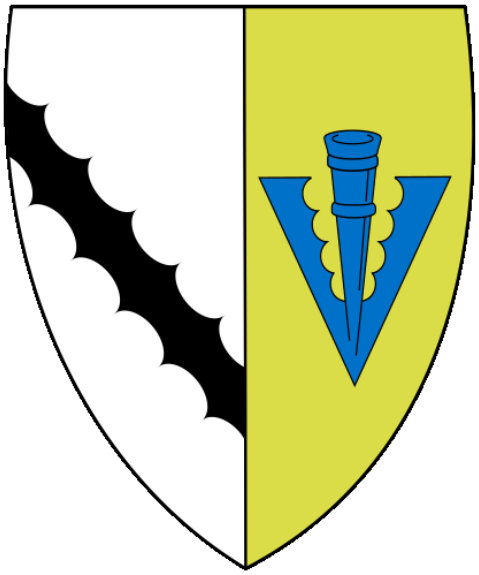
Max Carroll

Type Error Debugging in Hazel

Computer Science Tripos, Part II



Sidney Sussex College College
University of Cambridge
May 6, 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:	**22982** [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 [22], with programmers spending a *highly skewed* proportion of their time identifying and resolving a small proportion of *difficult* bugs [15].

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 [25]. 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* [41] 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: **type slicing**. Including formal mathematical foundations built upon the formal *Hazel calculus* [20]. Additionally, it generalises to highlight all code relevant to typing any expressions (not just errors).

2. Can we track 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.

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: **cast slicing**. Also having 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. [29] which devised a similar procedure for a subset of OCaml.

Hazel [1] is a functional, locally inferred, and gradually typed research language that allows writing *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 as well as how the two classes of errors interact.

Example: Figure 1.1 shows an attempt at writing an int list concatenation function. But list cons (`::`) is used instead of list concatenation (`@`). Type slicing will automatically highlight the code that caused `x` to synthesise the `[Int]` type and the code that enforces the requirement that `x` is an `Int`. If the error is still not clear, the search procedure can be used to generate inputs which evaluate to cast errors, for example `concat([[], []])`, giving a concrete evaluation trace to an error. Finally, cast slicing will allow the cast errors to be selected, highlighting source code that enforced the cast which can be more concise than statically computed type slices. The results of this example among others are explored in the evaluation (section 4.9.2).

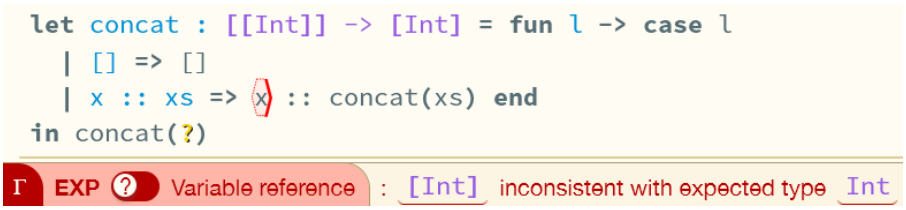


Figure 1.1: A Static Type Error from the Hazel Editor

1.1 Related Work

There has been extensive research into the field of programming languages and debugging, attempting to understand *what* is needed [34], *how* developers fix bugs [17], and a plethora of compiler improvements and tools. This project builds upon this body of research in new ways focusing on the Hazel language, which is itself a research project being taken in various directions but of particular note as a *teaching language* [16] for students, where this features can additionally help with teaching understanding of bidirectional type systems.

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 [81], though my definition of

program slices matches more with functional program slices [40]. The properties I explore are more similar to *dynamic program slicing* [77] but with type characteristics rather than evaluation characteristics, in a sense similar *type error slicing*[60, 56].

The *type witness search procedure* is based upon Seidel et al. [29], though there are significant differences in workings due to my use of Hazel and various extensions as compared to the subset of OCaml used by Seidel et al.

1.2 Dissertation Outline

Chapter 2 introduces the *type theories* (section 2.1.1) underpinning the *Hazel core calculus* (section 2.1.2), followed by basic background on the *Hazel implementation* (section 2.1.3), and methods of non-deterministic programming (section 2.1.4).

Section 3.1 and section 3.2 conceive and formalise the ideas of *type slices* and *cast slices* applied to the Hazel core calculus.

An implementation (section 3.3-3.4) of the slicing theories was created covering *most*[2] 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[3] (section 4.2), showing *effectiveness* (section 4.7) and being reasonably *performant* (section 4.6) over a corpus of *well-typed* and *ill-typed* Hazel programs respectively (section 4.5). Further, the features weaknesses were evaluation, with considered improvements implemented or devised (section 4.8). Additionally, a holistic evaluation was performed (section 4.9), 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.

[2]Except for type substitution.

[3]Including those on the project proposal, this evaluation is more extensive.

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. The following sections expect basic knowledge formal methods of type systems in terms of judgements (appendix A reviews this). Note that I will use *partial functions* to represent typing assumption contexts.

Dynamic Type Systems

Dynamic typing has purported strengths allowing rapid development and flexibility, evidenced by their popularity [52, 10]. Of particular relevance to this project, execution traces are known to help provide insight to errors [41], 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*[1] [74]. 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*[2] cast expressions which maintain and enforce runtime type constraints alongside a dynamic type written `?`.

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

[1] 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.

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

- *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 treated 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*, $v\langle\tau_1\Rightarrow?\Rightarrow\tau_2\rangle$, 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*[3] functions decomposes the cast into casting the applied argument and then the result:

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

Or if f has the dynamic type:

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

The direction of the casts reflects the *contravariance* [71, ch. 2] of functions[4] 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* [31, 54] combines static and dynamic typing. Terms may be annotated as dynamic, marking regions of code omitted from type-checking but still *interoperable* with static code. For example, the following type checks:

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

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

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

- The *external language* – where static type checking is performed which allows annotating expressions with the dynamic type.

[3] Wrapped in a cast between function types.
[4] A bifunctor.

- The *internal language* – where evaluation and runtime type checking is performed via cast expressions.[5] The example above would reduce to a *cast error*[6]:

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

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

Consistency must satisfy a few properties: that the dynamic type is consistent with every type, that $\sim$ is reflexive and symmetric, and two concrete types (having no dynamic sub-parts) are consistent iff they are equal[7]. Finally, differing this from type equality, that every type τ is consistent with the dynamic type $?$:

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

This is very similar to the notion of *subtyping* [58, 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, used to account for if $\Gamma \vdash e_1 : ?$, where we treat $?$ then as a dynamic function $? \blacktriangleright_\to ? \to ?$. Intuitively, $e_1(e_2)$ has type τ_2' if e_1 has type $\tau_1' \to \tau_2'$ or $?$ (then treated as $? \to ?$), 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 [63] by elaboration to an explicitly typed internal language XML [70]. The *elaboration judgement* $\Gamma \vdash e \rightsquigarrow d : \tau$ read as: external expression e is elaborated to internal expression d with type τ under typing context Γ. For example 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[8] on the function d_1 to $\tau_2 \to \tau$ and on the argument d_2 to the function's expected argument type τ_2 to perform runtime type checking of arguments. Intuitively, casts must be inserted whenever type consistency is used, but which casts to insert are non-trivial [26].

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

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

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

[8] 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.

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*.

Bidirectional Type Systems

A *bidirectional type system* [32] takes on a more algorithmic definition of typing judgements, being more intuitive to implement. They also allow some amount of local type inference [64], allowing programmers to omit *type annotations*, with some being inferred directly from code. Global type inference systems [58, ch. 22] can be *difficult to implement*, often via constraint solving [11, ch. 10], and difficult or impossible to *balance* with complex language features, for example global inference in System F (**??**) is undecidable [66].

This is done in a similar way to annotating logic programs [82, 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* [80]. 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[9] 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 have three obvious rules. That variables can synthesise their type, being accessible from the typing assumptions. Annotated terms synthesise their type from the annotation (after checking the validity). Subsumption: a synthesising term successfully checks against that same type.

2.1.2 The Hazel Calculus

Hazel is a language that allows the writing of incomplete programs, evaluating them, and evaluating around static and dynamic errors.[10]

[9] Highlighted in red as an error.

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

It does this via adding *holes*, which can both be typed and have evaluation proceed around them seamlessly. Errors can be placed in holes allowing continued evaluation.

The core calculus [20] 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 B, with only rules relevant *holes* 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.[11]

Syntax

The syntax, in Fig. 2.1, consists of *types* τ including the dynamic type ?, *external expressions* e including (optional) annotations, *internal expressions* d including cast expressions. The external language is *bidirectionally typed*, and therefore is a locally inferred surface syntax for the language, and is statically elaborated to (explicitly typed) *internal expressions*.

Notating $(\!|\,|\!)^u$ or $(\!|e|\!)^u$ for empty and non-empty holes respectively, where u is the *metavariable* or name for a hole. Internal expression holes, $(\!|\,|\!)^u_\sigma$ or $(\!|e|\!)^u_\sigma$, also maintain an environment σ mapping variables x to internal expressions d. These internal holes act as *closures*, recording which variables have been substituted during evaluation.[12]

$$
\begin{aligned}
\tau &::= b \mid \tau \to \tau \mid ? \\
e &::= c \mid x \mid \lambda x : \tau . e \mid \lambda x . e \mid e(e) \mid (\!|\,|\!)^u \mid (\!|e|\!)^u \mid e : \tau \\
d &::= c \mid x \mid \lambda x : \tau d \mid d(d) \mid (\!|\,|\!)^u_\sigma \mid (\!|d|\!)^u_\sigma \mid d\langle \tau \Rightarrow \tau \rangle \mid d\langle \tau \Rightarrow ? \Rightarrow \tau \rangle
\end{aligned}
$$

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:

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

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

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

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.

Internal Language

The internal language is non-bidirectionally typed and requires an extra *hole context* Δ mapping each hole *metavariable* u to it's *checked type* τ [13] 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) [49].[14]

The type assignment judgement $\Delta; \Gamma \vdash d : \tau$ means that d has type τ under typing and hole contexts Γ, Δ. The rules for holes take their types from the hole context and ensure that the hole environment substitutions σ are well-typed[15]:

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

A well-typed external expression will elaborate to a well-typed internal expression consistent with the external type. Hence, there is no need for an algorithmic definition of the internal language typing.

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 $\sigma = \mathrm{id}(\Gamma)$, i.e. no substitutions.

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

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

When elaborating a *type checked* hole, this checked type is used. Typing them instead as $?$ would lose type information.[16]

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

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

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

[16] Potentially leading to incorrect cast insertion.

Final Forms

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

- *Values* – Constants and functions.
- *Boxed Values* – Values wrapped in *injection* casts, or *function*[17] casts.
- *Indeterminate Final Forms* – Terms containing holes that cannot be directly evaluated, e.g. holes or function applications where the function is indeterminate, e.g. $(\!|\,|\!)^u(1)$.

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

Dynamics

A small-step contextual dynamics [27, ch. 5] is defined on the internal expressions to define a *call-by-value* evaluation order, values in this sense are *final forms*.

Like the *refined criteria* [31], Hazel presents a rather different cast semantics designed around *ground types*, that is, base types (`Int`, `Bool`) and least precise[18] 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.[19]

The cast calculus is therefore more complex. However, the fundamental logic is the same as 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). Substitutions are recorded in each hole's environment σ by substituting all occurences of x for d in each σ.

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, after first substituting any occurrences of u in the hole's environment σ. This corresponds to *contextual substitution* in CMTT [49].

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.

[17] Between function types

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

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

As Hazel is a *pure language*[20] 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.

2.1.3 The Hazel Implementation

The Hazel implementation [2] is written primarily in ReasonML and OCaml with approximately 65,000 lines of code. It implements the Hazel core calculus along with many additional features.

Language Features

The relevant additional language features not already discussed are:[21]

- **Lists** – Linked lists, in the style of ML. By use of the dynamic type, Hazel can represent *heterogeneous* lists, which may have elements of differing types.
- **Tuples & Labelled Tuples**[22] – Allowing compound terms to be constructed and typed [58, ch. 11.7-8].
- **Sum Types** – Representing a value as one of many labelled variants, each of possibly different types [58, ch. 11.10].
- **Type Aliases** – Binding a name to a type, used to improve code readability or simplify complex type definitions redefining types.
- **Pattern Matching** – Checks a value against a pattern and destructures it accordingly, binding it's sub-structures to variable names.
- **Explicit Polymorphism** – System F style parametric polymorphism [58, ch. 23]. Where explicit type functions bind arbitrary types to names, which may then be used in annotations. Polymorphic functions are then applied to a type, uniformly giving the corresponding monomorphic function. Implicit and ad-hoc polymorphic functions can still be written as dynamic code, without use of type functions.[23]
- **Iso-Recursive Types** – Types defined in terms of themselves, allowing the representation of data with potentially infinite or *self-referential* shape [58, ch. 22-23], for example linked lists or trees.

Evaluator

The Hazel implementation has a complex evaluator abstraction (module type `EV_MODE`) which is used extensively by the search procedure implementation. The evaluator implementations 'evaluate' parametric values, not necessarily having to be terms, for example:

[20] Having no side effects.
[21] Additionally, Hazel supports other features, which do not concern this project.
[22] Merged towards the end of the project's development.
[23] In which case, they are untyped.

- **Final Form Checker** – Returns whether a term is either: a evaluable expression, a value, or an indeterminate term. 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* as the implementation does not actually *perform* any evaluation steps which the abstraction presents it with.
- **Evaluator** – Maintains a stack machine to actually perform the reduction steps and fully evaluate terms.
- **Stepper** – Returns a list of *possible*[24] evaluation steps. This allows the evaluation order to be user-controlled in a stepper environment.

