Max Carroll

Type Error Debugging in Hazel

Computer Science Tripos, Part II

Sidney Sussex College College
University of Cambridge
May 10, 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:	**2328E**
College:	**Sidney Sussex College**
Project Title:	**Type Error Debugging in Hazel**
Examination:	**Computer Science Tripos, Part II – 05/2025**
Word Count:	**14949** [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] *Git diff, between* `Evaluation` *branch and* `dev` *branch. Includes* `.ml` *&* `.sh` *files. Excludes* `/evaluaton/data/*`

Contents

— 1 —
Introduction

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

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

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

In dynamic typing, type errors are often missed as they only appear during runtime with specific inputs. 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* [**TraceVisualisation**] 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 [**Hazel**]:[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* [**HazelLivePaper**]. 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*?

   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*?

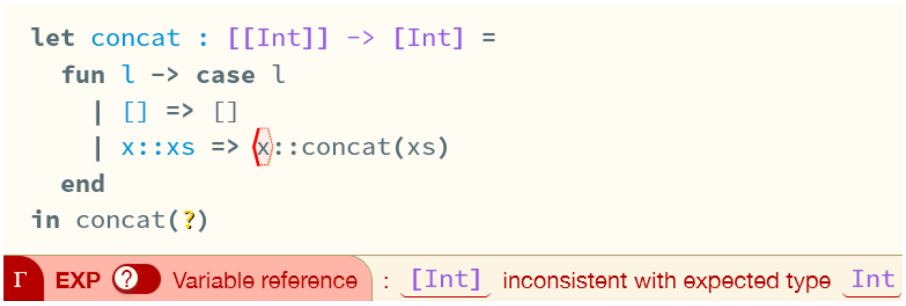
   This would provide an *intuitive* concrete explanation for static type errors.

   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. [**SearchProc**] which devised a similar procedure for a subset of OCaml.

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

Hazel [**Hazel**] 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).



```
let concat : [[Int]] -> [Int] =
  fun l -> case l
    | [] => []
    | x::xs => x::concat(xs)
  end
in concat(?)
```

Γ EXP ? Variable reference : [Int] inconsistent with expected type Int

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

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

— 1.2 — Dissertation Outline

Chapter 2 introduces the *type theories* (section 2.1.1) underpinning the *Hazel core calculus* (section 2.1.2), followed by 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 3.6). This involved created a non-deterministic evaluation method (section 3.5) 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.

— 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 [**DynamicLangShift**, **TIOBE**]. Of particular relevance to this project, execution traces are known to help provide insight to errors [**TraceVisualisation**], yet statically typed languages remove the ability to execute programs with type errors, whereas dynamically typed languages do not.

A *dynamically typed system* can be implemented and represented semantically by use of dynamic *type tags* and a *dynamic type*[1] [**DynamicTyping**]. Then, runtime values can have their type checked at runtime and *cast* between types. This suggests a way to encode dynamic typing via *first-class*[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:

- *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:

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

[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.

[3] Wrapped in a cast between function types.

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

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

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

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

- The *external language* – where static type checking is performed which allows annotating expressions with the dynamic type.
- The *internal language* – where evaluation and runtime type checking is performed via cast expressions.[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* [**TAPL**] with a *top* type $\top$, but with symmetry instead of transitivity.

[4] A bifunctor.

[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 $\tau_1 \to \tau_2 = \tau_1' \to \tau_2$

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

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

Where $\blacktriangleright_{\rightarrow}$ 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 $? \blacktriangleright_{\rightarrow} ? \rightarrow ?$. Intuitively, $e_1(e_2)$ has type τ_2' if e_1 has type $\tau_1' \rightarrow \tau_2'$ or $?$ (then treated as $? \rightarrow ?$), and e_2 has type τ_1 which is consistent with τ_1' and hence is assumed that it can be passed into the function.

But, for evaluation to work the static type information needs to be encoded into casts to be used in the dynamic internal language, for which the evaluation semantics are defined. This is done via *elaboration*, similarly to Harper and Stone's approach to defining (globally inferred) Standard ML [**StandardMLTypeTheory**] by elaboration to an explicitly typed internal language XML [**CoreXML**]. The *elaboration judgement* $\Gamma \vdash e \rightsquigarrow d : \tau$ 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:

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

If e_1 elaborates to d_1 with type $\tau_1 \sim \tau_2 \rightarrow \tau$ and e_2 elaborates to τ_2' with $\tau_2 \sim \tau_2$ then we place a cast[8] on the function d_1 to $\tau_2 \rightarrow \tau$ and on the argument d_2 to the function's expected argument type τ_2 to perform runtime type checking of arguments. Intuitively, casts must be inserted whenever type consistency is used, but which casts to insert are non-trivial [**Gradualizer**].

