a Scrutinee

```
case ?::?  | [] => [] | x::xs => xs
```

Figure 3.27: Branch-Directed Instantiation

NOT x::xs is [] NOT [?] is [] or ?::?::? Not 1 is the integers excepting 1 etc.

Figure 3.30: Not Instantiations

branch means the resulting instantiation might still be result in an indeterminate match. For example, the following would be an indeterminate match:

```
case ?::?  | [] => [] | x::y::[] => [] | x::xs => xs
```

Figure 3.29: More Specific Matches

To solve this, we can introduce least specific instantiations that do *not* match a pattern. These will generally not be representable by only *one* term. For example:

The use of inequalities here can become inefficient, and representing cases like *not 1* result in infinite search-spaces. Extending this to a full-blown SMT solver would be beneficial, see **??**.

3.6.7 User Interface

3.7 Search Procedure

Now that an framework for indeterminate evaluation has been specified, the following problems can be addressed:

?? How can indeterminate evaluation be abstracted into a generic search procedure?

?? What exactly are cast errors? How can they be detected? Which ones are actually *relevant*, causing evaluation to get stuck?[49]

?? How can different search methods be implemented: DFS, BFS, Iterative deepening DFS?

3.7.1 Abstract Indeterminate Evaluation

By making the search procedure take a higher-order `logic : Exp.t => TryStep.t => S.t('a)` function, which takes the an input expression d and it's class (value, indet, step possible), and returns nondeterministic results of type `'a`.

This allows defining search procedures returning other types, for example, the integer *size* of values. Or those concerning specific types of expressions, for example only those with cast errors.

Additionally, an 'expert' search abstract is given, which also allows the remaining search space to be defined. This allows, for example, customising the instantiation and evaluation methods.

[49] Some cast errors are known, e.g. from static type checking, but don't actually relate to the evaluation path.

let x = (1, "str") in let f = (fun x : (Int, Int) -¿ x) in f(x)

Figure 3.31: Inconsistent Lazy Casts Failure

3.7.2 Detecting Relevant Cast Errors

To search for cast errors, we must first define what one is. A reasonable definition is terms which contain *cast failures*, in Hazel these are casts between *inconsistent ground types*. However, this has some issues:

Multiple Cast Failures: Terms may have multiple cast failures, some of which discovered during static type checking and inserted via elaboration. The aim of this project was to allow give dynamic traces *explaining* a static type error. Clearly these failure must be ignored.

Additionally, the procedure has usage searching for errors in dynamic code. Situations which cause cast errors which don't actually stop evaluation should also be ignored. For example expressions which cannot be statically typed, but are safe, are within this class: `if true then "str" else 0`

Therefore, we consider only the casts which are *causing* a term to be indeterminate (therefore getting stuck), this is implemented similarly to choosing which hole to instantiate (section 3.6.4).

Cast Laziness Only casts between *ground* types are checked for consistency. Due to cast laziness (section 3.6.6), some are cast between inconsistent types, but *not* placed within a *cast failure*.

This can be fixed by performing eager cast transitions before checking for cast errors, or eagerly treating casts during indeterminate evaluation. However, it could be argued that this *should not* constitute a cast error, after all, the part of the compound type with the error is yet to actually be *used*.

3.7.3 Explaining Cast Errors

A static type error will place a term inside a cast error during elaboration. The id of this error can be tracked, and if it is the one at fault, associated with the static error. Additionally, the type slice enforcing the cast is tracked via cast slicing, and may be displayed as usual.

3.7.4 Searching Methods

Does interleaved DFS actually perform DFS? Reimplement both DFS and Interleaved DFS? Do I even need fair choice and conjunction?

I implement *four* different search methods, implementing the non-determinism signature specified in section 3.6.1.

Depth First Search

As mentioned previously **(REF)**, modelling lazy sequences by streams is a typical method, and implementing choice and conjunction via appending leads to depth-first search. In which case, `wrap` is just the identity function.

Show how streams & append represent depth first search

Figure 3.32: Depth First Search as Streams

Breadth First Search

Breadth first search needs to take specific account of the tree structure. Sequences of bags[50] can be used to represent solutions at each *level* of the tree [60]. Therefore, forcing each element of the stream gives the solutions at successive depths, i.e. iterating through these bags returns the solutions in a breadth first order.

Failure is the empty sequence as before, return injects a solution into a tree giving the solution in the first level.

Choice can be represented lazily merging of each level of the tree. The merged tree would produce all solutions of the first level of each choice at the same time, as the new first level. This is associative as required, when ignoring the ordering of each level, as is done with bags.

Merge trees, show that it maintains BFS.

Figure 3.33: Breadth First Choice

Wrapping a tree pushes every solution one level down in the tree, corresponding to prepending with the empty list. Essentially, this adds one cost to every solution.

For a bag `b` of elements `x1`, `x1` ..., `xn`. The tree which produces these solutions in the first level, $[b]$[51], is equivalent to `return(x1) <||> return(x2) <||> ... <||> return(xn)`.

A tree `b :: bs` is equivalent to `[b] <||> [] :: wrap(bs)`, and by the above, also equivalent to `return(x1) <||> ... <||> return(xn) <||> wrap(bs)`.

Therefore, as conjunction distributes over choice, and return is a left unit of conjunction then:

```
b :: bs >>= f = f(x1) <||> ... <||> f(xn) <||> wrap(bs >>= f)
```

Additionally, another law for non-determinism `fail >>= f = fail`, completes a recursive definition for bind.

As mapping and choice are associative, we get that conjunction is also. Additionally, it satisfies all the remaining monad and non-determinism laws.

However, as `f` is mapped to the entire bag `b`, it is difficult to maintain laziness in OCaml. We need partial applications for f... **TODO - explain how to do this in OCaml?!?!**

Iterative Deepening Depth First Search

While breadth first search is fair, avoiding non-termination (for finite branching factor), it has *exponential* space complexity in the depth of the level being explored [23]. When binding a function `f`, it gets lazily and *partially* instantiated at node of each level, tracking these partial instantiations is what causes the exponential complexity.

[50] Lists, but where ordering doesn't matter.

[51] Using list notation also for sequences

Figure 3.34: Monadic BFS Requires Exponential Space

In comparison, depth first search only requires space linear in the depth explored, that is, the number of choices encountered before reaching a certain depth.

The use of iterative deepening, successive depth-bounded depth first searches, retains low space complexity of DFS while also avoiding non-termination (due to the depth bound). However, upper parts of the search tree will be repeatedly explored, but this does not reduce the already exponential time complexity of the search for branching factor greater than 1. But, the deterministic evaluation part does have a branching factor of 1, so it may be the case that the recalculation has a significant impact**(Ref evaluation)**.

To represent iterative deepening [**SearchAlgebra**], we can use functions `int => list(('a, int))`, calculating every[52] solution found within an integer depth bound `d`, alongside it's remaining depth budget `r`. Tracking the remaining depth budget simplifies `bind` and allows only solutions on the fringe (zero budget) to be output upon each iteration, so the same solution will not appear twice.