Each evaluation method is transformed into an evaluator by being applied to an OCaml *functor* [35, ch. 10]. The resulting module produces a `transition` method that takes terms to evaluation results in an environment.[25]

2.1.4 Bounded Non-Determinism

The search procedure [29] 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 [83].

High Level

At a high level, non-determinism can be represented declaratively 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 non-deterministic result of the algorithm has type τ. These can be represented by operations:

$$\texttt{<||>} : \tau \to \tau \to \tau$$

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

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

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

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 represent and manage non-deterministic programs which I considered, of which a monadic representation over a tree based state-space model was chosen as a good balance of flexibility, simplicity, and familiarity to other Hazel developers. Appendix C reasons and details other options considered: continuations, effect handlers, tagless final DSLs, direct implementation.

[24] Of any evaluation order.
[25] Mapping variable bindings.

Monadic Non-determinism: Some monads (appendix D explains monads) can be extended to represent non-determinism by adding the `choice` and `fail` operators satisfying the usual laws. These operations interact by `bind` distributing over `choice`, and `fail` being a left-identity for `bind`.

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

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

In this context, bind can be thought of as *conjunction*: if we can map each guess to another set of choices, `bind` will conjoin all the choices from every guess. Figure 2.2 demonstrates how flipping a coin followed by rolling a dice can be conjoined, yielding the choice of all pairs of coin flip and dice roll.

Distributivity represents this interpretation: guessing over a combined choice is the same as guessing over each individual choice and then combining the results. `fail` being the left identity of `bind` states that you cannot make any guesses from the no choice (`fail`).

```
let coin = return(Heads) <||> return(Tails);
let dice = return(1) <||> ... <||> return(6);
let m = coin
    >>= flip => // Flip a coin
        dice
    >>= roll => // Roll a dice
        return((flip, roll)) // Return conjunction
```

Figure 2.2: Examples: Bind as Conjunction

While the `bind` and `return` operators are *not* required to represent non-determinism, they are a *familiar*[26] way to represent a large class of effects in general way.

Low Level

Computers are deterministic, therefore, the declarative specification abstracts over a low level implementation which will concretely try each choice, searching for solutions from the many choices. One way to resolve this non-determinism is by modelling choices as trees and searching the trees by a variety of either uninformed, or informed strategies.

2.2 Starting Point

Concepts

The foundations of most concepts in understanding Hazel from Part IB Semantics of Programming (and Part II Types later). The concept of gradual typing briefly appeared in Part IB Concepts of Programming Languages, but was not formalised. Dynamic typing, gradual typing, holes, and contextual modal type theory were not covered in Part IB, so were partially researched leading up to the project, then researched further in greater depth during the early stages. Similarly, Part IB Artificial Integlligence provided some context for search procedures.

[26] For OCaml developers and functional programmers.

Primarily, the OCaml search procedure for ill-typed witnesses Seidel et al. [29] and the Hazel core language [20] 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:

1. 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.
2. 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, made upon critical evaluation, are detailed throughout.

3.1 Type Slicing Theory

I develop a novel method, *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 non-obvious mathematical machinery: *context typing slices* (section 3.1.2), *checking contexts* (definition 20), and *type-indexed slices* (section 3.1.3).

3.1.1 Expression Typing Slices

First, I introduce what *slices* are in this context. The aim is to provide a formal representation of term *highlighting*.

Term Slices

A *term slice* is a term with some sub-terms omitted. The omitted terms are those that are *not* highlighted. For example if my slicing criterion is to *omit terms which are typed as* `Int`, then the following expressions highlights as:

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

Omitted sub-terms are replaced by a *gap* term, notated $\square$. Representing the highlighted example above, we get the slice:

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

Similarly, this one can omit variable names and parts of types. We get a syntax specified in definition 1, extending the Hazel core syntax from fig. 2.1:

Definition 1 (Term Slice Syntax) *Pattern expression slices p:*

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

Type slices υ:[1]

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

Expression slices ς:

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

We then have a *precision* relation on term 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 sub-terms may be gaps. Full definition in appendix **ref appendix**. For example, see this precision chain:

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

Precision forms a partial order [86], that is, a reflexive, antisymmetric, and transitive relation. 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 terms*[2] e which are *top* elements of their respective chains: if $e \sqsubseteq \varsigma$ then $e = \varsigma$. Sometimes I refer to this relation as ς_1 is a term slice *of* ς_2 iff $\varsigma_1 \sqsubseteq \varsigma_2$.

For any *complete term* t, the slices of t form a *bounded lattice structure* [85]. That is, every pair ς_1, ς_2 has a *join* $\varsigma_1 \sqcup \varsigma_2$ and *meet* $\varsigma_1 \sqcap \varsigma_2$:

Proposition 1 (Term Slice Bounded Lattice) *For any term t, the set of slices ς_1 and ς_2 of t forms a bounded lattice:*

[1] A *type slice* is also used to refer to this whole mechanism. The overall theory is referred to *typing slices* to distinguish from slices of type terms where ambiguous.

[2] With no gaps.

- *The join,* $\varsigma_1 \sqcup \varsigma_2$ *exists , being an upper bound for* ς_1, ς_2*:*

$$\varsigma_1 \sqsubseteq \varsigma_1 \sqcup \varsigma_2 \qquad \varsigma_2 \sqsubseteq \varsigma_1 \sqcup \varsigma_2$$

  And, any other slice $\varsigma \sqsubseteq t$ *which is also an upper bound of* ς_1, ς_2 *is more or equally precise than the join:*

$$\varsigma_1 \sqcup \varsigma_2 \sqsubseteq \varsigma$$

- *The meet,* $\varsigma_1 \sqcap \varsigma_2$ *exists, being a lower bound for* ς_1, ς_2*:*

$$\varsigma_1 \sqcap \varsigma_2 \sqsubseteq \varsigma_1 \qquad \varsigma_1 \sqcap \varsigma_2 \sqsubseteq \varsigma_2$$

  And, any other slice $\varsigma \sqsubseteq t$ *which is also a lower bound of* ς_1, ς_2 *is less or equally precise than the meet:*

$$\varsigma \sqsubseteq \varsigma_1 \sqcup \varsigma_2$$

- *And the join and meet operations satisfy the absorption laws for any slice* ς*:*

$$\varsigma_1 \sqcap (\varsigma_1 \sqcup \varsigma_2) = \varsigma_1 \qquad \varsigma_1 \sqcup (\varsigma_1 \sqcap \varsigma_2) = \varsigma_1$$

  And idempotent laws:

$$\varsigma_1 \sqcup \varsigma_1 = \varsigma_1 = \varsigma_1 \sqcap \varsigma_1$$

- *The lattice is bounded. That is,* $\square \sqsubseteq \varsigma \sqsubseteq t$*.*

Note that not every pair of slices has a join: consider 1 and 2. But, every pair does have a meet as for all ς_1, ς_2, $\square \sqsubseteq \varsigma_1$ and $\square \sqsubseteq \varsigma_2$.

Typing Assumption Slices

Expression typing is performed given a set of *typing assumptions*. Therefore, in addition to a slice of expressions, we also desire a slice of the typing assumptions, to represent the *relevant*[3] parts of the typing assumptions. Typing assumptions are *partial functions* mapping variables to types notated $x : \tau$ (see appendix A).

Therefore, *typing assumptions slices* can be represented by partial functions mapping variables to *type slices*. That is, a typing assumption set is a slice of another if it maps *no more* variables to *at most* as specific types. The precision relation, $\sqsubseteq$, can be extended to typing assumptions by extensionality [84] and using the subset relation:

Definition 2 (Typing Assumption Slice Precision) *For typing assumption slices* γ_1, γ_2*. Where* $\mathrm{dom}(f)$ *is the set of variables for which a partial function* f *is defined:*

$$\gamma_1 \sqsubseteq \gamma_2 \iff \mathrm{dom}(\gamma_1) \subseteq \mathrm{dom}(\gamma_2) \text{ and } \forall x \in \mathrm{dom}(\gamma_1).\ \gamma_1(x) \sqsubseteq \gamma_2(x)$$

Equally, joins and meet can be extended extensionality:

Definition 3 (Typing Assumption Slice Joins and Meets) *For typing slices* γ_1, γ_2*, and any variable* x*:*

[3] Relevant to the given criterion.

- *If* $\gamma_1(x) = \bot$ *then* $(\gamma_1 \sqcup \gamma_2)(x) = \gamma_2(x)$ *and* $(\gamma_1 \sqcap \gamma_2)(x) = \bot$.
- *If* $\gamma_2(x) = \bot$ *then* $(\gamma_1 \sqcup \gamma_2)(x) = \gamma_1(x)$ *and* $(\gamma_1 \sqcap \gamma_2)(x) = \bot$
- *Otherwise,* $(\gamma_1 \sqcup \gamma_2)(x) = \gamma_1(x) \sqcup \gamma_2(x)$.

Again, the slices γ of complete typing assumptions Γ form a bounded lattice. Similarly, not every pair of slices has a join, as not every type slice has a join: consider $x : \texttt{Int}$ and $x : \texttt{String}$.

Proposition 2 (Typing Assumption Slices form Bounded Lattices) *For any typing assumptions* Γ*, the set of slices* γ *of* Γ *form a bounded lattice with bottom element of the empty function* $\emptyset$ *and top element* Γ.

Expression Typing Slices

Finally, an *expression typing slice*, ρ, is a pair, ς^{γ}, of a term slice and a typing slice. Precision, joins and meets, can be extended pointwise to term typing slices with all the same properties:

Definition 4 (Expression Typing Slice Precision) *For expression typing slices* $\varsigma_1^{\gamma_1}$, $\varsigma_2^{\gamma_2}$*:*

$$\varsigma_1^{\gamma_1} \sqsubseteq \varsigma_2^{\gamma_2} \iff \varsigma_1 \sqsubseteq \varsigma_2 \text{ and } \gamma_1 \sqsubseteq \gamma_2$$

Definition 5 (Expression Typing Slice Joins and Meets) *For expression typing slices* $\varsigma_1^{\gamma_1}$, $\varsigma_2^{\gamma_2}$*:*

$$\varsigma_1^{\gamma_1} \sqcup \varsigma_2^{\gamma_2} = (\varsigma_1 \sqcup \varsigma_2)^{\gamma_1 \sqcup \gamma_2}$$
$$\varsigma_1^{\gamma_1} \sqcap \varsigma_2^{\gamma_2} = (\varsigma_1 \sqcap \varsigma_2)^{\gamma_1 \sqcap \gamma_2}$$

For any pair of complete expressions e and typing assumptions Γ, then the expression typing slices of these forms a bounded lattice.

Proposition 3 (Expression Typing Slices for Bounded Lattices) *For any expression* e *and typing assumption* Γ*, the set of expression typing slices* ς^{γ} *of* e^{Γ} *forms a bounded lattice with bottom element* $\square^{\emptyset}$ *and top element* e^{Γ}.

The *expression slices* can be *type checked* under the *type assumption slices* by replacing gaps $\square$ by: holes of arbitrary metavariable $(\!|\,|\!)^u$ in *expressions*, fresh variables in *patterns*, and the dynamic type in *types*. This transformation $[\![\cdot]\!]$ is stated formally in the appendix **(fig APPENDIX)**.

Definition 6 (Expression Typing Slice Type Checking) *For expression typing slice* ς^{γ} *and type* τ*.* $\gamma \vdash e \Rightarrow \tau$ *iff* $[\![\gamma]\!] \vdash [\![e]\!] \Rightarrow \tau$ *and* $\gamma \vdash e \Leftarrow \tau$ *iff* $[\![\gamma]\!] \vdash [\![e]\!] \Leftarrow \tau$.

3.1.2 Context Typing Slices

Next, some of an expression's type might be enforced by the surrounding *context*. For example, the type of the underlined expression below is enforced by the surrounding highlighted annotation:

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

Contexts & Context Slices

We represent these surrounding contexts by a *term context* $\mathcal{C}$. Which marks *exactly one* sub-term as $\bigcirc$:

Definition 7 (Contexts Syntax) *Pattern contexts – mapping patterns to patterns:*

$$\mathcal{P} ::= \bigcirc \mid x$$

Type contexts – mapping types to types:

$$\mathcal{T} ::= \bigcirc \mid \texttt{?} \mid b \mid \mathcal{T} \to \tau \mid \tau \to \mathcal{T}$$

Expression contexts – mapping patterns, types, or expression to expressions:[4]

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

Where $\mathcal{C}\{e\}$ substitutes expression e for the mark $\bigcirc$ in $\mathcal{C}$, the result of this is necessarily an expression. Similarly for pattern and type contexts $\mathcal{P}, \mathcal{T}$ whose substitutions return patterns and types. Additionally, contexts are *composable*: substituting a context into a context, $\mathcal{C}_1\{\mathcal{C}_2\}$ produces another valid context when the domain of $\mathcal{C}_1$ is from expressions. Notate this by $\mathcal{C}_1 \circ \mathcal{C}_2$. From here on I notate all contexts like functions, specifying their domain and co-domain where appropriate: $\mathcal{C} : \texttt{Typ} \to \texttt{Exp}$ for example.

Then, contexts can be extended to slices, c, analogously to the previous section. However, the precision relation $\sqsubseteq$ is defined differently, requiring that the mark $\bigcirc$ must remain in the same position in the structure. For example $\bigcirc(\square) \sqsubseteq \bigcirc(1)$, but $\bigcirc \not\sqsubseteq \bigcirc(1)$. This can be concisely defined by *extensionality*:

Definition 8 (Context Precision) *If $c : X \to Y$ and $c' : X \to Y$ are context slices, then $c' \sqsubseteq c$ if and only if, for all terms t of class X, that $c'\{t\} \sqsubseteq c\{t\}$.*

Again, we refer to a context slice c' *of* c as one satisfying that $c' \sqsubseteq c$.