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

— *Bidirectional Type Systems* —

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

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

We express this with two judgements:

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

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

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

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

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

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

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

We try to *check* e_2 with input τ_1[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]

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

The full semantics are documented in the Hazel Formal Semantics appendix 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]

— *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

[9]Highlighted in red as an error.

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

[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.

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

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

choice made possible by the use of gradual types:

$$\text{SNEHole}\frac{\Gamma \vdash e \Rightarrow \tau}{\Gamma \vdash \llparenthesis e \rrparenthesis^u \Rightarrow ?} \qquad \text{SEHole}\frac{}{\Gamma \vdash \llparenthesis\rrparenthesis^u \Rightarrow ?}$$

One notable consequence of combining gradual and bidirectional typing is that the *subsumption rule* in bidirectional typing is naturally extended to allow subsuming any terms of *consistent* types:

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

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

— *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) [**CMTT**].[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]:

$$\text{TAEHole}\frac{u :: \tau[\Gamma'] \in \Delta \qquad \Delta; \Gamma \vdash \sigma : \Gamma'}{\Delta; \Gamma \vdash \llparenthesis\rrparenthesis^u_\sigma : \tau}$$

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.

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

[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.

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

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

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

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

— *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 [**PracticalFoundations**] 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* [**GradualRefined**], 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

[16] Potentially leading to incorrect cast insertion.
[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*.

values are found by checking the holes *environment*, effectively making a *delayed substitution*. See the following rule:

$$[\![d/u]\!]\llparenthesis\rrparenthesis^u_\sigma = [[\![d/u]\!]\sigma]d$$

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 [**CMTT**].

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

As Hazel is a *pure language*[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 [**HazelCode**] is written primarily in ReasonML and OCaml with approximately 65,000 lines of code. It is under very active development, with much of the code being undocumented; this dissertation summarises the implementation as of April 2025. 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 [**TAPL**].
- **Sum Types** – Representing a value as one of many labelled variants, each of possibly different types [**TAPL**].
- **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 [**TAPL**]. 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 [**TAPL**], for example linked lists or trees.

[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.

— *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:

- **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* [**RealWorldOCaml**]. 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 [**SearchProc**] 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 [**NondeterministicAlgorithms**].

— *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* —

Only the basic foundations of most concepts in understanding Hazel were covered in 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. Monads and non-determinism were also present in this course, but not their intersection.

— *Tools and Source Code* —

My only experience in OCaml was from the Part IA Foundations of Computer Science course. This project builds directly upon the open-source Hazel language codebase [**HazelCode**]. The type witness search procedure is inspired by Seidel et al. [**SearchProc**], however my implementation differs significantly, being applied to Hazel rather than OCaml. Three (DFS, BFS, BDFS) searching methods for monadic non-determinism are based on Spivey [**Bunches**] with minor changes, due to OCaml being a strict language.

[26]For OCaml developers and functional programmers.

— 2.3 — Requirements Analysis

At a high level, this project aimed to achieve three goals:

1. Create a more complete highlighting system for static type errors in Hazel.
2. Create a (complete) source code highlighting system for dynamic errors in Hazel.
3. Provide dynamic witnesses for static errors in Hazel.

At the time of writing the project proposal (appendix I), the second aim was unexplored in existing research, so a concrete requirements were not yet known. Similarly, the first goal had not yet been considered, but naturally follows from the direction that the theorising of cast slicing took.

Developing theories was high risk and took longer than expected. But, significant slack time was allocated to mitigate this, and the original goals were left open to simpler formalisations than the one eventually taken.[27]

After formalisation (phase 1), a set of goals and extensions were devised. This includes those for the witness search procedure taken from the project proposal. The goals should be achieved for a subset of Hazel; Hazel features are classified in fig. 2.3 into: must implement, extensions, and won't implement[28]. Most extensions relate to *maintainability* of code, *conciseness* of slices, witness *coverage* improvements, and *usability* (UI improvements), the latter two considered with lower priority.

Class	Features
Must Implement	Base types: their constants & operations, lists, functions, bindings, type aliases, tuples, sum types & constructors, holes, casts.
Extensions	Pattern matching, labelled tuples (*), type functions, recursive types.
Won't Implement	Tests, deferrals, probes (*).