Then, `return` represents a single solution with unused depth budget, `return(x) = (d => [(x, d)])`. `fail` gives the empty list `fail = d => []`. Choice appends all solutions at any given depth bound `m <|> n = d => m(d) @ n(d)`. Binding, `m >>= f` takes solutions in `m` who have depth budget to use, and applies `f` and calculates all solutions from `f` using the remaining budget, concatenating these results to form a single list.

Then `wrap(m)` (the forest `m` wrapped up as a single tree) has solutions at depth incremented by 1, and none at depth 0: `wrap(m)(0) = []` and `wrap(m)(bound) = p(bound - 1)`.

I implement this as a functor, with a custom bound incrementing function `inc : bound => bound` and initial bound `init : bound`. Then `run` will search to depth `init`, produce all solutions, then search to depth `inc(init)` and produce all solutions with remaining depth budget `inc(init) - init`, other solution must have already been produced last iteration.

Interleaved Streams

TODO: split code into a new option, remove fair operators, move fair conjunction and chocie section to here.

The use of streams, but with fair choice and conjunction via interleaved ordering ensures termination for every solution. This has the advantage of working even for trees with infinite branching factor. However, interleaving has a linear space complexity, leading to the undesirable exponential space complexity as with breadth-first search.

`wrap` for this is the identity, same as DFS.

3.7.5 User Interface

3.8 Repository Overview

3.8.1 Branches

Up to date with dev branch up to **DATE**

3.8.2 Hazel Architecture

[52] Again, relying on finite branching factor.

```
((m >>= f) >>= g)(bound)
= m(bound)
  |> map((x, rem_bound) => f(x, rem_bound))
  |> concat
  |> map((y, rem_bound2) => g(y, rem_bound2))
  |> concat
```

Vs.

```
(m >>= (x => f(x) >>= g))(bound)
= m(bound)
  |> map((x, rem_bound) => rem_bound |> (f(x) >>= g))
  |> concat
= m(bound)
  |> map((x, rem_bound) => rem_bound |> (f(x)
  |> map((y, rem_bound2) => g(y, rem_bound2))
  |> concat))
  |> concat
= m(bound)
  |> map((x, rem_bound) => f(x, rem_bound)
  |> map((y, rem_bound2) => g(y, rem_bound2))
  |> concat)
  |> concat
= m(bound)
  |> map((x, rem_bound) => f(x, rem_bound))
  |> concat
  |> map((y, rem_bound2) => g(y, rem_bound2))
  |> concat
```

Figure 3.35: Associativity of Iterative DFS `bind`

Chapter 4

Evaluation

Add citations to evidence statements on usefulness, i.e. smaller slices are easier to comprehend

4.1 Goals

This project devised and implemented three features: *type slicing, cast slicing,* and a *static type error witness search procedure.* Each of which had a clear intention for it's use:

Type Slicing: Expected to give a greater static context to expressions. Explaining why an expression was given a specific type.

Cast Slicing: Expected to provide static context to runtime type casts and propagate this throughout evaluation. Explaining where a cast originated and why it was inserted.

Search Procedure: Finds dynamic type errors (cast errors) automatically, linked back to source code by their execution trace and *cast slice.* Therefore, a static type error can be associated automatically with a concrete dynamic type error *witness* to better explain.

4.2 Methodology

I evaluate the three features and their various implementations (where applicable) along *four* axes. Quantitative measures were evaluated over a corpus of ill-typed and well-typed Hazel programs (**??**):

Quantitative Analysis

Performance: *Are the features performant enough for use in interactive debugging? Which implementations perform best?*

Measures: The time and space usage was expected to be within the typical *debugging* build times and space usage for small programs.

The time limit chosen was *one minute* and space limit was **INSERT LIMIT HERE, 1GB?**.

Effectiveness: *Do the features effectively solve the problems? Are the results easily interpretable by a user?*

Measures: the *coverage* of the search procedure, what proportion of programs admit a witness. The *size* of witnesses, evaluation traces, type slices, and cast slices.

The intention is that a smaller size implies that there is less information for a user to parse, and hence easier to interpret.[1]

The search procedure was expected to provide a *reasonable* coverage, chosen at 70%.

Qualitative Analysis

Critical: *What classes of programs are missed by the search procedure? What are the implications of the quantitative results? What improvements were, or could be made in response to this?*

This section provides *critical* arguments on *usefulness* or *effectiveness*, which are *evidenced* by quantitative data.

Differing implementations and *subsequent* improvements are compared. Additionally, further improvements are proposed.

Holistic: *Do the features work well together to provide a helpful debugging experience? Is the user interface intuitive?*

Various example are given, demonstrating how all three features could be used to debug a type error. Improvements to the UI are discussed.[2]

4.3 Hypotheses

Various hypotheses for properties of the results are expected. The evidence and implications of these are discussed in the *critical evaluation*.

Search method space requirements: The space requirements for DFS and Iterative DFS are expected to be lower than that of BFS and interleaved DFS. Space limit is expected to be a more *pressing* issue than time limit.

Contribution slices are large: Contribution slices highlight all static regions of a program. This likely shows *too much* information to the user at once. The other slicing methods (synthesis, analysis, ad-hoc) are expected to produce significantly *smaller* slices.

The Small Scope Hypothesis: This hypothesis states that a high proportion of errors can be found by generating only *small* inputs. Evidence that this hypothesis holds has been provided for Java data-structures [50] and answer-set programs [33]. Does it also hold for finding dynamic type errors from small *hole instantiations*?

Smaller instantiations correlate with smaller traces: If this and the small scope hypothesis hold, then most errors could be found with *small execution traces*.

[1] Not necessarily *always* true, but a reasonable assumption to an extent.

[2] Improved UI being a low-priority extension.

	Number	Prog. Size		Trace Length	
		Avg.	Std. dev.	Avg.	Std. dev.
Well-Typed	-	-	-	-	-
Ill-Typed	-	-	-	-	-
(Both)	-	-	-	-	-

Figure 4.1: Hazel Program Corpus

4.4 Program Corpus Collection

A well-typed AND ill-typed program corpus. Note that ill-typed here means one with a type error, even if it isn't statically caught (due to missing annotations).

4.4.1 Methodology

Supervisor help, transpiler, data from Seidel...

4.4.2 Alternatives

OCaml → Hazel transpiler

4.4.3 Statistics

The program corpus contains **X** programs, with averages and standard deviations in size and trace size[3] shown in fig. 4.1.

4.5 Performance Analysis

Note: this section is the lowest priority results...

4.5.1 Slicing

The entire corpus of ill-typed and well-typed programs is used to analyse slicing methods.