We also get that filling contexts preserves the precision relations both on term slices *and* context slices:[5]

Conjecture 1 (Context Filling Preserves Precision) *For context slice $c : X \to Y$ and term slice ς of class X. Then if we have slices $\varsigma' \sqsubseteq \varsigma$, $c' \sqsubseteq c$ then also $c'\{\varsigma'\} \sqsubseteq c\{\varsigma\}$.*

Joins and meets can be defined via extensionality:

Definition 9 (Context Slice Joins & Meets) *For context slices $c_1 : X \to Y$ and $c_2 : X \to Y$ and any term t of class X:*

$$(c_1 \sqcup c_2)\{t\} = c_1\{t\} \sqcup c_2\{t\}$$
$$(c_1 \sqcap c_2)\{t\} = c_1\{t\} \sqcap c_2\{t\}$$

Again, these form a lattice:

Proposition 4 (Context Slices form Bounded Lattices) *For any context $\mathcal{C}$, the set of slices c of $\mathcal{C}$ form a bounded lattice with bottom element of the purely structural context*[6] *and top element $\mathcal{C}$.*

[4] Note that $\mathcal{C}$ is also used for generic term contexts sometimes.

[5] The conjectures are heavily suspected but not formally proven in the appendix.

[6] With the $\bigcirc$ is the *same position* but every other sub-term being a $\square$. Defined in appendix.

Typing Assumption Contexts & Context Slices

The accompanying notion of a *typing assumption slice* can be extended to *functions on typing assumptions*. This function represents what typing assumptions need to be *added*, or can be *removed* when typing an expression e and within it's context slice.

Definition 10 (Typing Assumption Contexts & Slices) *A typing assumption context $\mathcal{F}$ is a function from typing assumption to typing assumptions. A typing assumption context slice f is a function from typing assumption slices to typing assumption slices.*

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

Definition 11 (Typing Assumption Context Slice Precision) *If f' and f are typing assumption context slices, then $f' \sqsubseteq f$ if and only if, for all typing context slices γ, that $f'(\gamma) \sqsubseteq f(\gamma)$.*

Again, such functions are monotone:

Conjecture 2 (Function Application Preserves Precision) *For typing assumption slice γ and typing assumption context slice f. Then if we have slices $\gamma' \sqsubseteq \gamma$, $f' \sqsubseteq f$ then also $f'(\gamma') \sqsubseteq f(\gamma)$.*

Joins and meets are also defined extensionally:

Definition 12 (Typing Assumption Context Slice Joins & Meets) *For typing assumption context slices f_1 and f_2 and any typing assumption slice γ:*

$$(f_1 \sqcup f_2)(\gamma) = f_1(\gamma) \sqcup f_2(\gamma)$$

$$(f_1 \sqcap f_2)(\gamma) = f_1(\gamma) \sqcap f_2(\gamma)$$

Conjecture 3 (Typing Assumption Context Slices form Bounded Lattices) *For any typing assumption context $\mathcal{F}$, the set of slices f of $\mathcal{F}$ form a bounded lattice with bottom element being the constant function to the empty typing assumption function and top element $\mathcal{F}$.*

As usual, not all joins exist: if the assumption context slices map the slices to inconsistent typing assumption slices.

Context Typing Slices

Finally, an *expression context typing slice*, $\mathfrak{p}$, is a pair, c^f, of an expression context slice and a typing assumption context slice. Precision, composition, application, joins and meets, can be extended pointwise to term typing slices with all the same properties:

Definition 13 (Expression Context Typing Slice Precision) *For expression context typing slices $c_1^{f_1}$, $c_2^{f_2}$:*

$$c_1^{f_1} \sqsubseteq c_2^{f_2} \iff c_1 \sqsubseteq c_2 \text{ and } f_1 \sqsubseteq f_2$$

Definition 14 (Expression Context Typing Slice Composition & Application) *For expression context typing slices c^f, $c'^{f'}$, and expression typing slices ς^γ:*

$$c^f \circ c'^{f'} = (c \circ c')^{f \circ f'}$$

$$c_1^{f_1}\{\varsigma^\gamma\} = c_1\{\varsigma\}^{f_1(\gamma)}$$

Definition 15 (Expression Context Typing Slice Joins and Meets) *For expression typing slices $c_1^{f_1}$, $c_2^{f_2}$:*

$$c_1^{f_1} \sqcup c_2^{f_2} = (c_1 \sqcup c_2)^{f_1 \sqcup f_2}$$
$$c_1^{f_1} \sqcap c_2^{f_2} = (c_1 \sqcap c_2)^{f_1 \sqcap f_2}$$

For any pair of expressions contexts $\mathcal{C}$ and typing assumption contexts $\mathcal{F}$, then the expression typing slices of these forms a bounded lattice.

Proposition 5 (Expression Context Typing Slices form Bounded Lattices) *For any expression context $\mathcal{C}$ and typing assumption context $\mathcal{F}$, the set of expression context typing slices c^f of $\mathcal{C}^\mathcal{F}$ forms a bounded lattice with bottom element, the purely structural context and the function to the empty typing assumptions, and top element e^Γ.*

3.1.3 Type-Indexed Slices

Tagging slices by their type and allowing decomposition into sub-slices for each part of the type is required for *cast slicing* and useful in calculating slices according to *analysis slices* and *contribution slices* criteria. For example consider the same context slice:

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

This context slice would be tagged with type `Bool` → `Int` and the respective sub-slice considering only the return type would omit the `Bool` term:

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

This section will only consider *context slices*, but term slices are type-indexed in exactly the same way. By extending the class of possible contexts, an expression slice can be considered as a *terminal context*[7] taking any input to the expression slice.[8]

Definition 16 (Term Slices as Terminal Context Slices) *A terminal context slice for a term typing slice ρ is a context typing slice p such that for every context typing slice p' and expression typing slice ρ':*

$$p \circ p' = p \text{ and } p\{\rho'\} = \rho$$

The main property that we want these indexed-slices to maintain is that a slice can be *reconstructed* from their sub-parts. *Joining* the sub-slices should produce the slice for the entire type. As not every slice can be joined, we can pair sub-slices with contexts which place the two sub-slices inside the same structure, making them join-able.

[7] Being a terminal morphism when thinking of morphisms in a category.
[8] These will be used interchangeably from now on.

Definition 17 (Type-Indexed Context Typing Slices) *Syntactically defined:*

$$\mathcal{S} ::= p \mid p * \mathcal{S} \to p * \mathcal{S}$$

With any $\mathcal{S}$ only being valid if it has a full slice. The full slice of $\mathcal{S}$ is notated $\overline{\mathcal{S}}$ and defined recursively:

$$\overline{p} = p$$
$$\overline{p_1 * \mathcal{S}_1 \to p_2 * \mathcal{S}_2} = p_1 \circ \overline{\mathcal{S}_1} \sqcup p_2 \circ \overline{\mathcal{S}_2}$$

Then left *(incremental)* composition and right *(global)* composition can be defined, by composing at the upper type constructor or at the leaves respectively:

Definition 18 (Type-Indexed Context Typing Slice Composition) *For type-indexed context typing slices $\mathcal{S}$ and $\mathcal{S}'$. If $\mathcal{S} = p$ and $\mathcal{S}' = p'$:*[9]

$$p' \circ p = \overline{p'} \circ \overline{p} \qquad p \circ p' = \overline{p} \circ \overline{p'}$$

If $\mathcal{S} = p$ and $\mathcal{S}' = p'_1 * \mathcal{S}'_1 \to p'_2 * \mathcal{S}'_2$:[10]

$$S \circ \mathcal{S}' = (p \circ p'_1) * \mathcal{S}'_1 \to (p \circ p'_2) * \mathcal{S}'_2$$

If $\mathcal{S} = p_1 * \mathcal{S}_1 \to p_2 * \mathcal{S}_2$:

$$\mathcal{S} \circ \mathcal{S}' = p_1 * (\mathcal{S}_1 \circ \mathcal{S}') \to p_2 * (\mathcal{S}_2 \circ \mathcal{S}')$$

This definition stems from it representing regular context typing slice composition over it's full slices.

Proposition 6 (Type-Indexed Composition Preserves Full Slice Composition) *For type-indexed slices $\mathcal{S}$ and $\mathcal{S}'$:*

$$\overline{\mathcal{S} \circ \mathcal{S}'} = \overline{\mathcal{S}} \circ \overline{\mathcal{S}'}$$

3.1.4 Criterion 1: Synthesis Slices

The first slicing method aims to concisely explain why an expression *synthesises* a type. It omits all sub-terms which analyse against a type retrieved from synthesising some other part of the program. For example, the following term synthesises a `Bool` → `Bool` type, and the variable x : `Int` and argument are irrelevant:

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

Definition 19 (Synthesis Slices) *For a synthesising expression, synthesiseτ. A synthesis slice is an expression typing slice ς^{γ} of e^{Γ} which also synthesises τ, that is, $[\![\gamma]\!] \vdash [\![\varsigma]\!] \Rightarrow \tau$.*

Proposition 7 (Minimum Synthesis Slices) *A minimum synthesis slice of e^{Γ} is a synthesis slice ρ such that any other synthesis slices ρ' are at least as specific, $\rho \sqsubseteq \rho'$. This minimum always uniquely exists.*

[9] In shorthand, the overline for a base type slice is often omitted.
[10] In shorthand, brackets will be omitted.

These slices can be defined via a typing judgement $\Gamma \vdash e \Rightarrow \tau \dashv \mathcal{S}$, meaning $\mathcal{S}$ is the type-indexed synthesis slice of e^Γ synthesising τ. The non-indexed version can be retrieved by $\overline{\mathcal{S}}$. This judgement mimics the Hazel typing rules, while omitting portions of the program which are *type analysed* from the slice.

For example, when encountering application, the type of the function is synthesised purely by the function itself, so the argument can be omitted:

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \dashv \mathcal{S}_1 \qquad \tau_1 \blacktriangleright_\to \tau_2 \to \tau \qquad \mathcal{S}_1 \blacktriangleright_\to \mathcal{S}_2 \to \mathcal{S} \qquad \Gamma \vdash e_2 \Leftarrow \tau_2}{\Gamma \vdash e_1(e_2) \Rightarrow \tau \dashv \mathcal{S}}$$

Where function matching $\blacktriangleright_\to$ is extended to slices by eagerly applying the contexts paired with the slice sub-parts if applicable:

$$c \blacktriangleright_\to c \to c$$

$$c_1 * \mathcal{S}_1 \to c_2 * \mathcal{S}_2 \blacktriangleright_\to c_1 \circ \mathcal{S}_1 \to c_2 \circ \mathcal{S}_2$$

This approach correctly calculates the synthesis slices.

Conjecture 4 (Correctness) *If $\Gamma \vdash e \Rightarrow \tau$ then:*

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

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 way 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) is required to be consistent with `Int` due to the annotation. Therefore, the hole's analysis slice will 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 context slice was type synthesised, then the marked term would still be *required* to check against `Int`. However, the overall synthesised type may differ.[11]

[11] The previous example would synthesis `?` $\to$ `Int` as opposed to `Bool` $\to$ `Int`.

Checking Context

For this, we only want to consider the smallest context scope[12] that enforced the type checking. For example, the below term has 3 annotations, but only the inner one enforces the `Int` type on the integer 1:

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

I refer to this minimum context scope as the *checking context*, and checking contexts will always synthesise a type. Another way for types to be enforced is via application. The integer argument's type is enforced by the annotation on the function:

$$(\lambda\, x : \texttt{Int}.\ (\!|\,|\!)^u\,)(\,\underline{1}\,)$$

Definition 20 (Checking Context) *For term e checking against τ: $\Gamma \vdash e \Leftarrow \tau$. A checking context for e is an expression context $\mathcal{C}$ and typing assumption context $\mathcal{F}$ such that:*

- *$\mathcal{F}(\Gamma) \vdash \mathcal{C}\{e\} \Rightarrow \tau'$ for some τ'.*
- *The above derivation has a sub-derivation $\Gamma \vdash e \Leftarrow \tau$.*

Definition 21 (Minimally Scoped Checking Context) *For a derivation $\Gamma \vdash e \Leftarrow \tau$, it's minimally scoped expression checking context is a checking context of e such that no sub-context is also a checking context.*

All possible minimally scoped checking contexts can be constructed syntactically via rules. This coincides with the definition above, but is more amenable to performing proofs on. Given in the appendix **ref**.

If e' is a sub-term in a synthesising expression e, there will be a unique minimally scoped checking context $\mathcal{C}$ which matches the structure of the original term and typing assumption context $\mathcal{F}$ mapping the context used to type e' to the context used to synthesise e' in it's checking context.

Proposition 8 (Checking Context of a Sub-term in a Derivation) *If derivation $\Gamma \vdash e \Rightarrow \tau$ contains a analysis sub-derivation $\Gamma' \vdash e' \Leftarrow \tau'$ for sub-term e'. Then there is a unique minimally scoped context $\mathcal{C}$ for e' and typing assumption context $\mathcal{F}$ for Γ' such that:*

- *$\mathcal{C}\{e'\}$ is a sub-term of e, or is e itself.*
- *Derivation $\Gamma \vdash e \Rightarrow \tau$ contains a sub-derivation for $\mathcal{F}(\Gamma') \vdash \mathcal{C}\{e'\} \Rightarrow \tau''$, or is the derivation itself: $\tau'' = \tau, \mathcal{C}\{e'\} = e$, and $\mathcal{F}(\gamma') = \Gamma$.*

This checking context of a sub-term in a derivation can be calculated during type checking of the term. These checking contexts are what is actually useful for explaining type analysis in a *specific* program.