(*) New features merged from main branch not present in Hazel when proposal was written.

Figure 2.3: Hazel Subsets to Implement for

— *Core Goals* — Create/translate a corpus of ill-typed Hazel programs for evaluation usage. The witness search procedure must have reasonable coverage ($> 75\%$) in a time suitable for interactive debugging (30s) over the corpus. Implement synthesis and analysis type slice theory, and cast slice theory with a basic UI highlighting the source code.

— *Extensions* — Implement contribution slice theory. Customisable search procedure instantiation ordering. *(Maintainability)* Segregate type slicing logic from type checking semantics. Segregate cast slicing logic from transition semantics. *(Conciseness)* Error slices (*). Minimised error slices (**).

— *Low-Priority Extensions* — *(Coverage)* Extended pattern instantiation (**). *(Usability)* Trace visualisation & compression. Graphs for cast dependence. UI to select sub-parts of slices (*).

Extensions with (*) were added post-evaluation and (**) after re-evaluation.

[27]For example, a less complete highlighting based only upon traces.

[28]Those where the project is less applicable

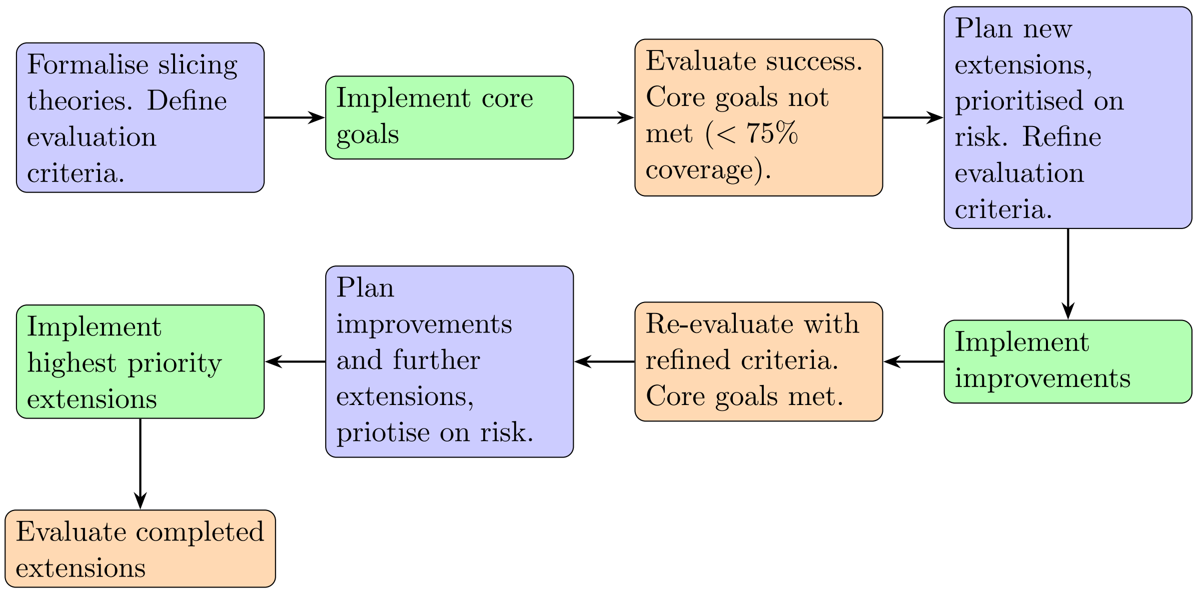

Formalise slicing theories. Define evaluation criteria.
Implement core goals
Evaluate success. Core goals not met (< 75% coverage).
Plan new extensions, prioritised on risk. Refine evaluation criteria.
Implement improvements
Re-evaluate with refined criteria. Core goals met.
Plan improvements and further extensions, priotise on risk.
Implement highest priority extensions
Evaluate completed extensions

Figure 2.4: Phases of development

— 2.4 — Software Engineering Tools and Techniques

— *Methodology* — A spiral development model was followed, with each iteration refining the implementation through defined milestones (see project proposal) and repeated evaluation (fig. 2.4). New extensions were added after each evaluation and prioritised based on their risk to not be fully implemented by a deadline (coverage improvements and usability extensions having higher risk).

— *Hazel Codebase & Interaction* — The Hazel codebase is extensive (65k lines), with much of it being undocumented. As such interaction with the Hazel development team was required to clarify workings. Equally, I found and raised issues on bugs throughout, some being fixed by myself and later merged into the main development branch.