To evaluate **type slicing** performance, the corpus is *type checked* using three slicing implementations and also no slicing. Time and space usage for each program is *normalised* by it's size, as larger programs take longer and use more space to type check by a linear factor in their size.[4]

To evaluate **cast slicing** performance, each program is *evaluated* using the same implementations. Time is *normalised* by the trace-length, as longer traces take linearly longer time. Space is *normalised* by program size.

In both cases, space usage is measured by the *peak* live memory usage over the period. Calculated via `GC.stat` and `GC.create_alarm`.

Figure 4.2 gives the results, where **(n)** means the results were normalised. `dev` is the Hazel main branch with no slicing. `witnesses-search` (`ad-hoc`) calculates smaller reduced

[3] When using normal, deterministic, evaluation.

[4] Not exact in the worst case, but true in practice **(VERIFY)**.

(could present this as a bar chart)

		Branch			
	unit	`dev`	`ad-hoc`	`full`	`contribution`
Time Avg.	ms	-	-	-	-
Std. dev.		-	-	-	-
Time (n) Avg.	ms/prog. size	-	-	-	-
Std. dev.		-	-	-	-
Space Avg.	words	-	-	-	-
Std. dev.		-	-	-	-
Space (n) Avg.	words/prog. size	-	-	-	-
Std. dev.		-	-	-	-

(a) Type Checking

Time Avg.	ms	-	-	-	-
Std. dev.		-	-	-	-
Time (n) Avg.	ms/trace size	-	-	-	-
Std. dev.		-	-	-	-
Space Avg.	words	-	-	-	-
Std. dev.		-	-	-	-
Space (n) Avg.	words/prog. size	-	-	-	-
Std. dev.		-	-	-	-

(b) Execution

Figure 4.2: Normalised Performance Statistics for Slicing

ad-hoc slices (see section 4.7.1). `full-analysis-slices` (`full`) calculates the full synthesis & analysis slices as proposed by the theory in section 3.1, where analysis slices take priority if both are present. `contribution-slices` (`contribution`) implements full contribution slices in section 3.1.6.

Cast slicing has *negligible* impact on performance, whereas type slicing has a *relatively large* impact. However, in absolute terms still small, and far below the 60s and **(MEM TARGET)** targets.

4.5.2 Search Procedure

Only the ill-typed program corpus is used for evaluating the search procedure. After all, any well-typed program cannot have a dynamic type error.

As the search procedure may be non-terminating, the performance goals are directly input, by setting a 60s time limit and **MEM LIMIT** memory limit. The succeeding *effectiveness* analysis considers the search procedures results under these constraints.

Only **X**% of inputs failed due to insufficient time or memory, see fig. 4.4.

4.6 Effectiveness Analysis

4.6.1 Slicing

Type slice sizes were calculated over the entire corpus, where the size is the number of constructs highlighted. For `full` and `contribution` slices this corresponds directly with the size of the

(could present this as a bar chart)

		Branch			
	unit	`dev`	`ad-hoc`	`full`	`contribution`
Type Slice Avg.	size	-	-	-	-
Std. dev.		-	-	-	-
Type Slice (n) Avg.	%	-	-	-	-
Std. dev.		-	-	-	-
Cast Slice Avg.	size	-	-	-	-
Std. dev.		-	-	-	-

Figure 4.3: (Normalised) Slice Sizes

— Show **pie chart** for what proportion of programs had a witness found. What proportion of them failed due to time limit vs memory limit. Add a label for total programs failing due to limits. What proportion failed due to no witness under my semantics/no progress possible. Ideally a proportion for programs with NO witness in actuality would be useful, but this would require knowing from the dataset (manual analysis...).
One chart for each search procedure.

Figure 4.4: Search Procedure Coverage

Show a chart of each procedures coverage over increasing time limits.

sliced expression. This statistic is *normalised* by the size of the expression being sliced, to give a *proportion* of the term which is highlighted. As expected, the contribution slices highlight a much larger proportion of the term.[5]

Cast slices are measured the same way, but it does not make sense to normalise them against anything. Cast slices are generally much smaller than type slices, as casts to complex types are split up into multiple casts between smaller types[6] (hence, smaller slices).

4.6.2 Search Procedure

Witness Coverage

The majority of programs had a witness discovered, meeting the 70% target for all search methods. Other programs failed due to insufficient time or insufficient space. Additionally, some programs *provably* have no witness under my semantics, that is, the search procedure terminated without finding an error. Finally, some programs actually have no witness, but are classified as running out of time or space due to trying infinitely many instantiations.[7]

The pie charts in fig. 4.4 give the proportions of all these classes of programs for each search procedure. As expected, BFS and interleaved DFS have a higher proportion failing due to insufficient memory requirements and DFS has a higher proportion failing due to insufficient time. Iterative deepening fails to prove any programs have *no* witness, as it repeatedly tries to search to a lower depth, even if the tree is finite.

In fact, most witnesses are found in much *less* than 60s, with the coverage reaching an asymptote at 60s. This suggests that the remaining **X**% of programs that fail in this case

[5]The only reason the number is this low is due to the presence of mostly-dynamic code in the corpus.
[6]For example, ground types.
[7]Checking for these requires manual inspection of the corpus. Hence lack of numbers for this.

—- Pie chart of code coverage over the entire corpus

(a) All Searches

— PIe chart for only the failed searches

(b) Failed Searches

Figure 4.5: Code Coverage

	DFS	BFS	Iterative	Interleaved
Witness Size Avg.	-	-	-	-
Std. dev.	-	-	-	-
Trace size Avg.	-	-	-	-
Std. dev.	-	-	-	-

Figure 4.6: Witness & Trace Sizes

Plot graph of the size and trace pairs with a (linear?) correlation.

Figure 4.7: Witness size-Trace size Correlation

are *significantly* more difficult to find a witness for (or have no witness). Section 4.7.4 splits this class of programs, considering what causes these programs to fail to find a witness, and suggests improvements to increase this coverage.

Code Coverage

Exhaustive instantiation does direct the search procedure to explore branches in code which have not yet been explored by other instantiations. Yet, the code coverage was still high overall (**X**%). However, for failed searches, the code coverage was significantly worse (**Y**%); this suggests that the error may be persistently missed in the uncovered code (see Section 4.7.4).

Witness & Trace Size

As predicted by the small-scope hypothesis, most programs admitted *small* witnesses. Further, there was a positive linear correlation (Pearson correlation coefficient [**PearsonCorrelation**]: r = **X**, fig. 4.7), suggesting that the smaller witnesses do indeed lead to smaller traces, which are easier for users to comprehend [**SmallerTraces**]. Hence, the generated witnesses are generally *better quality* than other larger witnesses, for example those that random generation may provide.

4.7 Critical Analysis

This section discusses the implications of the previous results and delves deeper into the reasoning behind them. As a response to this analysis, many improvements have been devised, some of which have been implemented.