But, more generally, analysis slices are then a slice of any *minimally scoped checking context.*

Definition 22 (Analysis Slice) *For a term e analysing against τ: $\Gamma \vdash e \Leftarrow \tau$, and a checking context $\mathcal{C}^{\mathcal{F}}$ for e. An analysis slice is a slice, $c^{\prime}$, of $\mathcal{C}^{\mathcal{F}}$ which is also a checking context for e. That is:*

[12] The *size* of the context around the marked term.

- $f(\Gamma) \vdash c\{e\} \Rightarrow \tau'$ *for some* τ'.
- *The above derivation has a sub-derivation for* $\Gamma \vdash e \Leftarrow \tau$.

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

Conjecture 5 (Minimum Analysis Slices) *The minimum analysis slice analysing* e *in a checking context* $\mathcal{C}^{\mathcal{T}}$ *is an analysis slice* $\mathcal{p}$ *such that for any other any other analysis slice* $\mathcal{p}'$ *is at least as specific,* $\mathcal{p} \sqsubseteq \mathcal{p}'$. *This minimum always uniquely exists.*

This can again be represented by a judgement read as, e *which type checks against* τ *in checking context* $\mathcal{C}$ *has (type-indexed) analysis slice* $\mathcal{S}$:[13]

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

There are two situations which enforce *checking contexts*, of which application is most interesting, calculating the minimum indexed synthesis slice of the context and retrieving just the sub-slice relevant to typing the *argument*:[14]

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \dashv \mathcal{S}_1 \qquad \tau_1 \blacktriangleright_{\to} \tau_2 \to \tau \qquad \mathcal{S}_1 \blacktriangleright_{\to} \mathcal{S}_2 \to \mathcal{S} \qquad \Gamma \vdash e_2 \Leftarrow \tau_2}{\Gamma; \mathcal{C} \vdash e_2 \Leftarrow \tau_2 \dashv (c^f \mapsto c(\bigcirc)^f) \circ S_1}$$

Where $(c^f \mapsto c(\bigcirc)^f)$ maps the outer expression slices to context slices matching the input checking context. **(add this 'two-hole-context' to the type-indexed slice section, maybe make type-indexed expression slices explicit.**

This definition does indeed find the minimum analysis slice:

Conjecture 6 (Correctness) *If* $\mathcal{C}$ *is a checking context for* e *and* $\Gamma \vdash e \Leftarrow \tau$. *Then* $\Gamma; \mathcal{C} \vdash e \Leftarrow \tau \dashv \mathcal{S}$ *and* $\mathcal{S}$ *is the minimum analysis slice of* e.

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 there exists a type that if the sub-term changed it's type to, then the overall term would also change type, 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:

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

The terms which *contribute* to the *underlined* 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:

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

[13] Again, non-indexed version retrieved by $\overline{\mathcal{S}}$.

[14] Note that e_1 synthesises τ_2 exactly here. Contribution slices consider situations involving subsumption which may synthesise different but consistent types.

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*.[15] 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 ?.

Therefore, this criterion just *omits all dynamically annotated regions* of the program.[16] And, the contextual part of the slices are just *analysis slices*.

Definition 23 (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 $\varsigma \sqsubseteq e'$.*
- *Under restricted typing context γ, that ς checks against any τ_2' at least as precise as τ':*[17]

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

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.

The synthetic parts of these slices can be calculated in exactly the same way as synthesis slices, except now also considering the *subsumption* rule. The analytic parts are regular analysis slices.

The subsumption rule synthesises a type for some term, then checks consistency with the checked type. This is where the dynamic portions can be omitted, to give a contribution slice for the checked term. Representing contribution slices with the judgement $\Gamma \vdash e \Rightarrow \tau \dashv_C \mathcal{S}$ and $\Gamma; \mathcal{C} \vdash e \Leftarrow \tau \dashv_C \mathcal{S}_e \mid \mathcal{S}_c$, where $\mathcal{S}_e$ is the expression slice part and $\mathcal{S}_c$ is the contextual part:

$$\frac{\Gamma \vdash e \Rightarrow \tau \dashv_C \mathcal{S}_s \qquad \tau \sim \tau' \qquad \Gamma; \mathcal{C} \vdash e \Leftarrow \tau' \dashv \mathcal{S}_c}{\Gamma; \mathcal{C} \vdash e \Leftarrow \tau' \dashv_C \text{static}(\mathcal{S}_c, \mathcal{S}_s) \mid \mathcal{S}_c}$$

Where static is takes omits the *right* (synthetic) slice parts where *left* (analytic) slice is a leaf but the right is not. As the types are consistent, these leaves will be the unknown type. This means that portions of the synthetic part of the slice which match against ? will be omitted:
SHOW A DIAGRAM

$$\text{static}(p, p') = p'$$

$$\text{static}(p, \_ \to \_) = \square^{\emptyset}$$

$$\text{static}(c_1 * \mathcal{S}_1 \to C s_2 * \mathcal{S}_2,\ c_1' * \mathcal{S}_1' \to C s_2' * \mathcal{S}_2') = c_1 * \text{static}(\mathcal{S}_1, \mathcal{S}_1') \to c_2' * \text{static}(\mathcal{S}_2, \mathcal{S}_2')$$

[15] Note that this logic does *not* extend to globally inferred languages.

[16] But does not omit the dynamic annotations themselves.

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

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

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

TODO: UPDATE THIS FIGURE, move to appendix!!

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

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

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

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

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

Figure 3.1: Contribution Slices

3.1.7 Type-Indexed Slicing Context

It seems natural to extend the typing context to become a type-indexed slicing context. Then, when accessing typing assumptions via variable references, the slice information which derived this type can be revealed. But, as no context is propagated, the slices would have no .

3.2 Cast Slicing Theory

This section is not written yet, but will be very short... The only addition is cast dependence, which is literally just a partial order on casts through dynamics.

Fairly trivial, just treat tag types in casts with their slices and decompose accordingly.

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.

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[18], holes, operators, variables, let bindings, functions & closures, type functions, type aliases, pattern match expressions, casts, explicit fix[19] expression.
- **Types**: Unknown type[20], 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).

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

[19] Fixed point combinator for recursion.

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

- **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).

```
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

Directly storing expression slices directly as ASTs is both *space and time inefficient*. The AST type is a persistent data structures [67, 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.

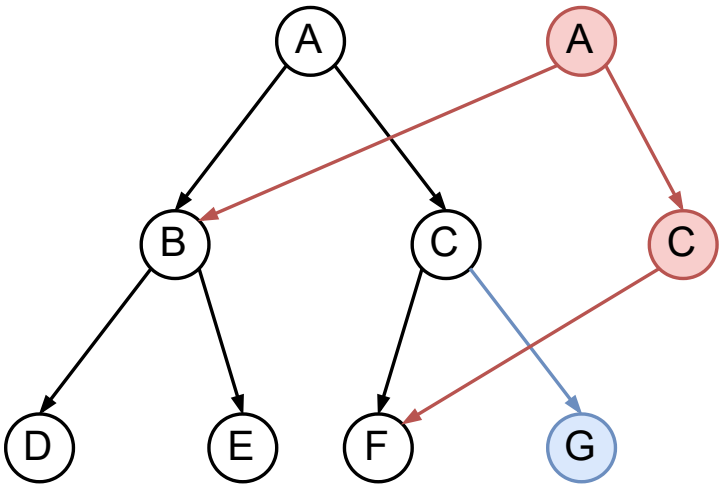


Figure 3.3: Persistent Tree

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 [59, 48].

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[21] and still require copying the path to the cursor as before.[22]

Produce Tikz Diagram Here.

Figure 3.4: Tree Zipper

NOTE: Actually the below might be inaccurate, I think I'm confusing myself with persistence. I think I could actually space-efficiently construct contextual and synthetic slices.

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

The benefits of unstructured slices being more flexible apply either way. Plus, I don't actually need the slice structure for highlighting terms. Plus ids take up less space by a constant factor anyway.

Unstructured Code Slices

With this in mind, given that the structure of expression slices does not actually matter for highlighting[23], 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.

[21] In the depth of the cursor.

[22] 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.

[23] 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

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 *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*[24] injection from types to typeslices by tagging with `Typ`. Note that this means the empty slices `TypSlice` no longer needs to include atomic types.[25]

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:[26] **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)
...
```

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

Polymorphic Variants [35, ch. 7.4], notated `[ | ... ]` are used here for convenience in incrementally writing functions, explained for an example *apply* function below. These are variants which exhibit *row polymorphism* [68] [11, ch. 10.8] where these variants are related by a *structural subtyping* relation [76] 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 [71, ch. 2] with respect to the subtyping relation. For example, id tagging is covariant[27], 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[28] (**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 **??**, 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.

[27] It is just a labelled pair type.

[28] 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[29] 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*[30] (`Syn`) or *analysing* (`Ana(Typ.t)`).

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

[30] 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 [36, ch. 13][31]. Hazel-specific details in [14].

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*[32] 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

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

[32] 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 3.1.7 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.[33]

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

Type slice of selected expression is shown. Show figures.

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.

[33] 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. B.8), there are two rules where new[34] casts are *inserted* ITGround, ITExpand (fig. B.10). The new types are both created via a *ground matching* relation (fig. B.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 [14].

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.[35] 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.

[34] As opposed to being derived from decomposition.

[35] 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 ...

Note: Very complex!!! I probably need to ask whoever made it what the terminology means/to give a simple explanation. The reader could skip this section, it is purely technical. In fact, given word count might be removed entirely, even though it is interesting technically.

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* [41]. When debugging static type errors, traces leading to a corresponding dynamic error are normally *unavailable*.[36] This section concerns the building blocks leading up to a search procedure that finds inputs which lead to dynamic errors *automatically*.

Seidel et al. [29] 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

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

within Hazel's runtime casts. My implementation extends Seidel's to consider more classes of expressions[37] and differs mechanically in many ways due to language differences and fundamental design differences.[38] 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[39], and allows search methods that avoid non-termination[40].

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[41] 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 create a monadic and tree-based high-level representation. With the tree model allowing for varying low level search traversals.

This sections describes an abstract DSL for this. It's *module type* of combinators is `Nondeterminism.Sea` Section 3.7.4 discusses the actual implementations of this interface, giving three different searching procedures.

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.

[37] Notably, sum types.
[38] Differences, and Hazel-specific challenges are noted throughout. **(ENSURE THIS!)**
[39] For countable types.
[40] For example, if one particular instantiation leads to an infinite loop in code.
[41] There may be multiple.

- `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 [4]. 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.

Abstracting Search Order: Tree Model

Typical stream-based models of non-determinism [79] 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 [65]? 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)[65].

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

Figure 3.16: Tree of Choices with Breadth-First Search

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

Figure 3.17: Forest Defined Using `wrap`

Figure 3.18: Unfair vs Fair Choice

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

Figure 3.19: Unfair vs Fair Conjunction

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

A combinator, `wrap`, allows for the programmer to control directly what constitutes a *fork* in the tree as suggested by Spivey [55]. 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*[42] 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`[43]. 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.

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?**

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

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

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.[44] 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)
```
[45]

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.

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

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

```
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:[46]

- 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.

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

[46] 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?

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:

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 4.8.5 for further discussion.