— *Version Control & Merges* — Git and GitHub were used for version control and back-ups, upon a fork of the Hazel codebase. My project had various extensions and alternate implementations and improvements, for which multiple branches were created.

Hazel is a very active research project, so many bugfixes (and bugs introduced) and new features were added over the course of developing this project. These updates were regularly merged into my project, often requiring extensive conflict resolution (Slicing touches almost the entire codebase). Included in this were two major merges for labelled tuples and a UI architecture rewrite. See appendix I for a list of merges.

— *Continuous Integration & Deployment* — The main branch of this project is integrated with Hazel's continuous integration and deployment system (using GitHub actions). As such, unit tests and coverage are performed automatically, and the main branch of this project can be accessed at https://hazel.org/build/witnesses-type-slicing/.[29]

— *Testing* — Hazel performs it's testing by listing code samples with errors labelled by comments. Or, more recently, shifting towards unit regression tests using the *Alcotest* package [**AlcoTest**]. I reuse these to ensure type checking and evaluation is not broken by my additions. However, as slicing involves random term IDs, unit testing is more difficult. Therefore, I use the earlier method of testing directly within the editor, querying the slicing UI to test for calculation errors.

[29]This is continuously deployed. New functionality implemented *after* the deadline may also be present.

— *Tools* — No new dependencies were introduced into Hazel, instead existing dependencies were used, e.g. `Js_of_ocaml` [**JSOO**] for regex matching, Jane Street `Base.Sequence` [**Base**] for streams. However, micro benchmarking of the search procedure was performed using *Bechamel* [**Bechamel**]

— *Licences* — Hazel is open-source available under an MIT licence. The source OCaml corpus of ill-typed programs [**OCamlCorpus**] translated into Hazel is freely available under a Creative Commons Zero (CC0) licence. We again license the project with an MIT licence and the translated Hazel corpus with a CC0 licence.

— 3 — Implementation

This project was conducted in *two* major phases:

1. I devised a mathematical theory for *type slicing* and *cast slicing* and considered changes to the system of Seidel et al. [**SearchProc**] for creating a *type error witnesses search procedure* in Hazel.

2. I iteratively implement and evaluate the theories and search procedure, extending to the majority of Hazel. Suitable deviations from the theory, made upon critical evaluation, are detailed throughout.

— 3.1 — Type Slicing Theory

I develop a novel method, *type slicing*, to aid programmers in understanding *how* a bidirectional type system works. First I define typing slices. Then, three slicing criteria: synthesis, analysis, and contribution slices, each associate typing derivations with different explanatory slices.

The first two criteria give insight on the synthesised and analysed type contributions. The third completes a picture of code regions contributing in any way to a term's type.

The second and third criterion were *very challenging* to formalise, requiring non-obvious mathematical machinery: *context typing slices* (section 3.1.2), *checking contexts* (appendix E.4), and *type-indexed slices* (section 3.1.3). Only the basic definitions are given here, the full theory is found in appendix E.

— 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 example above, we get:

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

We can then define a *precision* partial order [**PartialOrder**] 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. For example:

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

— *Lattice Structure* — For any *complete term*[1] t, the slices of t form a *bounded lattice structure* [**Lattice**]. That is, every pair ς_1, ς_2 has a *join* $\varsigma_1 \sqcup \varsigma_2$ and *meet* $\varsigma_1 \sqcap \varsigma_2$. In general, not all slices slices have joins: $1 \not\sqcup\ 2$, but do have meets as $\square \sqsubseteq \varsigma$ for all ς.

[1]Having no gaps.

— *Typing Assumption Slices* —

Expression typing is performed given a set of *typing assumptions*. Therefore, in addition, we also desire a slice taking the *relevant*[2] assumptions. Typing assumptions are *partial functions* mapping variables to types (see appendix A).

Hence, their slices slices are partial functions to *type slices*. Such that, a slice maps no more variables to no more precise types. This, and meets and joins, can be extended by extensionality [**Extensionality**]:

Definition 1 (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:*

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

Definition 2 (Typing Assumption Slice Joins and Meets). *For typing slices γ_1, γ_2, and any variable x:*

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

Again, slicing complete typing assumptions Γ forms a bounded lattice. In general, some slices have no join: consider $x : \texttt{Int}$ and $x : \texttt{String}$.

— *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.

— *Typing Checking* — *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*. Notated by $[\![\cdot]\!]$.