4.7.1 Ad-Hoc Slicing

Contribution slicing produces large slices (fig. 4.3, of which, much of the information is not particularly *useful*, with much of it explaining why the use of *subsumption* in subterms was valid, see fig. 4.8 and 4.9.

On the other hand, an *analysis slice* or *synthesis slice* provides the key information explaining why that expression was checked against some type, or synthesised a type. The drawback being that changing the type of a *non-highlighted* sub-term might cause type checking to fail, causing a new type error. However, if the term is *already* highlighted as a static type error, then changing such a sub-term could *never* fix the type error.[8] So these slices are a better choice for use in *debugging* type errors.

Additionally, some of the constructs highlighted are purely structural, not changing the type of the expression. For example, bindings when the bound variable is dynamic or unused in the particular expression considered. These cannot contribute to the type of the expression in any way, so can be omitted. I call these *ad-hoc* slices, because they no longer produce valid expression slices **(ref)** as per the theoretical definition, this requires *unstructured slices* (**??**). *I might also remove things like functions from the slices too... not as easily justifiable*

Compare the same program with a large contribution slice, vs a smaller analysis slice, vs an even smaller ad-hoc slice. Synthesis slice... Have bindings etc.

Figure 4.8: A Verbose Contribution Slice: Analysis

One-Level-Deep Contribution Slices

To understand why a term *successfully* type checked, you may need to look at *both* the synthesis and analysis slice for a term. But, if the analysing type has a dynamic sub-part, then the corresponding part of the synthesis slice is not actually relevant for type checking to succeed, so can be trimmed.[9]

Therefore, the UI was implemented to, by default, display this combined analysis and trimmed synthesis slice **TODO**. However, casts use *only* the analysis slices, since this is what actually *enforces* the type, understanding why it *successfully* type checks has less use for this.[10]

[8] As a type error is caused by inconsistency between types retrieved from the direct analytic and synthetic type contexts.

[9] This is the same trim as performed in contribution slices **(REF)**, but not doing so *recursively*.

[10] Instead, we just want to why it has it's type.

Compare the same program with a large contribution slice, vs a smaller analysis slice, vs an even smaller ad-hoc slice. Synthesis slice

Figure 4.9: A Verbose Contribution Slice: Synthesis

One level deep contribution slices help give a simpler explanation for expression type checks successfully vs using analysis and synthesis seperately.

Figure 4.10: One-Level-Deep Contribution Slices

Figure to show how consistent parts of the syn/ana slices can be omitted, to draw attention to the erroneous parts.

Figure 4.11: Error Slice

Errors

When looking specifically at a *type error*, there will be an inconsistency between the term's analysis and synthesis slices. Or, for synthesising branch statements, may also occur from inconsistent branches (whom are *all* synthesising).

But, some portions of the type might actually be consistent, and hence not causing to the error. A form of type joining which extracts only the inconsistent slice parts was therefore implemented **(TODO)**, and provides the default UI when clicking on a type error.

4.7.2 Structure Editing

Hazel uses a structure editor with an update calculus [**HazelStructureCalculus**]. Statics are recalculated upon edits and even *cursor movements*. Figure 4.2 shows that slicing makes type checking significantly slower. Slicing is not so suitable for such a rapid use case. Therefore, a way to turn off slicing would be beneficial[11], only performing slicing upon requesting slices or performing elaboration.

Another improvement would be to only partially recalculate types upon updates. The Hazel structure editor allows efficient[12] updating of the *syntax* tree via the use of a zipper [**HuetZipper**, 53]. It might be possible to extend this zipper idea into the abstract syntax tree and it's statics (typing information), allowing local changes (and type changes) to propagate in the AST zipper. However, some local changes can propagate type changes *non-locally* (e.g. inserting a new binding) which would require extensive recalculation of the typed AST. But, these non-local updates would be relatively rare, especially given most edit actions leave the tree in an incomplete state [**IncompleteEditStates**], whose expressions are dynamically typed (quickly calculable).

This is very complex and would require an entire rewrite of the Hazel statics.

4.7.3 Static-Dynamic Error Correspondence

Section 3.7.3 and **??** show how a static error that elaborates a cast failure, can be associated with a dynamic error whose cast error is dependent on this failed cast. This works well for most static type errors, which occur from inconsistent analysis and synthesis types.

However, static errors caused by synthesised branches being inconsistent, do not actually directly cause cast errors. Only representing that the branches cannot be placed within any cast (except to the dynamic type). Such errors can only be detected if both branches are used within a static context during the same evaluation path. The search procedure will find such cases if they exist, but cannot associate it back to the static error.

[11] My implementation architecture allows easy implementation of this, just use `` `Typ `` for everything.
[12] Constant time.

4.7.4 Categorising Programs Lacking Type Error Witnesses

Non-Termination & Unfairness

My original search method (DFS) has significant issues with non-termination (fig. 4.4). Any time evaluation goes into an infinite loop, no more solutions can be explored. Iterative deepening, BFS, and interleaved DFS were implemented in response to this.

Even so, substituting holes inside closures and enforcing the invariant that the bindings must be values (requiring potentially non-terminating evaluation), can still cause this. As Hazel is pure, it is safe to ignore this invariant, avoiding this class of non-termination, at the expense of efficiency in the general case when variables are used multiple times.[13]

Repeated Instantiations

Sometimes, a hole can get repeatedly refined to increasingly specific types without every actually causing a cast error, see fig. 4.12. This situation would be caught statically, but cannot actually directly lead to a cast error.[14]

See the 'safe' list example from Seidel.

Figure 4.12: Instantiations of Increasingly Specific Types Loop

Dead Code

Dead code may contain or be involved in static type errors, but is unreachable dynamically, such static errors have no dynamic witness. Dead code detection [**DeadCodeDetection**] could alert the user of this.

Dynamically Safe Code

Some code with static errors is inherently dynamically safe, the typical example being `if true then 0 els` These should have *no* dynamic witness.

Early versions of the search procedure would incorrectly detect cast failures that were inserted statically during elaboration, but weren't actually causing dynamic, leading to dynamically safe code being incorrectly asserted as an error. Section 3.7.2 detailed how cast errors were detected *correctly*.

Needle in a Haystack

Some programs have cast errors that are intricately dependent on specific instantiations. These casts only manifest within a branch which requires very specific inputs. When multiple holes are involved, this is an even bigger issue, as the search space increases exponentially in the number of holes.

The fact the a large proportion of programs which failed due to insufficient time or space had lower code coverage (fig. 4.5), suggests that this might be the reason behind a large proportion of the failures.

[13] Corresponding with a *call-by-name* evaluation strategy.

[14] So is not classified as having a witness under these semantics.

Pair of single inputs leading to cast error

Figure 4.13: Branch Requiring Specific Instantiations

Proportions found using iterative deepening under the time limit. Also, code coverage comparison.