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

```
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

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 5.1.5) 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 4.8.5), 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.

An example step leading to a specific eval obj.

Figure 3.25: `EvalObj.t` Example

Therefore, I instead use the existing term ids to represent metavariables, and ensure these ids are propagated correctly throughout statics, elaboration, and evaluation.[47]

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.

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*. However, this is a significant change to the Hazel semantics, so was opted against. Section 4.8.4 discussed the consequence of this choice.

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

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.

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

Figure 3.27: Branch-Directed Instantiation

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 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:

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

Figure 3.26: Instantiating a Scrutinee

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

Figure 3.30: Not Instantiations

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?[48]

?? 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.

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

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

Show how streams & append represent depth first search

Figure 3.31: Depth First Search as Streams

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*.

Dynamic Match Statements: When matching dynamically on values with different types, the instantiations wrap the scrutinee in casts to each type. If any of these casts failed, they should not count as witnesses, as they were introduced entirely by the instantiation procedure.

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

Note: Should remove fair choice and conjunction, in favour of it just being an interleaved search method (see interleaved DFS below)

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.

Breadth First Search

Breadth first search needs to take specific account of the tree structure. Sequences of bags[49] can be used to represent solutions at each *level* of the tree [65]. Therefore, forcing each element

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

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 3.32: 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, $[b]$[50], 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 - can't actually work out 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 [28]. 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.

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)**.

[50] Using list notation also for sequences

Figure 3.33: Monadic BFS Requires Exponential Space

To represent iterative deepening [**SearchAlgebra**], we can use functions `int => list(('a, int))`, calculating every[51] 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

[51] 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 3.34: Associativity of Iterative DFS `bind`

Chapter 4

Evaluation

This section evaluates how successfully and effectively the implemented features achieve the goals stated in the introduction.

4.1 Success

This evaluation is more ambitious than that presented in the project proposal goals. As demonstrated below, the type witness search procedure and cast slicing features exceeds all core goals[1] presented in the project proposal, while type slicing had not been conceived, but naturally complements cast slicing. Some extension goals were reached: almost of Hazel was supported[2]. The search procedure was not mathematically formalised, but the slicing mechanisms were.

4.2 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.

[1] That is, reasonable coverage and performance.

[2] Except the type slicing of two constructs: type functions and labelled tuples (which was a new feature merged by the Hazel dev team in February).

4.3 Methodology

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

Quantitative Analysis

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

The time and space usage of the search procedure implementations were micro-benchmarked for each ill-typed program in the corpus. Up to 100 runs were taken per program with estimated time, major and minor heap allocations were estimated using an ordinary linear regression (OLS) to $r^2 > 0.95$[3] via the *bechamel* library [7].

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

The *coverage*, what proportion of programs admit a witness, for each search procedure implementation was measured. The search procedure does not always terminate, a 30s time limit was chosen. The coverage was expected to be *reasonable*, chosen at 75%.

Additionally, the *size* of witnesses, evaluation traces, type slices, and cast slices were measured. The intention being that a smaller size implies that there is less information for a user to parse, and hence easier to interpret.[4]

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*[5] are compared. Additionally, further unimplemented improvements are proposed.

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

Various program examples are given, demonstrating how all three features can be used together to debug a type error. Improvements to the UI are discussed.[6]

4.4 Hypotheses

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

[3] TODO: get values

[4] Not necessarily *always* true, but a reasonable assumption.

[5] Which were then implemented and analysed.

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

Search method space requirements: The space requirements for DFS and Bounded DFS are expected to be lower than that of BFS and interleaved DFS.

Type Slices are larger than Cast Slices: Casts are de-constructed during elaboration and evaluation, so cast slices are expected to be smaller than the original type slices, and therefore more directly explain why errors occur.

The Small Scope Hypothesis: This hypothesis [37] 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 [57] and answer-set programs [39]. Does it also hold for finding dynamic type errors from small *hole instantiations*?

Smaller instantiations correlate with smaller traces: As functional programs are often written recursively, destructuring compound data types on each step. If this and the small scope hypothesis hold, then most errors could be found with *small execution traces*.

4.5 Program Corpus Collection

A corpus of small and mostly ill-typed programs was produced, containing both dynamic (unannotated) programs and annotated programs (containing statically caught errors). We have made this corpus available on GitHub [8].

4.5.1 Methodology

There are no extensive existing corpora of Hazel programs, nor ill-typed Hazel programs. Therefore, we opted to transpile parts of an existing OCaml corpus collected by Seidel and Jhala [24]. Which is freely available under a Creative Commons CC0 licence.

I am grateful for my supervisor who created a best-effort OCaml to Hazel transpiler [9]. This translates the OCaml examples into both a dynamic example, and a (possibly partially) statically typed version according to what type the OCaml type checker expects expression to be.[7]

This corpus contains both OCaml unification and constructor errors. When translated to Hazel, these may manifest as differing errors. The only errors that the search procedure is expected to detect are those which contain *inconsistent expectations* errors. Hence, the search procedure is ran on the corpus of annotated programs filtering those without this class of errors. Additionally, the search procedure requires the erroneous functions to have holes applied to start the search, these are inserted automatically by the evaluation code after type checking the programs.

4.5.2 Statistics

The program corpus contains **698** programs of which **203** were applicable to performing the search procedure on. Averages and standard deviations in size and trace size[8] shown in fig. 4.1.

[7] The annotations may be consistent, as we are translating ill-typed code.

[8] When using normal, deterministic, evaluation.

	Count	Prog. Size		Trace Length	
		Avg.	Std. dev.	Avg.	Std. dev.
Unannotated	404	117	81	9	9
Annotated	294	117	76	9	9
Searched	203	120	77	10	10
(Total)	698	117	79	9	9

Figure 4.1: Hazel Program Corpus

	Averages	Implementations			
	unit	`DFS`	`BDFS`	`IDFS`	`BFS`
Time	ms	7.6	73	140	120
Major Heap	mB	3.7	32	5.9	25
Minor Heap	mB	66	680	1900	1300

Figure 4.2: Benchmarks: Search Implementations

4.6 Performance Analysis

4.6.1 Slicing

The type and cast slicing mechanisms don't increase the time complexity of the type checker nor evaluator. Hence, they are still as performant as the original, and can still be used interactively in the web browser up to medium sized programs.

4.6.2 Search Procedure

Only the annotated ill-typed corpus containing inconsistency errors are used in 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 results are found under a 30s time limit. Each search implementation gives terminates on a different set of programs with potentially different witnesses and traces. The succeeding *effectiveness* analysis considers the search procedures results under these constraints.

However, by micro-benchmarking only the programs which do not time-out, the time and space used searching for each witness can be estimated. The averages over all successful programs for each implementation are given in fig. 4.2. These results suggest that, as expected, BFS and interleaved DFS (IDFS) use more memory in total[9] than bounded DFS (BDFS) and DFS, while DFS is the fastest[10].

This is not a fair test as the sets of programs averaged over are different, for example, BFS only succeeds on the smaller programs. Hence, the ratios between each implementation as compared to DFS on only the programs which *both* succeed on are given in fig. 4.3. Under this, the results are even more pronounced.

[9] Although, IDFS efficiently keeps most memory allocated short lived, being in the *minor* heap.
[10] Having the least overhead.

Ratios vs. DFS	Implementations BDFS	IDFS	BFS
Time	8.3	52	230
Major Heap	9.0	3.2	270
Minor Heap	9.7	83	390

Figure 4.3: Benchmarks: Performance ratios to DFS over common programs

4.7 Effectiveness Analysis

4.7.1 Slicing

Type slice sizes were calculated by the type checker over the entire corpus, where the size is the number of constructs highlighted. While *cast slice* sizes were calculated over in the elaborated expressions after type checking.

Figure 4.4 shows that both type and cast slices are generally small. In particular, the proportion of the context[11] highlighted is very low, generally less than 5% for dynamic code and 10% for annotated code. Therefore, they concisely explain the types. However, some slices are still large, as seen by the relatively large standard deviations.

Additionally, for errors, there will be multiple inconsistent slices involved. Section 4.8.1 describes how these slices can be summarised to only report the inconsistent parts. We find that these *minimised* error slices are significantly (3x) smaller than directly *combining* the slices.

Averages			**Subdivisions**					
			Combined Error Slices			**Minimised Error Slices**		
	unit	ok	expects	branches	all	expects	branches	all
Type Slice	size	8.2	13	22	15	5.7	3.2	5
Std. dev.		11	10	24	15	4.31	4.1	4.4
Proportion	%	5	8	14	9	3	2	3
Std. dev.		7	8	14	10	3	3	3
(Unannotated)								
Type Slice	size	7.5	21	133*	22.6	8.2	2.0*	8.2
Std. dev.		13	22	42*	25.2	12.6	0.0*	12.6
Proportion	%	4	14	0.48*	15	6	1*	5
Std. dev.		9	18	0.07*	18	9	0*	12
(Annotated)								

* only 2 annotated programs had inconsistent branches

Figure 4.4: Effectiveness: Type Slices

Casts have a pair of slices, *'from'* a type and *'to'* a type, both of which are smaller on average (fig. 4.5) than type slices. This is because casts are split between ground types, i.e. to a 'dynamic function' type, resulting parts of the type information being spread out over a number of casts; therefore, casts are can more precisely point to which part of an expressions type caused them.

[11] Calculated by close approximation by the *program size*. As each program in the corpus is just one definition. Calculating the context itself is non-trivial.

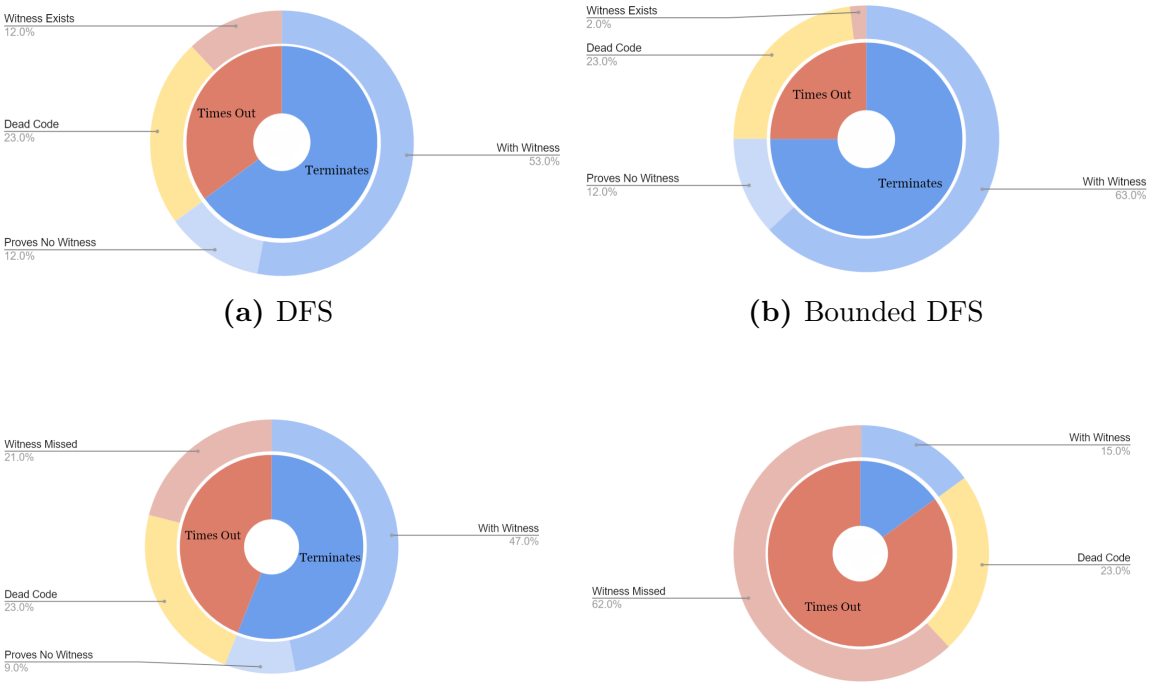
Unnanotated code has larger *'from'* slices as it relies more on synthesis for typing code, and vice versa for annotated code.

Averages		**Subdivisions**			
		Ok		**Errors**	
	unit	to	from	to	from
Cast Slice	size	5.5	1.2	5.9	1.5
Std. dev.		8.1	3.7	7.1	2.0
Proportion	%	1	0.2	1	0.2
Std. dev.		1	0.5	1	0.4
(Unannotated)					
Type Slice	size	4.8	6.3	6.9	4.4
Std. dev.		11	13	9.1	9.3
Proportion	%	1	2	2	1
Std. dev.		1	1	1	1
(Annotated)					

Figure 4.5: Effectiveness: Cast Slices

4.7.2 Search Procedure

Witness Coverage



(a) DFS **(b)** Bounded DFS

(c) Interleaved DFS **(d)** BFS

Figure 4.6: Search Procedure Coverage

The search procedure terminates either with a witness or proving no witness exists. A majority of programs terminated when using BDFS, DFS and IDFS, with BDFS meeting the

75% target directly (fig. 4.6). IDFS and BFS perform relatively poorly likely due to excessive memory usage (fig. 4.2).

However, not all programs actually have an execution path leading to a dynamic type error. For example, errors within dead code cannot be found. I manually classified each failed program to check if a witness does exist, but was not found, or no witness exists. Of which, 89% was found to be dead code and 5% due to Hazel bugs[12], the bugs being excluded from the results. This gives only 2% cases where BDFS failed to find an *existing* witness; DFS and IDFS also meet the goal in failing in less than 25% of cases.

Section 4.8.4 goes into further detail on categorising the programs which time out, and how this could be avoided.

Witness & Trace Size

As predicted by the small-scope hypothesis, most programs admitted *small* witnesses. And the depth first searches produce longer and more varied trace lengths, which allowed them, in combination with their lower memory overhead, to attain higher coverage (many witnesses were at a deeper depth than IDFS or BFS were able to reach).

However, there was no correlation (Pearson correlation [**PearsonCorrelation**]) between witness sizes and trace sizes, even when normalised by the original deterministic evaluation trace lengths. This is likely because most errors are in the base cases, so few large witnesses are even found, with the noise from trace lengths to different programs' base cases dominating.

	DFS	BDFS	BFS	IDFS
Witness Size Avg.	1.1	1.9	1.4	2
Std. dev.	1.2	2.3	1.4	2.3
Trace size Avg.	33	32	11	17
Std. dev.	35	33	2.4	5

Figure 4.7: Witness & Trace Sizes

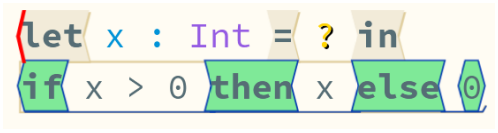
4.8 Critical Analysis

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.

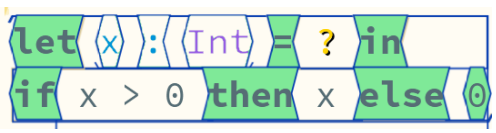
4.8.1 Slicing

Contribution slicing produces large slices, highlight all branches, with much of the information is not particularly *useful*, with much of it explaining why the use of *subsumption* in subterms was valid, which is never directly related to type errors. As such, I opted to use and evaluate synthesis and analysis slices for the task of explaining static errors more concisely. If a user were unsure as to why a sub-term is consistent with it's expected type, they can still select it to retrieve the slices.

[12]Being also present on the main branch, not just in my code.



(a) Ad-hoc Slice: Let expression omitted



(b) Ordinary Slice

Figure 4.8: An Ad-hoc Slice vs. Ordinary Slice

The type slicing theory (section 3.1) required that highlighted parts must form valid expressions or contexts.[13] But, some of the constructs highlighted are purely structural, not influencing the type, which could be omitted in practice. For example, bindings when the bound variable is dynamic or unused when typing the particular expression considered. This was a primary motivation in implementing slices via *unstructured slices* (**??**). I call these 'ad-hoc' slices.

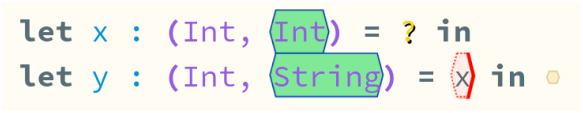
Figure 4.8 binds a integer `x`, but this variable is only used in one branch, where the type information for the `if` is already known from the other branch, so not included in the slice (highlighted in green). So, even though we selected[14] the whole program, the let expression is omitted from the slice. Further, a contribution slice would be even more verbose here, highlighting *everything*.[15]

Error Slices

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 multiple inconsistent branches (whom are *all* synthesising). Meaning understanding the error requires looking at each of these.

However, some portions of the type may be consistent, not causing to the error. A form of type joining which extracts only the *inconsistent parts* was therefore implemented. This became the default UI when clicking on a type error, see how branches synthesising or expecting inconsistent types `(Int, Int)` vs `(Int, String)` only highlights the right element in fig. 4.9.

(a) Partially Inconsistent Branches

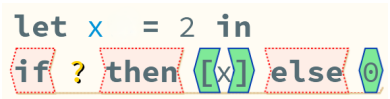


(b) Partially Inconsistent Expectations

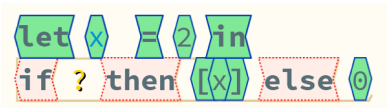
Figure 4.9: Error Slices

Additionally, when the inconsistent parts are compound types, they must differ at their *outermost* constructor (i.e. a `List` vs an `Int`). This can be considered the *primary* cause of the error, inconsistencies deeper within the type were not the ones causing the error. These minimised slices (fig. 4.10) were significantly smaller than naively combining the full slices (by 3x, see **??**); there would be an even larger ratio for more complex programs.[16]

[13] Allows useful mathematical properties to be formalised.
[14] See the red cursor & let expression filled in a darker colour.
[15] Both branches must be highlighted, and the conditional is only well-typed if `x` checks against an integer.
[16] The corpus has few *partially* inconsistent errors.



(a) Minimised Error Slice: Inner `Int` slice within list is omitted



(b) Ordinary Error Slice

Figure 4.10: Minimised Error Slices

4.8.2 Structure Editing

Hazel uses a structure editor with an update calculus [**HazelStructureCalculus**]. Statics are recalculated upon edits and even *cursor movements*. While type checking with slices is still very fast, it is not so suitable for such a rapid use case in larger programs.

Parsing and structural edit actions are made efficient[17] by use of a zipper data structure [**HuetZipper**, 59]. 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. These non-local updates would be relatively rare. This is very complex and would require an entire rewrite of the Hazel statics, but would be beneficial for Hazel in general, not just type slicing.

4.8.3 Static-Dynamic Error Correspondence

sec:ErrorCorrespondence Section 3.7.3 and 3.2.4 show that a static error that elaborates a cast failure, which can be associated with a dynamic error whose cast error is dependent on this original failed cast. This works well for *inconsistent expectations* static type errors.

However, *inconsistent branches* errors, static errors caused by synthesised branches being inconsistent, do not directly cause cast errors. Such errors will only be detected if both branches are used within a static context during the same evaluation run. The search procedure might find such cases if they exist, but cannot so easily associate it back to the static error. However, elaborating these terms does insert casts: on each branch to the dynamic type; these could be associated together with the error and tracked accordingly.

4.8.4 Categorising Programs Lacking Type Error Witnesses

47 programs which timed out under the BDFS search procedure were manually inspected and classified as either:

- Witness Exists: BDFS failed to find an existing witness.
- Dead Code: The error was in unreachable code by any type consistent instantiations. These then subclassified into:

[17] Constant time.

- Pattern Cast Failure: An error was within a pattern matching branch. This additionally makes the branch unreachable. These cast failures would be found by an extended pattern directed instantiation algorithm (section 3.6.6).[18]
- Unbound Constructor: A pattern matching branch matching an unbound constructor. These unbound errors are also detectable with extended pattern directed instantiation.
- Wildcards: Error within trivially dead code bound to the inaccessible wildcard pattern: `let _ = ... in ...`.
- Non-Trivial: Dead code which is less easily detectable. There was only one example which was due to infinite recursion for all inputs.

- Hazel Bugs: Evaluation errors encountered stemming from *unboxing bugs* in the main branch of Hazel. These were excluded from the statistics.

Figure 4.11 shows this distribution and three (paraphrased) examples are given in fig. 4.12. The full classification is in `failure-classification.txt`.

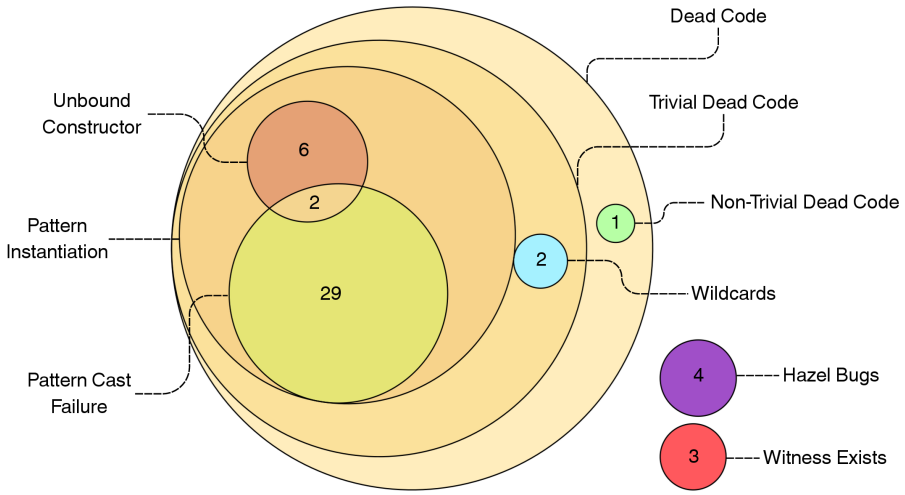


Figure 4.11: Distribution of Failed Program Classes

Non-Termination, Unfairness, and Search Order

DFS is fast, but has significant insurmountable issues with non-termination, resulting in it having worse coverage than BDFS (fig. 4.6). Any time evaluation goes into an infinite loop, no more instantiations can be explored. Similarly, DFS will never backtrack to try previous instantiations, for example instantiating a pair of integers `(?,?)` would only try `(0,?)`, `(0, 0)`, `(0, 1)`, `(0, -1)`, `(0, 2)` etc. but never `(1, 0)` for example (fig. 4.13). This was a common issue, occurring in 10% of the search corpus. Bounded DFS, BFS, and interleaved DFS, were implemented in response to this. With BDFS performing best due to preference of evaluation over instantiation (and it's lesser memory footprint: fig. 4.2).

[18] Where inconsistent patterns would be attempted, and subsequently reduced to concrete cast failures in expressions.