Definition 3 (Expression Typing Slice Type Checking). *For expression typing slice ς^γ and type τ. $\gamma \vdash \varsigma \Rightarrow \tau$ iff $[\![\gamma]\!] \vdash [\![\varsigma]\!] \Rightarrow \tau$ and $\gamma \vdash \varsigma \Leftarrow \tau$ iff $[\![\gamma]\!] \vdash [\![\varsigma]\!] \Leftarrow \tau$.*

— 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 and Their Slices* —

We represent these surrounding contexts by a *term context* $\mathcal{C}$. Which marks *exactly one* sub-term as $\bigcirc$. Where $\mathcal{C}\{t\}$ substitutes term t for the mark $\bigcirc$ in $\mathcal{C}$, only allowed if the marked position expects a term of the same class as t (pattern `Pat`, type `Typ`, or expression `Exp`). Notate the classes by $\mathcal{C} : \texttt{X} \to \texttt{Y}$. Contexts are *composable*: $(\mathcal{C}_1 \circ \mathcal{C}_2)(t) = \mathcal{C}_1\{\mathcal{C}_2\{t\}\}$ when $\mathcal{C}_1 : \texttt{X} \to \texttt{Y}$ and $\mathcal{C}_2 : \texttt{Y} \to \texttt{Z}$. Composition is associative.

These extend to slices analogously to term slices. However, the precision relation $\sqsubseteq$ more restrictive, requiring the mark $\bigcirc$ to remain in the same structural position. For example: $\bigcirc(\square) \sqsubseteq \bigcirc(1)$, but $\bigcirc \not\sqsubseteq \bigcirc(1)$. Concisely defined by *extensionality*:

Definition 4 (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 $c'\{t\} \sqsubseteq c\{t\}$.*

[2]To the given criterion.

We find that filling contexts preserves the precision relations both on term slices *and* context slices:

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

Joins and meets can be defined extensionally as before, still forming bounded lattices over complete contexts. The lattice bottom is the *purely structural context*, consisting of only gaps with the mark in the correct position. In general, in addition to joins, not all contexts have meets: $\bigcirc \not\sqcap \bigcirc(\square)$.

— Typing Assumption Contexts and Their Slices —

The accompanying notion typing notion can be represented by *endomorphisms on typing assumption slices*. These functions represents which *relevant* typing assumptions must be *added*, and those safely *removable* when typing an expression within a context slice.

Precision, joins, and meets can be defined via extensionality. As usual forming bounded lattices on complete functions, the bottom being the constant function to the empty typing assumptions. Again, such functions are monotone:

Proposition 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 $f'(\gamma') \sqsubseteq f(\gamma)$.*

— Context Typing Slices —

Finally, an *expression context typing slice*, ρ, is a pair, c^f, of an expression context slice and a typing assumption context slice defined pointwise, with all the same properties (including composition).

— Type Checking — An analogous $[\![\cdot]\!]$ translation can be defined.

— 3.1.3 — Type-Indexed Slices

Decomposing slices by their type is required for cast slicing and useful in calculating slices according to analysis slices and hence also contribution slices. For example consider the following context slice explaining why the underlined term analyses $\texttt{Bool} \to \texttt{Int}$:

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

This would be tagged with the type $\texttt{Bool} \to \texttt{Int}$ where a sub-slice considering *only* the $\texttt{Int}$ return type (omitting the $\texttt{Bool}$ annotation) can be extracted:

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

This section will only consider *context slices*, but term slices are type-indexed analogously.

The main property that indexed-slices should maintain is that slices can be *reconstructed* from their sub-parts. Joining the sub-slices will produce the full type. As sub-slices may slice different regions of code, we pair them with contexts which place the sub-slices within the same context, making them join-able.

Definition 5 (Type-Indexed Context Typing Slices)**.** *Syntactically defined:*

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

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

$$\overline{\rho} = \rho$$

$$\overline{\rho_1 * \mathcal{S}_1 \to \rho_2 * \mathcal{S}_2} = \rho_1 \circ \overline{\mathcal{S}_1} \sqcup \rho_2 \circ \overline{\mathcal{S}_2}$$

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

Definition 6 (Type-Indexed Context Typing Slice Composition)**.** *For type-indexed context typing slices $\mathcal{S}$ and $\mathcal{S}'$. If $\mathcal{S} = p$ and $\mathcal{S}' = p'$:*

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

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

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

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

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

Application, notated $|\!>$ be defined similarly, converting indexed context slices into valid indexed expression slices. The opposite direction is more difficult and can be found in appendix (appendix E.3.3).