Figure 4.14: Pattern-directed Hole Instantiation Using Iterative Deepening

4.7.5 Improving Code Coverage

Of the discussed classes of programs lacking a type error witness (section 4.7.4), only one actually had concrete witnesses that were not found within the time limit. This was due to the solution being a small number of possible instantiations within a large search space. I pointed out that that the search procedure had lower code coverage in these situations, and the possible cast error is likely to be in portions of the code that were *missed*.

These errors require rather specific, often interdependent, inputs to be missed by the current search procedure. Therefore, it is likely that programmers are *also* more likely to not notice errors (in dynamic code), or not understand the errors (in static code).[15]

Intelligently directing hole instantiations to better explore the code would help. Purely structural pattern-directed instantiation was discussed in section 3.6.6, but was not found to significantly increase code coverage nor reduce failed searches. This might be because the examples involving very specific inputs in the corpus happen to be mostly to do with *if* statements, rather than *match* statements.

In order to extend this structural pattern-directed instantiation to worth also with base types (integers), requires more efficient ways to representing constraints. To extend this to include floats, inequalities, list lengths, etc. requires full-blown symbolic execution [17].

Symbolic execution and program test generation is a well-researched area with numerous dedicated constraint solvers [**CITE MANY HERE**] and translations in to theories for SMT solvers [**CITE SMTS HERE**]. However, even without this, the search procedure still retains a high coverage, finding most errors.

4.8 Holistic Evaluation

This section considers a number of examples of ill-typed Hazel programs, *holistically* and *qualitatively* evaluating how a user might use the three features and the existing bidirectional type error error localisation [**HazelErrors**] to debug the errors.

4.8.1 Type Error Localisation in Hazel

Ref to each example here...

Hazel has three types of errors which can be addressed by this project:[16] *inconsistent expectations* (where a term's analytic and synthetic types are inconsistent), *inconsistent branches* (where branches are synthetic and inconsistent, this also applies to values in a list literal), and *inexhaustive match* warnings.

[15] After all, spotting the error might require understanding interdependent inputs.

[16] Others being syntax errors, unbound names, duplicate labels, deferrals, or redundant pattern matches.

The witness search procedure can easily associate *inconsistent expectation* errors with a witness: a dynamic cast error and trace. Inconsistent branches will be detected by the procedure, but are not automatically associated to the static error. Explaining inexhaustive match statements was not a core aim of this project, but the indeterminate evaluation can be adapted for this anyway (see the `inexhaustive-matches` branch **TODO**). Static generation of counter-examples to match statement exhaustivity is already a solved problem **(CITE)**, and is under development for Hazel.[17]

Bidirectional type error localisation is generally very good [**BidirectionalTyping**]. Type slicing can provide an explanation of where type expectations in the error were sourced. However, there are still cases where the error highlighted is not the actual location of the error.[18] These often arise due to *misunderstanding* of the programmer about the actual types of the code. Fundamentally, there is *no* way for a type system to always correctly localise such errors. Use of the Hazel context inspector and type slices can help clear these misunderstandings.

Additionally, some errors might require changes to *multiple* parts of the code [20]. Unlike most languages, Hazel *does* allow *multiple* errors to be detected *completely* [**HazelErrors**]. However, these errors are typically *not associated* together.[19] Type slicing can give a more complete view of regions that might contain the multiple parts pertaining to the *actual* error.

Finally, there may be errors in *dynamic regions* of the code. These are not found statically. The search procedure tests dynamic code for such type errors automatically. Indeterminate evaluation could also, in future, be used to perform property testing of code, in the sense of SmallCheck [44]. Adding annotations to find type errors is time-consuming, often a large number of, concrete, annotations are required to make the code static enough to detect errors. For example, fig. 4.19 shows a partially annotated program, which *still* doesn't provide enough type information to detect the error statically.

4.8.2 Examples

Merge these examples into the paragraphs above?? Want an example for each. All the examples can apply to dynamic errors.

Error in the @ operator. Due to annotation, search procedure can help give reasoning. let map : (Int -¿ Int) -¿ [Int] -¿ [Int] = fun f -¿ fun l -¿ case l — [] =¿ [] — x::xs =¿ f(x) @ map(f)(xs) end in

Figure 4.15: Inconsistent Expectations Example

Error in the @ operator. But arises as inconsistent branches. Use type slicing to work out types of branches. **use search procedure to find location of the actual error! via a cast slice!** let map = fun f -¿ fun l -¿ case l — [] =¿ [] — x::xs =¿ f(x) @ map(f)(xs) end in

Figure 4.16: Inconsistent Branches Example

[17] Indeterminate evaluation is very inefficient in comparison. This counter-examples logic is useful for directing hole instantiation (section 4.7.5).

[18] Also, Hazel has plans to introduce global inference in the future, where correct localisation becomes even more difficult.

[19] Except for *inconsistent branches* errors.

More difficult, non-local example where it's harder to work out the types of the branches immediately: let f = fun x -¿ x + 1 in let g = fun x -¿ x ++ "a" in let broken = fun x -¿ if x then f(x) else g(x)

Figure 4.17: Inconsistent Branches Example 2

COMPLEX ERROR Trying to use the curried fold function with uncurried add function. let fold : ((Int, Int) -¿ Int) -¿ Int -¿ [ ] -¿ Int = fun f -¿ fun init -¿ fun l -¿ case l — [] =¿ init — x::xs =¿ f(x, fold(f)(init)(xs)) end in let add = fun x -¿ fun y -¿ x + y in let sum = fold(add)(0) in sum([1,2,3])

Figure 4.18: Confused Currying Example

Type error is NOT spotted let map : (? -¿ ?) -¿ [?] -¿ [?] = fun f -¿ fun l -¿ case l — [] =¿ [] — x::xs =¿ f(x) @ map(f)(xs) end in

(a) Program

let map : (? -¿ ?) -¿ [?] -¿ [?] = fun f -¿ fun l -¿ case l — [] =¿ [] — x::xs =¿ f(x) @ map(f)(xs) end in

(b) Static Regions (Contibution Slice)

Figure 4.19: Partially Annotated Program

4.8.3 Usability Improvements

As designing an intuitive user interface was not a core goal, the actual use of the features can be quite awkward or unintuitive. Below, I propose various improvements, for which the architecture, of both Hazel and the newly implemented features, is sufficiently abstract and flexible to easily support:[20]

- Displaying type slices only *upon request* using the Hazel *context inspector*, which displays the *analysing* and *synthesising* types of the selected expression, see fig. 4.20. This can also improve live-editing performance (see section 4.7.2).
- A UI allowing the user to deconstruct type slices to query, for example: if a term has a function type, they could select the *return type* and get only the part of the slice relevant to that.
- Graphs visualising cast dependence (**??**). Showing the *execution* context that leads to a cast error; these would be more concise summaries than full evaluation traces.
- Graphs visualising the search procedure's execution traces and instantiations.

[20] Some of which were detailed in the Implementation chapter.

Screenshot the Hazel context inspector, including a case with an error, and with analysing and synthesising types.