```
type expr =
  + VarX
  + Sine(expr)
  + Cosine(expr)
  + Average(expr, expr)
in let exprToString : forall a -> expr -> [a] = typfun a -> fun e -> case e
  | VarX => []
  | Sine(e1) => exprToString@<a>(e1)
  | Cosine(e1) => exprToString@<a>(e1)
  | Average(e1, e2) => exprToString@<a>(e1) ++ exprToString@<a>(e2)
end in ?
```

Depth-first bias caused the procedure to try mostly permutations of `Sine(...)` and `Cosine(...)`. The error was on the `Average(...)` branch, not found within the time limit.

(a) Witness Exists: prog2270.typed.hazel

```
type expr =
  + VarX
  + Times(expr, expr)
in let exprToString : expr -> String = fun e -> case e
  | VarX => "x"
  | (Times(e1), e2) => exprToString(e1) ++ " * " ++ exprToString(e2)
end in ?
```

A tuple pattern is used when an `expr` is expected. Instantiation only tries value of type `expr`. Further, another error exists inside this inaccessible branch.

(b) Dead Code – Wildcard: prog0080.typed.hazel

```
type expr =
  + VarX
  + Sine(expr)
  + Average(expr, expr)
  + MyExpr(expr, expr, expr, expr)
in let exprToString : expr -> String = fun e -> case e
  | VarX => "x"
  | Sine(m) => "sin(pi*" ++ exprToString(m) ++ ")"
  | Average(m, n) =>
  "((" ++ exprToString(m) ++ "+" ++ exprToString(n) ++ ")/2)"
  | MyExpr(m, n, o, p) => ?
end in let _ = exprToString(MyExpr((VarX, ?, VarX))) in ?
```

Product arity inconsistency is present in inaccessible code bound to the wildcard pattern.

(c) Dead Code – Pattern Cast Failure: prog0339.typed.hazel

Figure 4.12: (Paraphrased) Failure Examples

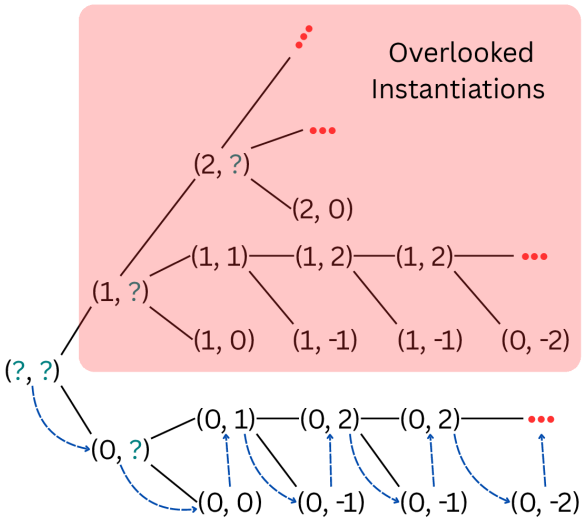


Figure 4.13: Non-exhaustive Instantiations in DFS

BFS and IDFS would do significantly better if the search algorithm was tweaked to less frequently wrap evaluation steps deeper in the search tree, effectively reducing the *cost* of evaluation. For example, it could wrap evaluation in one extra search depth only once every 10 steps, allowing 10 steps of evaluation to be explored for every instantiation. This is a delicate balancing act.

Dead Code & Nested Errors

Dead code caused most time outs for the search procedure using BDFS. Errors within dead code cannot have a witness as they are not dynamically reachable. Therefore, dead code analysis will reduce timeouts.

Additionally, code can become dead due to errors. For example, 12 (32%) of the dead code failed searches also had a nested error within a branch which was unreachable due to an error[19] within the branch pattern. Even if we find witnesses for these branch errors, the nested errors will still be unreachable.

Dynamically Safe Code

Some code with static errors is inherently safe dynamically, the typical example being: `if true then 0 else "str"`. These have *no* dynamic witness, and the search procedure can often prove this by running out of possible instantiations: BDFS proved no witness 12% of the time.

However, in general, the procedure may repeatedly instantiate infinitely many witnesses in dynamically safe code, and thereby not terminating. Therefore, search procedure timing out does not necessarily mean that a witness has not been found or that a witness does not exist.

Cast Laziness

Hazel treats casts lazily, only pushing casts on compound data to their elements upon usage. The instantiation procedure is therefore unable to instantiate parts of compound data until it

[19]Unbound constructor or inconsistent expectations.

is de-structured: for example, extracting the first element of a tuple. Further, Hazel does not detect cast errors between non-ground compound types (e.g. `[Int]` and `[String]`). Therefore, type inconsistencies between such types will not be found by the search procedure. These errors are less common, in the search corpus there were *no* programs exhibiting this.

Fixing this requires reworking the Hazel semantics to be based around eager casts. Dynamic type systems with this cast semantics have been explored [69] and applied also to gradual type systems [44].[20] Alternatively, instantiation could be performed less lazily, using more sophisticated methods to determine a holes type (accumulating information over *multiple* casts). This would significantly expand the search space, and still require some form of eager cast failure checking.

The eager cast semantics would be able to detect more dynamic errors at an earlier point in execution. For example, fig. 4.14 shows an expression which evaluates to casts between inconsistent types, yet is not caught as a dynamic error yet. The error is only found when extracting the second element.

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

```
≡ (1, "str") : (○, ○) : (Int, Int)
```

Figure 4.14: Inconsistent Lazy Casts: No Cast Failure

Combinatorial Explosion

When multiple holes are involved when searching, the search space increases exponentially.

Combinatorial explosion clearly has a particularly big affect on IDFS and BFS, who prioritise trying different instantiations over evaluating. The space usage then becomes a big bottleneck on speed, causing the witnesses (which have been instantiated by now) to not actually evaluate far enough to detect the error.

Further, some errors will only occur for very specific inputs, for example only for `(23, 31)` and `(31, 23)` in fig. 4.15. Directing instantiation to maximise code coverage earlier would result in these errors to be found attempting fewer instantiations.

[20] These papers refer to eager casts as *coercions*.