— 3.1.4 — Criterion 1: Synthesis Slices

Synthesis slices aim to explain why an expression *synthesises* a type. They 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` $\to$ `Bool` type, and the variable x : `Int` and argument are irrelevant:

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

Definition 7 (Synthesis Slices)**.** *For a synthesising expression, $\Gamma \vdash e \Rightarrow \tau$. 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 4 (Minimum Synthesis Slices)**.** *A minimum synthesis slice of $\Gamma \vdash e \Rightarrow \tau$, under $\sqsubseteq$, exists and is unique.*

These slices can be calculated 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 judgement rules mimic Hazel's typing rules, giving an algorithm to calculate minimum synthesis slices (see appendix E.5).

— 3.1.5 — Criterion 2: Analysis Slices

A similar idea can be devised for type analysis, represented using *context slices*. After all, 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 $\mathtt{Int}$ due to the annotation and lambda constructor present in it's context. The analysis slice will be:

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

In other words, if the context slice was type checked, then the inner hole would *still* required to analyse against $\mathtt{Int}$. However, the overall synthesised type may differ, this sliced example would synthesis $\mathtt{?} \to \mathtt{Int}$ vs. the unsliced $\mathtt{Bool} \to \mathtt{Int}$.

— *Checking Context* —

We only want to consider the smallest context scope[3] that enforced the type checking. For example, the below term has 3 annotations, but only the inner one enforces the $\mathtt{Int}$ type on the integer 1:

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

I refer to this as the *minimally scoped checking context.* Note that checking contexts will always synthesise a type.

To give another example, an integer argument's type is enforced by the annotation on the function:

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

Definition 8 (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:*

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

Definition 9 (Minimally Scoped Checking Context)**.** *For a derivation $\Gamma \vdash e \Leftarrow \tau$, a minimally scoped expression checking context is a checking context of e such that no sub-context is also a checking context.*

All minimally scoped checking contexts can be constructed syntactically via rules (appendix E.4). For a sub-term in a program, there will be a unique minimally scoped context which matches with the program structure (appendix proposition 24). Analysis slices are slices of minimally scoped checking contexts.

Definition 10 (Analysis Slice)**.** *For $\Gamma \vdash e \Leftarrow \tau$ with a minimally scoped checking context $\mathcal{C}^{\mathcal{F}}$. An analysis slice is a context slice c^f of $\mathcal{C}^{\mathcal{F}}$ where $[\![c^f]\!]$ is also a checking context for e.*

Conjecture 1 (Minimum Analysis Slices)**.** *A minimum analysis slice of $\Gamma \vdash e \Leftarrow \tau$ in a checking context $\mathcal{C}^{\mathcal{F}}$, under $\sqsubseteq$, exists and is unique.*

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

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

The rules build upon rules defining minimally scoped checking contexts, and are more involved, making use of *type-indexed* synthesis slices[4], see fig. 3.1:

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

[4]Hence, requiring conversion of (indexed) expression slices to context slices

$(\lambda x : ?.\lambda y : \texttt{Int}.y)(\texttt{true})$

(a) A function: synthesising Int → Int.

$(\lambda x : ?.\lambda y : \texttt{Int}.y)(\texttt{true})$

(b) Its slice.

$(\lambda x : ?.\lambda y : \texttt{Int}.\ y)(\texttt{true})$

(c) The sub-slice relating *only* to the input part Int.

$(\lambda x : ?.\lambda y : \texttt{Int}.\ y)(\texttt{true})(\underline{1})$

(d) The analysis slice of the function's argument ($\underline{1}$) when applied.

Figure 3.1: Analysis slice application uses synthesis slices

— 3.1.6 — Criterion 3: Contribution Slices

This criterion highlights all regions of code which *contribute* to typing succeeding. That is, all sub-terms who could change their type to make the term ill-typed. For example, for the underlined term:

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

Both contextual and synthetic parts (in dark yellow) contribute:

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

Notice that the only sub-terms which do not contribute have their type changed without affecting the overall type must be expected to be dynamically annotated. Therefore, this criterion omits the dynamic code regions.

These can be calculated mimicking the Hazel typing rules. During subsumption, remove synthesis slice sub-part which match with dynamic parts of the analysing type. Contextual parts are found by passing context slices directly as inputs to type analysis rules.

— 3.2 — Cast Slicing Theory

Cast slicing propagates type slice information during evaluation, by tagging casts types with type slices. A primary reason in formalising type-indexed slices was to make slices decomposable during evaluation, retaining only sub-slices relevant to the part of the type involved in the decomposed cast. This requires changing the syntax of casts to $\langle \mathcal{S} \Rightarrow \mathcal{S} \rangle$ and inserting casts between slices during elaboration. The first two criteria work together during elaboration, analysis slices inserted when casting on a elaborated expression who had it's type *analysed*, and the synthesised for synthesis slices. The rules are found in appendix E.8.