Figure 4.20: The Hazel Context Inspector

- Key bindings to more quickly cycle through indeterminate evaluation instantiation paths.
- Integration with the Hazel stepper,[21] which could also allow (optionally) user-directed hole instantiations. The Hazel stepper could also be improved to allow other standard techniques like stepping over functions calls.

[21] The inbuilt trace visualiser, which also allows arbitrary evaluation order.

Chapter 5

Conclusions

This project aimed to improve the type error debugging experience in Hazel. Two novel features, *type slicing* and *cast slicing*, were mathematically formalised and implemented successfully. Additionally, upon evaluation, variations on these ideas were devised and implemented.

These type slicing implementations provided more complete *explanations* for why expressions were given their static type, and therefore, why type errors were detected. The feature was much more *expressive* than originally aimed for (applying to *all* expression rather than just errors), and it's variants applied to *more situations*[1] than set out in my core goals.

Cast slicing successfully provided a link between *dynamic errors* (cast errors), and the static context (source code) which sourced the parts of the cast. These slices were found to be very small, giving very specific and concise information to the programmer.

Additionally, a *type error witness search procedure* improve the explanation of both static and dynamic type errors in the Hazel language. Together, these features provide richer, more intuitive diagnostics by offering both abstract and concrete perspectives on type errors. Evaluation results show that the features are largely effective and performant, complementing each other and the existing error highlighting, to form a cohesive debugging experience.

5.1 Further Directions

This section presents some *extensions* to improve the features as justified by the evaluation (section 4.7 & **??**). Also, further interesting *applications* of the features are considered.

5.1.1 Extension to Full Hazel Language

Some of the Hazel language was not covered, or was partially covered. Type functions (System-F style polymorphism) was only partially supported. Type functions have especially difficult typing rules, which may lead to more subtle errors, where this project would be *particularly useful*.

5.1.2 UI Improvements, User Studies

The current UI is relatively simple. Section 4.8.3 suggested various improvements to the usability of all three features.

[1] Including: calculating static code regions, extracting inconsistent sub-parts of slices.

To assess how well these features actually work, and compare the usefulness of the different slicing methods, user studies would be beneficial.

Seidel et al. [24] implemented a similar *type error witness search procedure*, evidencing it's utility by a user study. They additionally made use of various techniques to improve usability including: jump-compressed execution traces[2], graph visualisation, and interactive steppers.

5.1.3 Cast Slicing

Cast slicing purely propagated type slice information throughout evaluation, and tracked the a dependency relation on casts. While this is useful for debugging type errors, as demonstrated, it provides no slicing based on *execution properties* of the program. Extensions could include slicing methods which, for example, provide a minimal program which evaluates to the *same* cast around the *same* value. This would be similar to traditional *dynamic slicing* methods [70, 34] which take parts of the program relevant to producing a *given* result.

5.1.4 Proofs

Giving proofs of the type slicing theory was an unmet *extension* goal. However, *much thought*[3] was given to ensuring that the theory is still sensible, and correct, as is demonstrated by the reasoning provided in the implementation (section 3.1).

Additionally, proofs and formal semantics for cast slicing and the search procedure would be useful to formalise what exactly cast slices, and witness actually *are*.

5.1.5 Property Testing

Indeterminate evaluation allows for exhaustive generation of inputs to functions, by applying a hole to the function. The searching logic is sufficiently abstract to allow arbitrary property testing of resulting values, indeterminate expressions, and even intermediate expressions.

The current algorithm generates smaller inputs first,[4] similarly to SmallCheck [44]; the approach being justified by the *small scope hypothesis* (section 4.3).

Other property testing models could be implemented, for example QuickCheck [57] performs random generation, along with input reduction to find simpler inputs which cause the same error.

Testing of intermediate expressions is *not possible*[5] in either SmallCheck or QuickCheck. Indeterminate evaluation would easily allow testing a property like: *Does every execution trace have a length of 5?*.

5.1.6 Non-determinism, Connections to Logic Programming, and Program Synthesis

Discuss this...

[2] Hiding the execution of functions, except for when the error occurs.
[3] The largest time-investment of the entire project.
[4] By default. This is also customisable, as in SmallCheck.
[5] Without refactoring the tested code.

Allowing holes to evaluate non-deterministically could be harnessed directly to write non-deterministic algorithms themselves in Hazel. The results to search for could be defined in Hazel code, and injected into the indeterminate evaluation algorithm.

This way of treating holes is reminiscent of *free logic variables* in logic programming, of which functional logic programming languages [37] like *Curry* [**CurryLang**]. Adding unification [68] to turn these into full-blown logic variables would allow full logic programming in Hazel. A needed-narrowing evaluation strategy [56] would be an interesting, and more efficient, way to handle and extend indeterminate evaluation in this situation.

Logic programming has historically been used for *general-purpose program synthesis* [**LogicProgramSy**
An incomplete Hazel program could then be used as a specification for it's incomplete parts, where executing the program would synthesise the incomplete parts. However, this approach has been found to be inefficient due to the need to solve *every* possible program, and prone to underspecified problems, among other drawbacks [66].

More modern approaches to program synthesis often use logic programming as specification, but use more specialised and scalable methods making use of machine learning to synthesise the programs [19, 38]. Therefore, an incomplete Hazel (logic) program sketch could be used as a specification to synthesise the incomplete portions automatically, using more modern methods.

5.1.7 Symbolic Execution

The search procedure was not always able to find witnesses, this was suspected to be due to low code coverage (**??**). Improving code coverage for testing has been extensively researched, often via symbolic execution [17]. Often, constraints along execution paths are modelled via *satisfiability modulo theories* (SMTs) [36], for which there exist many efficient solvers [41].

The evaluation of holes can already be consider a form of *symbolic execution*. However, for indeterminate evaluation, the only constraints considered are the type of the hole. Section 3.6.6 considered constraining instantiation based on structures of pattern within match statements. Full symbolic execution would also consider more complex constraints on the *values* of base types.

Polymorphism

The set of possible types is infinite. Generating witnesses for polymorphic values is therefore a much expanded state-space. Symbolic theories and solvers for polymorphic operations, would help in this situation [40].

5.1.8 Let Polymorphism & Global Inference

Errors in the presence of global inference are often more subtle ([**SubtleOCamlErrors**]), as there are fewer annotations which concretely assert the types for expressions. In this situation, type slicing, cast slicing, and the witness search procedure would be particularly useful.

Global inference is difficult to combine with complex type systems, for example polymorphism where global inference for Hazel's System-F style polymorphism is undecidable [61]). However, ML-family languages often implement restricted *let-polymorphism* via principal type schemes [**PrincipalTypeSchemes**].

The intersection of gradually typing and principal type schemes has been explored by Garcia and Cimini [25] Miyazaki et al. [15], the latter of which would integrate most smoothly with Hazel and indeterminate evaluation.