```
let f = fun (x, y) ->
  if x * y == 713 then 1 + "str" else 0
in ○
```

Figure 4.15: Witness Requiring Very Specific Instantiations

4.8.5 Improving Code Coverage

Of the discussed classes of programs lacking a type error witness (section 4.8.4), only 3 actually had concrete witnesses that were not found within the time limit. These occurred as the instantiations generated did not happen to take the branches involving the error.

These errors often require rather specific, 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).[21]

Intelligently directing hole instantiations to better cover the code would help here. Symbolic execution and program test generation is a well-researched area with numerous dedicated constraint solvers [**CITE MANY HERE**] and translations to theories for SMT solvers [**CITE**].

One form of this, purely structural pattern-directed instantiation, discussed in section 3.6.6, would have discovered 2 of the 3 missed witnesses by BDFS. 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 [21].

However, even without this, the BDFS search procedure still retains a very high coverage over the search procedure (97%) when excluding dead code.

4.9 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 localisation [12] to debug the errors.

4.9.1 Interaction with Existing Hazel Type Error Localisation

Hazel has three types of errors which can be addressed to varying extent by this project:[22] *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.

?? showed how dynamic witnesses can be associated to both classes of inconsistency errors, with inconsistent branch errors being more difficult to detect in general. Generating examples for inexhaustive match statements was not a core aim of this project, but indeterminate evaluation can do this automatically (see the `inexhaustive-matches` branch **TODO**). Faster, static generation of counter-examples to match statements is already a (mostly)[23] solved problem in static languages [51], and is under development for Hazel.[24]

The situations where a programmer does not understand why an error occurs generally arise due to a *misunderstanding* about the types of the code. In which case, existing error localisation is not so useful as the location of an error is determined while making *differing assumptions*

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

[22] Other errors not addressed being being: syntax errors, unbound names, duplicate & malformed labels, redundant pattern matches etc.

[23] Data abstraction can cause issues: for example Views [78] and Active Patterns [53]. Hazel is implementing modules, but yet to consider pattern matching on abstract data.

[24] Indeterminate evaluation is very inefficient compared to pattern matrices. These matrices would also be useful in directing hole instantiation (section 4.8.5).

about the types than the programmer. Type slicing, along with the context inspector (which shows the type, term, and describes the error of a selected expression: fig. 4.16), can help the programmer understand which assumptions the type system is making and *why* it is.

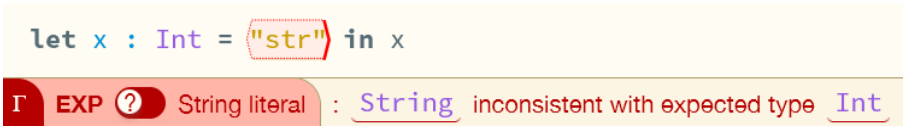


Figure 4.16: Selected Static Error described by Hazel Context Inspector

On the other hand, when the programmer and system agree on the types of a program, bidirectional typing generally localises the error(s) correctly and intuitively [32, 12]. However, Hazel has explored introducing global inference, where errors involve wider interaction between multiple regions of code, making error localisation (even when the type checker and user agree on the types) more difficult [25]. Further, deviations between the systems and the programmer mental model become more common with fewer annotations. Dynamic witnesses can help here [29], and forms of slicing supporting global inference is an interesting further direction.[25]

Finally, there may be errors in *dynamic regions* of the code. These are not found statically. The search procedure can test dynamic code for such type errors automatically. Indeterminate evaluation could also, in future, be used to perform more general property testing of code, in the sense of SmallCheck [50]. Adding annotations to find type errors in dynamic code is time-consuming, large numbers of annotations are required to make the code static enough to detect errors. In this case, it can make sense to use the search procedure to find errors more quickly. For example, fig. 4.17 shows a relatively annotated map functionwhich still does not provide enough type information to detect the error statically, that concatenation (`@`) operator is mistakenly used instead of cons (`::`).