Hazel currently has a branch exploring global inference **(branch name here)**, although without polymorphism as of yet.

Constraint Slicing

The search procedure and cast slicing would be relatively easily extended to a let-polymorphic Hazel in the style of Miyazaki et al. However, type slicing would now have to consider non-local constraints involved and the code which sourced them, differing significantly from the proposed theory for bidirectional typing. Somewhat similar ideas have been explored in *constraint-based type error slicing* by Haack and Wells [49].

5.2 Reflections

TODO

Bibliography

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

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

[3] *Mercury Reference Manual: Clauses*. URL: https://mercurylang.org/information/doc-latest/mercury_ref/Clauses.html#Overview-of-Mercury-semantics (visited on 04/12/2025).

[4] *OCaml Effects Examples*. URL: https://github.com/ocaml-multicore/effects-examples/tree/master (visited on 03/26/2025).

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

[6] TIOBE Software. *TIOBE Programming Community Index*. 2025. URL: https://www.tiobe.com/tiobe-index/ (visited on 02/27/2025).

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

[8] The OCaml Development Team. *The OCaml Manual: release 5.2*. 2024. URL: https://ocaml.org/manual/5.2/index.html (visited on 02/28/2025).

[9] Eric Zhao et al. "Total Type Error Localization and Recovery with Holes". In: *Proceedings of the ACM on Programming Languages* 8.POPL (Jan. 2024), pp. 2041–2068. ISSN: 2475-1421. DOI: 10.1145/3632910. URL: http://dx.doi.org/10.1145/3632910.

[10] Yongwei Yuan et al. "Live Pattern Matching with Typed Holes". In: *Proc. ACM Program. Lang.* 7.OOPSLA1 (Apr. 2023).
DOI: 10.1145/3586048. URL: https://doi.org/10.1145/3586048.

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

[12] Jeremy G. Siek. *Parameterized Cast Calculi and Reusable Meta-theory for Gradually Typed Lambda Calculi*. 2021. arXiv: 2001.11560 [cs.PL]. URL: https://arxiv.org/abs/2001.11560.

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

[14] Oleg Kiselyov. "Effects Without Monads: Non-determinism – Back to the Meta Language". In: *Electronic Proceedings in Theoretical Computer Science* 294 (May 2019), pp. 15–40. ISSN: 2075-2180.
DOI: 10.4204/eptcs.294.2. URL: http://dx.doi.org/10.4204/EPTCS.294.2.

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

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

[17] Roberto Baldoni et al. "A Survey of Symbolic Execution Techniques". In: *ACM Comput. Surv.* 51.3 (2018).

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

[19] Lisa Zhang et al. "Neural Guided Constraint Logic Programming for Program Synthesis". In: *Advances in Neural Information Processing Systems*. Ed. by S. Bengio et al. Vol. 31. Curran Associates, Inc., 2018. URL: https://proceedings.neurips.cc/paper_files/paper/2018/file/67d16d00201083a2b118dd5128dd6f59-Paper.pdf.

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

[21] Matteo Cimini and Jeremy G. Siek. "The gradualizer: a methodology and algorithm for generating gradual type systems". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. ACM, Jan. 2016, pp. 443–455.
DOI: 10.1145/2837614.2837632. URL: http://dx.doi.org/10.1145/2837614.2837632.

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

[23] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. pearson, 2016.

[24] Eric L. Seidel, Ranjit Jhala, and Westley Weimer. "Dynamic witnesses for static type errors (or, ill-typed programs usually go wrong)". In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. ICFP'16. ACM, Sept. 2016, pp. 228–242.
DOI: 10.1145/2951913.2951915. URL: http://dx.doi.org/10.1145/2951913.2951915.

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

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

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

[28] Ohad Kammar, Sam Lindley, and Nicolas Oury. "Handlers in action". In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13. Boston, Massachusetts, USA: Association for Computing Machinery, 2013, pp. 145–158. ISBN: 9781450323260. DOI: 10.1145/2500365.2500590. URL: https://doi.org/10.1145/2500365.2500590.

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

[30] Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. *Real World OCaml: Functional programming for the masses*. O'Reilly Media, Inc., 2013.

[31] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2012.

[32] Oleg Kiselyov. "Typed Tagless Final Interpreters". In: *Generic and Indexed Programming: International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures*. Ed. by Jeremy Gibbons. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 130–174. ISBN: 978-3-642-32202-0. DOI: 10.1007/978-3-642-32202-0_3. URL: https://doi.org/10.1007/978-3-642-32202-0_3.

[33] Johannes Oetsch et al. "On the small-scope hypothesis for testing answer-set programs". In: *Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning*. KR'12. Rome, Italy: AAAI Press, 2012, pp. 43–53. ISBN: 9781577355601.

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

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

[36] Leonardo De Moura and Nikolaj Bjørner. "Satisfiability modulo theories: introduction and applications". In: *Commun. ACM* 54.9 (Sept. 2011), pp. 69–77. ISSN: 0001-0782. DOI: 10.1145/1995376.1995394. URL: https://doi.org/10.1145/1995376.1995394.

[37] Sergio Antoy and Michael Hanus. "Functional logic programming". In: *Commun. ACM* 53.4 (Apr. 2010), pp. 74–85. ISSN: 0001-0782. DOI: 10.1145/1721654.1721675. URL: https://doi.org/10.1145/1721654.1721675.

[38] Armando Solar-Lezama. "The Sketching Approach to Program Synthesis". In: *Proceedings of the 7th Asian Symposium on Programming Languages and Systems*. APLAS '09. Seoul, Korea: Springer-Verlag, 2009, pp. 4–13. ISBN: 9783642106712. DOI: 10.1007/978-3-642-10672-9_3. URL: https://doi.org/10.1007/978-3-642-10672-9_3.

[39] Philip Wadler and Robert Bruce Findler. "Well-Typed Programs Can't Be Blamed". In: *Programming Languages and Systems*. Springer Berlin Heidelberg, 2009, pp. 1–16. ISBN: 9783642005909.
DOI: 10.1007/978-3-642-00590-9_1. URL: http://dx.doi.org/10.1007/978-3-642-00590-9_1.

[40] François Bobot et al. "Implementing polymorphism in SMT solvers". In: *Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning*. SMT '08/BPR '08. Princeton, New Jersey, USA: Association for Computing Machinery, 2008, pp. 1–5. ISBN: 9781605584409.
DOI: 10.1145/1512464.1512466. URL: https://doi.org/10.1145/1512464.1512466.

[41] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[42] Conor McBride. "Clowns to the left of me, jokers to the right (pearl): dissecting data structures". In: *SIGPLAN Not.* 43.1 (Jan. 2008), pp. 287–295. ISSN: 0362-1340.
DOI: 10.1145/1328897.1328474. URL: https://doi.org/10.1145/1328897.1328474.