```
let map : (? -> ?) -> [?] -> [?] =
  fun f -> fun l -> case l
    | [] => []
    | x::xs => f(x) @ map(f)(xs)
  end in ?
```

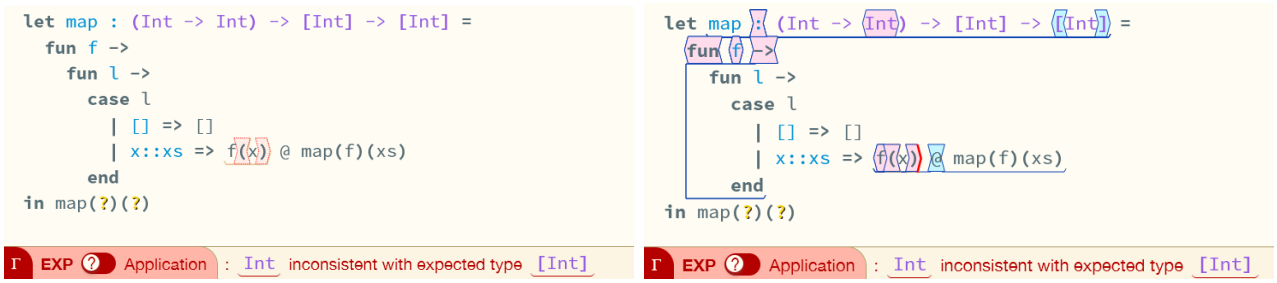
Figure 4.17: Partially Annotated Program misses Error

4.9.2 Examples

Returning to the map example, when fully annotated a static error can located at `f(x)` which has type `Int` but expects `[Int]`. The error slice shows (in pink) why `Int` is synthesised (due to input `f` being annotated `Int -> Int` and applied) and (in blue) why a list[26] is expected (being an argument of list concatenation). See fig. 4.18.

[25] Global constraint-based inference would require some form of additional constraint slicing. Similar ideas have been explored in error slicing [60, 56].

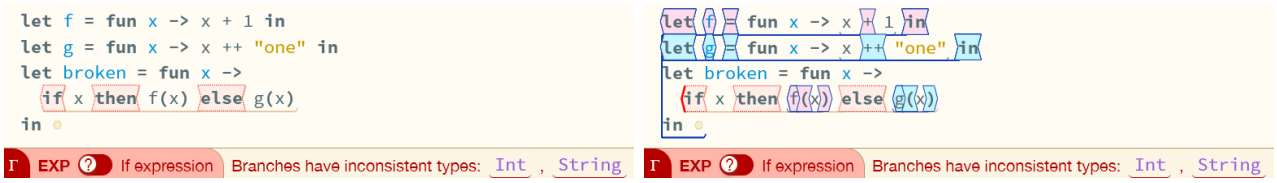
[26] A full analysis slice would also include the full `[Int]` annotation. But the inconsistency arises from the list part.



(a) Error **(b)** With Error Slice

Figure 4.18: Example: Type Slice of Inconsistent Expectations

Similarly, error slicing works for inconsistent branches: fig. 4.19 demonstrates a somewhat more complex inconsistency involving non-local bindings. With minimal error slices highlighting only the bindings, application, and the addition `+` and string concatenation `++` operators.



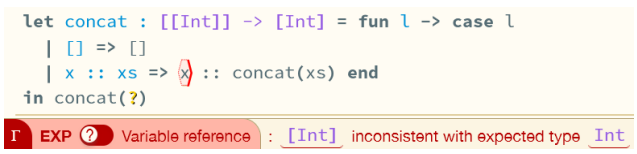
(a) Error **(b)** With Error Slice

Figure 4.19: Example: Type Slice of Inconsistent Branches

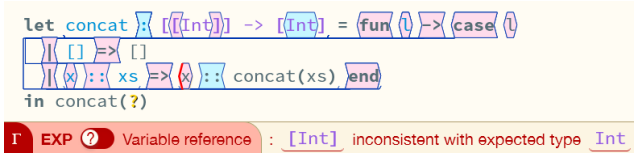
However, not every static type error slice is concise or obvious, especially if the user is still misunderstanding the types of operators involved. I demonstrate this by returning to the example from the introduction (fig. 4.20) where the user is mistaking list cons `::` for list concatenation `@`. The type slice here does highlight the `::` and annotation as enforcing `x` to be an `Int`, but the user would not understand why if they still think expect `::` to perform concatenation. Additionally, there is considerable noise[27] distracting from this. In comparison, the type witnesses demonstrate concretely how the program goes wrong: that the lists are not being concatenated. The user can cycle through increasingly larger witnesses[28] (the 2nd fully determinate witness given in the figure). Finally, the cast slice to `Int` concisely retrieves relevant part of the original type slice.

[27] A large synthesis slice part explaining why `x` is a list

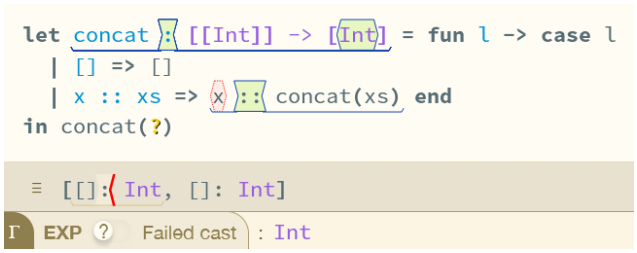
[28] Looking at larger witnesses can be useful to spot pattern with what is happening overall, i.e. that the lists are being consed, not concatenated.



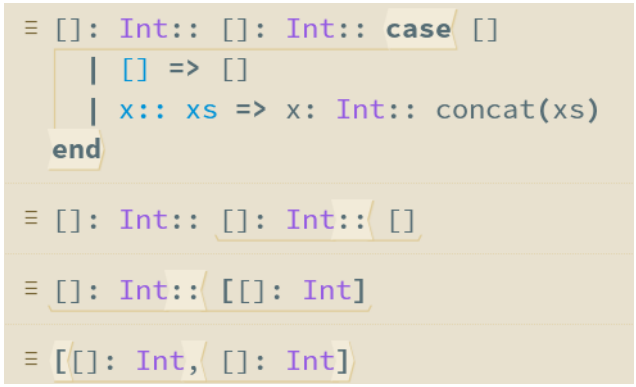
(a) Error: Mistaken Cons for Concat



(b) Verbose Type Slice



(c) A Witness (`[[], []]`) leading to concise Cast Slice



(d) Last 3 steps in witness trace: Shows `::` being List Cons

Figure 4.20: Example: Witnesses and Cast Slice more Understandable than Type Slice

Another, more subtle, example of this is confusing curried and non-curried functions: fig. 4.21 implements a `fold` function taking curried functions, but tries to apply an uncurried `add` function to sum int lists. This example shows how a single error can result in *multiple* cast errors, each having a more concise slice explaining them. See how the type slice had two inconsistencies (in both the argument and return type of add), which were then separated into two different cast errors: `add` expected it's argument to be a tuple, `fold` expected the result of `add` to be a function.

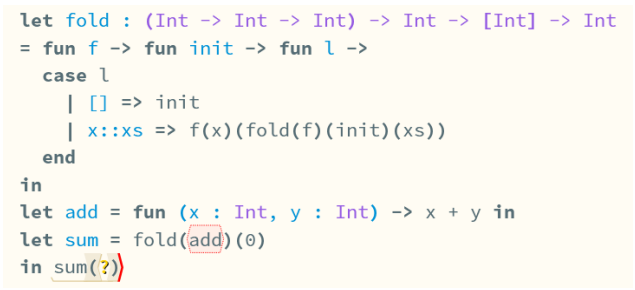
Finally, when code is dynamic, errors might not be statically found, and type slices have less information to work with. Figure 4.22 considers finding a simple error in a dynamic function. Cast slicing still works even without type annotations, allowing errors to blame regions of the source code.

4.9.3 Usability Improvements

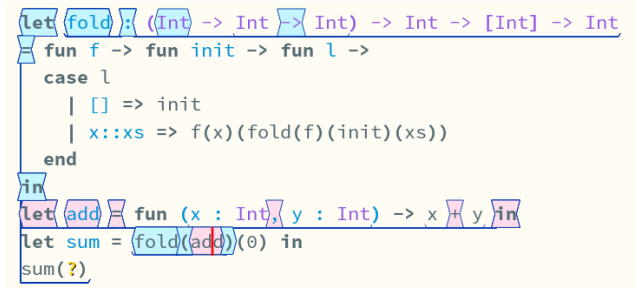
As designing an intuitive user interface was not a core goal, therefore actual use of the features can be quite awkward or unintuitive. Below, I propose various improvements, for which the architecture, of both Hazel and the newly implemented features, is sufficiently abstract and flexible to easily support:[29]

- Displaying type slices only *upon request* using the Hazel context inspector, which displays the *analysing* and *synthesising* types of the selected expression, see fig. 4.16.
- 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, or select the function arrow to get the part that makes it a *function* (ignoring the argument

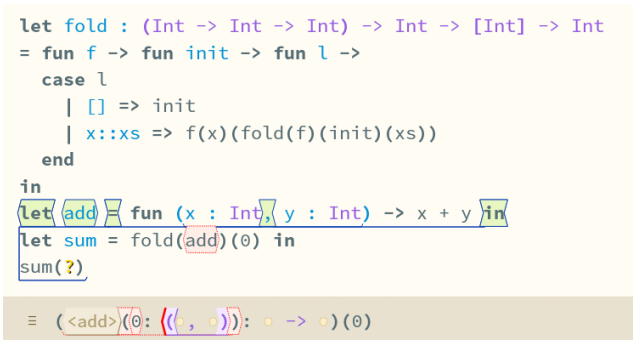
[29] Some of which were detailed in the Implementation chapter.



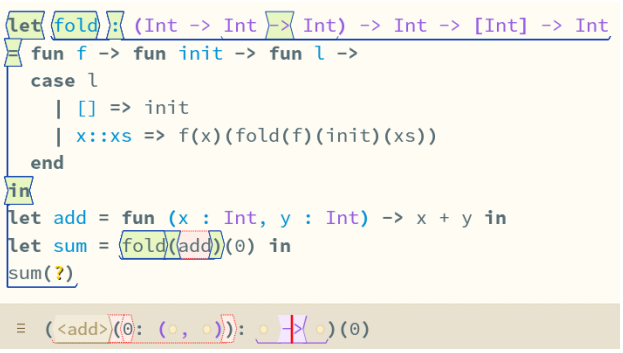
(a) Confused uncurried for curried `add`



(b) Somewhat Verbose Type Slice

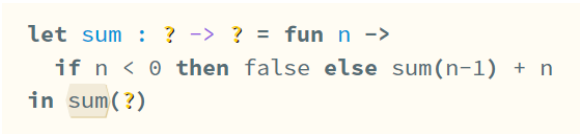


(c) Witness (`[0]`) expects input to `add` to be a tuple

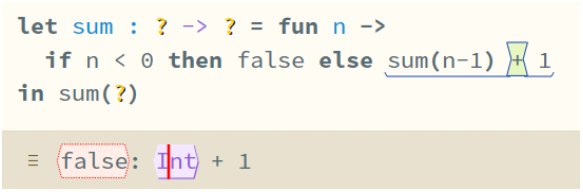


(d) Witness (`[0]`) expects output of `add(0)` to be a function

Figure 4.21: Example: Subtle Currying Error



(a) Error



(b) Automated witness found with cast slice blaming `+` operator

Figure 4.22: Example: Searching for Error in Dynamic Code

and return slice parts). The UI for this could again be via clicking on the type within the context inspector.

This interactive version could really help a user to understand how parts of code come together to produce their types. This is a powerful feature made possible by type-indexed slices.

- Graphs visualising cast dependence (section 3.2.4). Showing the *execution* context that leads to a cast error; these would be more concise summaries than full evaluation traces.
- UI for visualising the search procedure's execution traces and instantiations. Currently, the search procedure only gives the final result containing cast error. This can be integrated with Hazel's existing trace visualiser. Further, compression of traces to make them more readable, e.g. skipping over irrelevantIrrelevant to the error. function calls.
- Key bindings to more quickly cycle through indeterminate evaluation instantiation paths.

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 4.8 & 4.9). Also, further interesting applications of the features are considered.

5.1.1 UI Improvements, User Studies

The current UI is relatively simple. Section 4.9.3 suggested various improvements to the usability of all three features.

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. [29] 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.

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

[2] Hiding the execution of functions, except for when the error occurs.

5.1.2 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 [77, 40] which take parts of the program relevant to producing a *given* result.

5.1.3 Proofs

Giving proofs of the search procedure and slicing theories was an unmet *extension* goal. However, *much thought*[3] was given to ensuring that the theory is still sensible, and correct.

5.1.4 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 [50]; the approach being justified by the *small scope hypothesis* (section 4.4).

Other property testing models could be implemented, for example QuickCheck [62] 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?*.

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

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 [43] like *Curry* [6]. Adding unification [75] to turn these into full-blown logic variables would allow full logic programming in Hazel. A needed-narrowing evaluation strategy [61] 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*. An incomplete Hazel program could be used as a specification for it's incomplete parts, where executing the program would synthesise these. However, this approach has been found to be inefficient due to the need to solve *every* possible program, and prone to underspecified

[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.

problems, among other drawbacks [73]. Most naturally written Hazel programs would be underspecified.

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 [23, 45]. Therefore, an incomplete Hazel (logic) program sketch could still be used as a partial specification, but using more modern methods.

5.1.6 Symbolic Execution

The search procedure was not always able to find witnesses, with some branches containing the errors never being searched. This problem has been extensively researched for automated test generation and program verification, often using symbolic execution [21]. Constraints along execution paths can be modelled via *satisfiability modulo theories* (SMTs) [42], for which there exist many efficient solvers [47].

The evaluation of holes can already be consider a form of simple 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.

5.1.7 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 [66]). 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 [30] Miyazaki et al. [19], the latter of which would integrate most smoothly with Hazel and indeterminate evaluation.

Hazel currently has a branch exploring global inference `(thi)`, 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 [56].

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] Oleg Kiselyov. *Delimited Control in OCaml*. URL: https://okmij.org/ftp/continuations/implementations.html.

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

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

[6] *The Curry Programming Language*. URL: https://curry-lang.org/ (visited on 04/16/2025).

[7] *Bechamel Micro Benchmarking Library*. 2025. URL: https://github.com/mirage/bechamel.

[8] Patrick Ferris. *A Corpus of annotated and unannotated ill-typed Hazel Programs*. May 2025. URL: https://github.com/patricoferris/hazel-corpus.

[9] Patrick Ferris. *OCaml to Hazel transpiler*. May 2025. URL: https://github.com/patricoferris/hazel_of_ocaml.

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

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

[12] 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.

[13] David S. Warren. "Introduction to Prolog". In: *Prolog: The Next 50 Years*. Ed. by David S. Warren et al. Cham: Springer Nature Switzerland, 2023, pp. 3–19. ISBN: 978-3-031-35254-6.
DOI: 10.1007/978-3-031-35254-6_1. URL: https://doi.org/10.1007/978-3-031-35254-6_1.

[14] 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.

[15] 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.

[16] Hannah Potter and Cyrus Omar. "Hazel tutor: Guiding novices through type-driven development strategies". In: *Human Aspects of Types and Reasoning Assistants (HATRA)* (2020).

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] Roberto Baldoni et al. "A Survey of Symbolic Execution Techniques". In: *ACM Comput. Surv.* 51.3 (May 2018). ISSN: 0360-0300.
DOI: 10.1145/3182657. URL: https://doi.org/10.1145/3182657.

[22] 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.

[23] 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.

[24] Eric L Seidel and Ranjit Jhala. *A Collection of Novice Interactions with the OCaml Top-Level System*. June 2017.
DOI: 10.5281/zenodo.806814. URL: https://doi.org/10.5281/zenodo.806814.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

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

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

[37] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2012. ISBN: 0262017156.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] David Herman, Aaron Tomb, and Cormac Flanagan. “Space-efficient gradual typing”. en. In: *High.-order Symb. Comput.* 23.2 (June 2010), pp. 167–189.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] LUC MARANGET. “Warnings for pattern matching”. In: *Journal of Functional Programming* 17.3 (2007), pp. 387–421. DOI: 10.1017/S0956796807006223.

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

[53] Don Syme, Gregory Neverov, and James Margetson. "Extensible pattern matching via a lightweight language extension". In: *SIGPLAN Not.* 42.9 (Oct. 2007), pp. 29–40. ISSN: 0362-1340.
DOI: 10.1145/1291220.1291159. URL: https://doi.org/10.1145/1291220.1291159.

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

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

[56] 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.

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

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

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

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] Fritz Henglein. "Dynamic typing: syntax and proof theory". In: *Sci. Comput. Program.* 22.3 (June 1994), pp. 197–230. ISSN: 0167-6423. DOI: 10.1016/0167-6423(94)00004-2. URL: https://doi.org/10.1016/0167-6423(94)00004-2.

[70] 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.

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

[72] Olivier Danvy and Andrzej Filinski. "Abstracting control". In: *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*. LFP '90. Nice, France: Association for Computing Machinery, 1990, pp. 151–160. ISBN: 089791368X.
DOI: 10.1145/91556.91622. URL: https://doi.org/10.1145/91556.91622.

[73] 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.

[74] 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.

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

[76] 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.

[77] 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.

[78] P. Wadler. "Views: a way for pattern matching to cohabit with data abstraction". In: *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '87. Munich, West Germany: Association for Computing Machinery, 1987, pp. 307–313. ISBN: 0897912152.
DOI: 10.1145/41625.41653. URL: https://doi.org/10.1145/41625.41653.

[79] 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.

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

[81] 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.

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

[83] 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.

[84] Ruth Barcan Marcus. “Extensionality”. In: *Mind* 69.273 (1960), pp. 55–62. ISSN: 00264423, 14602113. URL: http://www.jstor.org/stable/2251588 (visited on 04/16/2025).

[85] Garrett Birkhoff. *Lattice theory*. Vol. 25. American Mathematical Soc., 1940.

[86] Holbrook Mann MacNeille. “Partially ordered sets”. In: *Transactions of the American Mathematical Society* 42.3 (1937), pp. 416–460.

Appendix A

Overview of Semantics and Type Systems

Syntax

Expression-based language syntax are the core foundation behind formal reasoning of programming language semantics and type systems. Typically languages are split into expressions e, constants c, variables x, and types τ. Hazel additionally defines patterns p.

A typical lambda calculus can recursively define it's grammar in the following form:

$$b ::= \text{A set of base types}$$

$$\tau ::= \tau \to \tau \mid b$$

$$c ::= \text{A set of constants of base types}$$

$$x ::= \text{A set of variable names}$$

$$e ::= c \mid x \mid \lambda x : \tau . e \mid e(e)$$

Judgements & Inference Rules

A *judgement*, J, is an assertion about *expressions* in a language [27]. 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 use rules to define a judgement by taking 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. The *typing assumptions*, Γ, is a *partial function*[1] [**PartialFunctions**] from variables to types for variables, notated $x_1 : \tau_1, \ldots, x_n : \tau_2$. For example the SLTC[2] [58, 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 its argument e_2 has type τ_1.

Small Step Operational Semantics

A relation $e_1 \to e_2$ can be defined via judgement rules to determine the evaluation semantics of the language. It's multi-step (transitive closure) analogue is notated $e_1 \to^* e_2$, meaning $e_1 = e_2$ or there exists a sequence of steps $e_1 \to e_1' \to ... \to e_2$.

$$\frac{}{e \to^* e} \qquad \frac{e_1 \to e_2 \quad e_2 \to^* e_3}{e_1 \to^* e_3}$$

Evaluation order can be controlled by considering classifying terms into *normal forms* (values). A call by value language would consider normal forms v as either constants or functions:

$$v ::= c \mid \lambda x : \tau.e$$

Hazel evaluations around holes by treating them as normal forms (final forms). A call by value semantics for the lambda calculus would include, where $[v/x]e$ is capture avoiding substitution of value v for variable x in expression e:

$$\frac{}{(\lambda x : \tau.\ e)v \to [v/x]e} \qquad \frac{e_2 \to e_2'}{e_1(e_2) \to e_1(e_2')}$$

[1] A function, which may be *undefined* for some inputs, notated $f(x) = \bot$.

[2] Simply typed lambda calculus.

Appendix B

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.

B.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 B.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.

B.2 Static Type System

B.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 B.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 B.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 B.4: Type Matching

B.2.2 Elaboration

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

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

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

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

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

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

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

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

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

$$\text{EASubsume}\ \frac{e \neq (\!|\,|\!)^u \quad e \neq (\!|e'|\!)^u \quad \Gamma \vdash e \Rightarrow \tau' \rightsquigarrow d \dashv \Delta \quad \tau \sim \tau'}{\Gamma \vdash e \Leftarrow \tau \rightsquigarrow d : \tau' \dashv \Delta}$$

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

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

Figure B.5: Elaboration judgements

B.2.3 Internal Language

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

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

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

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

$$\text{TACastError}\ \frac{\Delta; \Gamma \vdash d : \tau_1 \quad \tau_1 \text{ ground} \quad \tau_2 \text{ ground} \quad \tau_1 \neq \tau_2}{\Delta; \Gamma \vdash d\langle \tau_1 \Rightarrow ? \nRightarrow \tau_2 \rangle : \tau_2}$$

Figure B.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 B.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 B.8: Ground types

B.3 Dynamics

B.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 B.9: Final forms

B.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 B.10: Instruction transitions

B.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 B.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 B.12: Contextual dynamics of the internal language

B.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.

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

Figure B.13: Hole substitution

Appendix C

On Representing Non-Determinism

High Level Representation

Other high level representations, besides monads, include:

Logic Programming DSL: Languages like Prolog [13] and Curry [6] express non-determinism by directly implementing *choice* via 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 [18] created a tagless final style [38] domain specific language within OCaml. This approach fully abstracts the search procedure from the non-deterministic algorithm constructs.

Delimited Continuations: Delimited continuations [3, 72] are a control flow construct that captures a portion of the program's execution context (up to a certain delimiter) as a first-class value, which can be resumed later (in OCaml, a function). This enables writing non-deterministic code by duplicating the continuations and running them on each possibility in a choice.

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 [5, 33], while handlers can flexibly define the search procedure and accounting logic, e.g. storing solutions in a list. As with delimited continuations, to try multiple solutions, the continuations must be cloned.

Reasoning Against

Direct implementation: This would not allow for easily abstracting the search order and would obfuscate the workings of the indeterminate evaluation and instantiation algorithms. Some sort of high level representation is also massively beneficial for readability and understanding.

Effect handlers: Multiple continuation effect handlers were not supported by Javascript of OCaml (JSOO).[1]

[1] An important dependency of Hazel.

Continuations: Directly writing continuations is difficult and generally more unfamiliar to OCaml developers as opposed to monadic representations.

Optimised DSL : Introducing a formal DSL including optimisations, such as the proposed Tagless-Final DSL [38, 18], is very complex. However, this would allow more flexibility in writing non-deterministic evaluation, with some optimisations made automatically by the DSL.

Appendix D

Monads

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 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 D.1: Monad Definition

Non-Determinism : Choice and failure can be additionally defined on monads:

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

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

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

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

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

Appendix E

Slicing Theory

E.1 Precision Relations

E.1.1 Patterns

E.1.2 Types

E.1.3 Expressions

E.2 Program Slices

E.2.1 Syntax

Including simplified syntaxes

E.2.2 Typing

E.3 Program Context Slices

E.3.1 Syntax

Include pattern contexts

E.3.2 Typing

E.4 Type Indexed Slices

E.4.1 Syntax

Including simplified syntaxes

E.4.2 Properties

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

E.6 Criterion 2: Analysis Slices

E.7 Criterion 3: ...

E.8 Elaboration

Appendix F

Category Theoretic Description of Type Slices

Appendix G

Hazel Term Types

Could be merged into Hazel architecture

Appendix H

Hazel Architecture

More full description of the Hazel code-base architecture.

Appendix I

Code-base Addition Clarification

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

Appendix J

Merges

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

Appendix K

Selected Results

K.1 Type Slicing

Selected examples from the corpus demonstrating each of the slicing techniques

K.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 L

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. [29]. 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 [46], error and dynamic program slicing [60, 77], 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 [20].
- **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.