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

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

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

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

[47] Michael Spivey. "Algebras for Combinatorial Search". In: July 2006.
DOI: 10.14236/ewic/MSFP2006.11.

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

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

[50] Alexandr Andoni et al. "Evaluating the "Small Scope Hypothesis"". In: (Oct. 2002).

[51] Robert Bruce Findler and Matthias Felleisen. “Contracts for higher-order functions”. In: *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*. ICFP ’02. Pittsburgh, PA, USA: Association for Computing Machinery, 2002, pp. 48–59. ISBN: 1581134878.
DOI: 10.1145/581478.581484. URL: https://doi.org/10.1145/581478.581484.

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

[53] Conor McBride. “The derivative of a regular type is its type of one-hole contexts”. In: *Unpublished manuscript* (2001), pp. 74–88.

[54] Frank Pfenning and Rowan Davies. “A judgmental reconstruction of modal logic”. In: *Mathematical Structures in Computer Science* 11.04 (July 2001). ISSN: 1469-8072.
DOI: 10.1017/s0960129501003322. URL: http://dx.doi.org/10.1017/S0960129501003322.

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

[56] Sergio Antoy, Rachid Echahed, and Michael Hanus. “A needed narrowing strategy”. In: *J. ACM* 47.4 (July 2000), pp. 776–822. ISSN: 0004-5411. DOI: 10.1145/347476.347484. URL: https://doi.org/10.1145/347476.347484.

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

[58] Robert Harper and Christopher Stone. “A Type-Theoretic Interpretation of Standard ML”. In: *Proof, Language, and Interaction*. The MIT Press, May 2000, pp. 341–388. ISBN: 9780262281676. DOI: 10.7551/mitpress/5641.003.0019. URL: http://dx.doi.org/10.7551/mitpress/5641.003.0019.

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

[60] Michael Spivey. “Combinators for breadth-first search”. In: *J. Funct. Program.* 10.4 (July 2000), pp. 397–408. ISSN: 0956-7968.
DOI: 10.1017/S0956796800003749. URL: https://doi.org/10.1017/S0956796800003749.

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

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

[63] Benedict R Gaster and Mark P Jones. *A polymorphic type system for extensible records and variants*. Tech. rep. Technical Report NOTTCS-TR-96-3, Department of Computer Science, University of Nottingham, 1996.

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

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

[66] Y. Kodratoff, M. Franova, and D. Partridge. "Logic programming and program synthesis". In: *Systems Integration '90. Proceedings of the First International Conference on Systems Integration*. 1990, pp. 346–355. DOI: 10.1109/ICSI.1990.138700.

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

[68] Kevin Knight. "Unification: A multidisciplinary survey". In: *ACM Computing Surveys (CSUR)* 21.1 (1989), pp. 93–124.

[69] Luca Cardelli. "Structural subtyping and the notion of power type". In: *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1988, pp. 70–79.

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

[71] Philip Wadler. "How to replace failure by a list of successes a method for exception handling, backtracking, and pattern matching in lazy functional languages". In: *Functional Programming Languages and Computer Architecture*. Ed. by Jean-Pierre Jouannaud. Berlin, Heidelberg: Springer Berlin Heidelberg, 1985, pp. 113–128. ISBN: 978-3-540-39677-2.

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

[73] Mark Weiser. "Program slicing". In: *Proceedings of the 5th International Conference on Software Engineering*. ICSE '81. San Diego, California, USA: IEEE Press, 1981, pp. 439–449. ISBN: 0897911466.

[74] David HD Warren. "Applied logic: its use and implementation as a programming tool". PhD thesis. The University of Edinburgh, 1978.

[75] Robert W. Floyd. "Nondeterministic Algorithms". In: *J. ACM* 14.4 (Oct. 1967), pp. 636–644. ISSN: 0004-5411.
DOI: 10.1145/321420.321422. URL: https://doi.org/10.1145/321420.321422.

Appendix A

Hazel Formal Semantics

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

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

A.1 Syntax

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

Figure A.1: Syntax: *types* τ, *external expressions* e, *internal expressions* d. With x ranging over variables, u over hole names, σ over $x \to d$ *internal language* substitutions/environments, b over base types and c over constants.

A.2 Static Type System

A.2.1 External Language

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

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

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

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

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

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

Figure A.2: Bidirectional typing judgements for *external expressions*

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

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

Figure A.3: Type consistency

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

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

Figure A.4: Type Matching

A.2.2 Elaboration

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

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

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

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

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

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

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

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

A.2.3 Internal Language

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

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

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

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

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

Figure A.6: Type assignment judgement for *internal expressions*

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

$\Delta; \Gamma \vdash \sigma : \Gamma'$ 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

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

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

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

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

Figure A.9: Final forms

A.3.2 Instructions

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

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

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

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

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

Figure 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'

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

Figure A.12: Contextual dynamics of the internal language

A.3.4 Hole Substitution

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

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

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

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

Figure A.13: Hole substitution

Appendix B

Slicing Theory

B.1 Precision Relations

B.1.1 Patterns

B.1.2 Types

B.1.3 Expressions

B.2 Program Slices

B.2.1 Syntax

Including simplified syntaxes

B.2.2 Typing

B.3 Program Context Slices

B.3.1 Syntax

Include pattern contexts

B.3.2 Typing

B.4 Type Indexed Slices

B.4.1 Syntax

Including simplified syntaxes

B.4.2 Properties

B.5 Criterion 1: Synthesis Slices

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

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

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

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

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

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

Figure B.1: *Minimum synthesis slice* calculation

$\boxed{\Gamma \vdash e \dashv\!\Rightarrow \upsilon[\rho]}$ e synthesising type $\upsilon \blacktriangleright_{\text{type}} \tau$ 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.6 Criterion 2: Analysis Slices

B.7 Criterion 3: ...

B.8 Elaboration

Appendix C

Category Theoretic Description of Type Slices

Appendix D

Hazel Term Types

Could be merged into Hazel architecture

Appendix E

Hazel Architecture

More full description of the Hazel code-base architecture.

Appendix F

Code-base Addition Clarification

Clarification on which areas of the Hazel code-base were affected by my code, and what it did

Appendix G

Merges

List of merges performed during development which had overlap with my work.

Appendix H

Selected Results

H.1 Type Slicing

Selected examples from the corpus demonstrating each of the slicing techniques

H.2 Search Procedure

Selected examples for search procedure corpus, demonstrate examples which fail due to each class of difficulty discussed in evaluation. Plus some general examples which work.

Appendix I

Symbolic Execution & SMT Solvers

Discuss this further extension and feasibility. Especially for pattern matching. Does the code coverage percentage suggest that it is even needed?

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

Make sure the use of terms is consistent throughout dissertation.

Index

Project Proposal

Description

This project will add some features to the Hazel language [1]. 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 localised
  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. [24]. 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 [39], error and dynamic program slicing [55, 70], 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 [1] 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 [16].
- **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 [2].
- 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.