— 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 terms as a standard abstract syntax tree (AST) via mutual recursion. Every sub-term is tagged with an identifier (ID, `ID.t`). Terms are grouped similarly to the calculus (see section 2.1.2), but combining external and internal expressions, and adding patterns and environments, see fig. 3.2 & 3.3.

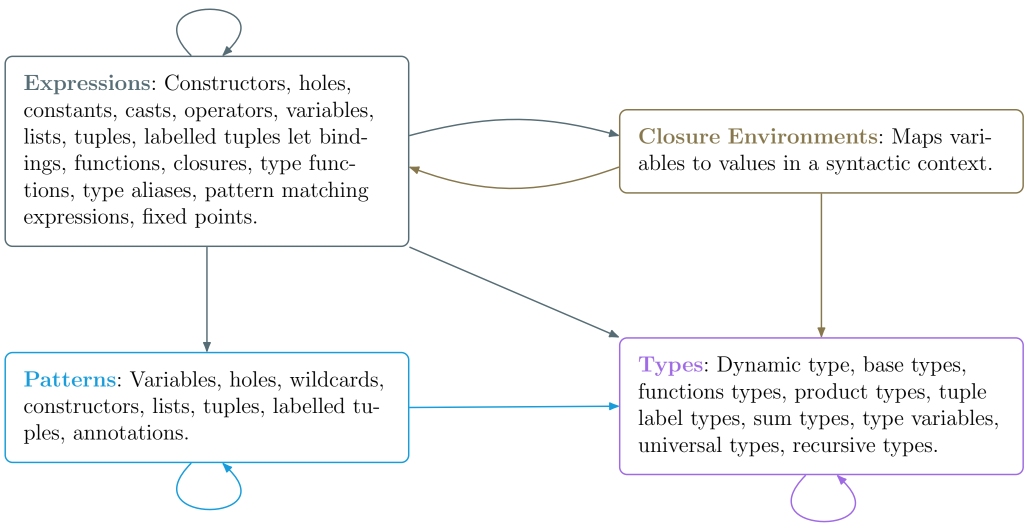

Expressions: Constructors, holes, constants, casts, operators, variables, lists, tuples, labelled tuples let bindings, functions, closures, type functions, type aliases, pattern matching expressions, fixed points.
Closure Environments: Maps variables to values in a syntactic context.
Patterns: Variables, holes, wildcards, constructors, lists, tuples, labelled tuples, annotations.
Types: Dynamic type, base types, functions types, product types, tuple label types, sum types, type variables, universal types, recursive types.

Figure 3.2: Mutually Recursive Hazel Terms.

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

Figure 3.3: Let binding a tuple patter with a type annotation.

— 3.3.2 — Type Slice Data-Type

— Expression slices as ASTs —

Directly storing expression slices directly as ASTs is both *space and time inefficient*, even when accounting for the persistence [**PurelyFunctionalDataStructures**] of trees in OCaml.

For highlighting purposes, there is no need to retain the structure (unlike the theory, we do not need to type check the results, instead just assuming correctness).

— Unstructured Code Slices —

With this in mind, I represent slices indirectly by their IDs with an *unstructured* list, referred to now as a *code slice*. Additionally, this allows more *granular* control over slices, as they need not conform with the structure of expressions, which is taken advantage of in reducing slice sizes section 4.8.1.

— Type-Indexed Slices —

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

```
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 3.4: Initial Type Slice Data-Type

However, this did not model the structure of type slices particularly well. Analysis slices add ids to *all* sub-slices, giving *linear* space complexity in the depth of the type.

— *Incremental Slices* —

Therefore, slices are represented incrementally. With *incremental slices* for synthesis slice parts and *global slices* for analysis slice parts.

A *global slice* is only tagged once, then *lazily* tagged to sub-slices if used. That is, when de-constructing types, e.g. in function matching. We get the type in fig. 3.5:

```
...
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)
...
```

Figure 3.5: The type slice data-type

The invariant that a slice has at most *one* incremental and/or global slice is maintained by splitting into three types (`empty_term`, `incr_term`, `term`). Regular un-sliced types `` `Typ(...) `` are maintained to provide easier interoperability with the rest of the code-base, also allowing type slicing to be turned off.

Polymorphic Variants [**RealWorldOCaml**], notated `[ | ... ]` are used to more conveniently write functions on slices. This is possible due *row polymorphism* [**PolymorphicVariants**] [**ATTAPL**] relating the variants by a *structural subtyping* relation [**StructuralSubtyping**]. We have that:[5]

$$\texttt{typslice\_empty\_term} :> \texttt{typslice\_incr\_term} :> \texttt{typslice\_term}$$

Type constructors are either co-variant or contra-variant [**BasicCatTheory**] with respect to the subtyping relation. For example, id tagging is covariant, so:

$$\texttt{IdTagged.t(typslice\_incr\_term)} :> \texttt{IdTagged.t(typslice\_incr\_term) = typslice\_t}$$

Functions are bifunctors: contravariant in their input and covariant in their output, for example:

$$\texttt{typslice\_incr\_term} \to \texttt{typslice\_incr\_term} :> \texttt{typslice\_empty\_term} \to \texttt{typslice\_term}$$

This function subtyping property significantly reduces work in defining functions on slices (see fig. 3.6).

— *Utility Functions* — Functions on slices often only concern the underlying type, e.g. checking if a slice is a list type. Writing direct pattern matching code on `typ_term` and `typslice_term` is easier. An `apply` function can apply these direct functions to the term inside a slice. The bottom two branches can both be passed into `apply` function as they are

```
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)
  }
```

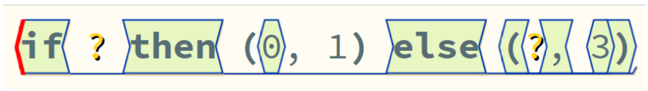
Figure 3.6: Apply Utility Function

sub-types of `typslice_term`. Many other utility functions are implemented, including mapping functions, wrapping functions, unpacking functions, matching functions.

— *Type Slice Joins* —

Type joins (section 3.1.1) are extensively used in the Hazel implementation for *branching statements* and in the theory of contribution slices.

For basic type slices, when the same type information is available from multiple branches, highlighting only one branch is required, but both for contribution slices. See that the 1 in fig. 3.7 is not highlighted (in green). However, we did still need some information from the left (the 0).



```
if ? then (0, 1) else ((?, 3))
```

Figure 3.7: Type Slice Joins

This gets complex, requiring tracking all the data relating to which branches should be highlighted, and accumulating the inconsistent parts of failed joins for use in minimised error slices (demonstrated in section 4.8.1). I make use of custom OCaml let bindings to retain clarity while automatically combining branches in successful joins and combining inconsistencies in failed joins. For example, see fig. 3.8, where comments describe the logic abstracted by the custom bindings; note that all join-able type combinations use this same parallel binding logic to combine branches and inconsistencies.

— 3.3.3 — Static Type Checking

Hazel is *bidirectionally typed*, where the *mode* (synthesis, analysis) is specified by the `Mode.t` type. Type checking calculates a type information object `Info.t` for each term, stored efficiently in a map from ID keys. `Info.t` is demonstrated in fig. 3.9 with arrows representing dependencies (e.g. a term's type depends on it's mode, self, typing context, and status).

— *Self and Mode* —

Slicing logic relating to synthesis slices and analysis slices is factored into `Self.t` and `Mode.t` respectively, cleanly segregated from the type checking code. Although, doing this still required full understanding of the type checker implementation, ensuring the correct IDs are sliced. Two examples of the slicing logic are shown in appendix F.

[5] $x :> y$ meaning x a subtype of y.

```
...
| (Arrow(s1, s2), Arrow(s1', s2')) =>
      let+ s1j = join'(s1, s1') // If successful: binds the join to s1j and
                                // propagates the branch used joining s1, s1'
                                // Else: Propagates inconsistencies in s1, s1'
      and+ s2j = join'(s2, s2') // If both successful: binds the join to s2j and
                                // merges previously used branches joining s1, s2'
                                // with those used joining s2 and s2'
                                // Else: propagates inconsistency in s1, s2' (if any)
                                // alongside new inconsistencies in s2, s2' (if any)
      and! branches_used = ();  // If both successful: Binds combined branches
                                // Else: propagates inconsistencies
      ( // If nothing failed, Returns the successful join.
        `SliceIncr((
          Slice(Arrow(s1_join, s2_join)), // The successful join
          choose_branch(branches_used, slice_incr1, slice_incr2),
        )) // Wrap with only one slice: the left if both branches used
        |> temp,
        branches_used, // combined branches used
      );
...
```

Figure 3.8: Joining function types

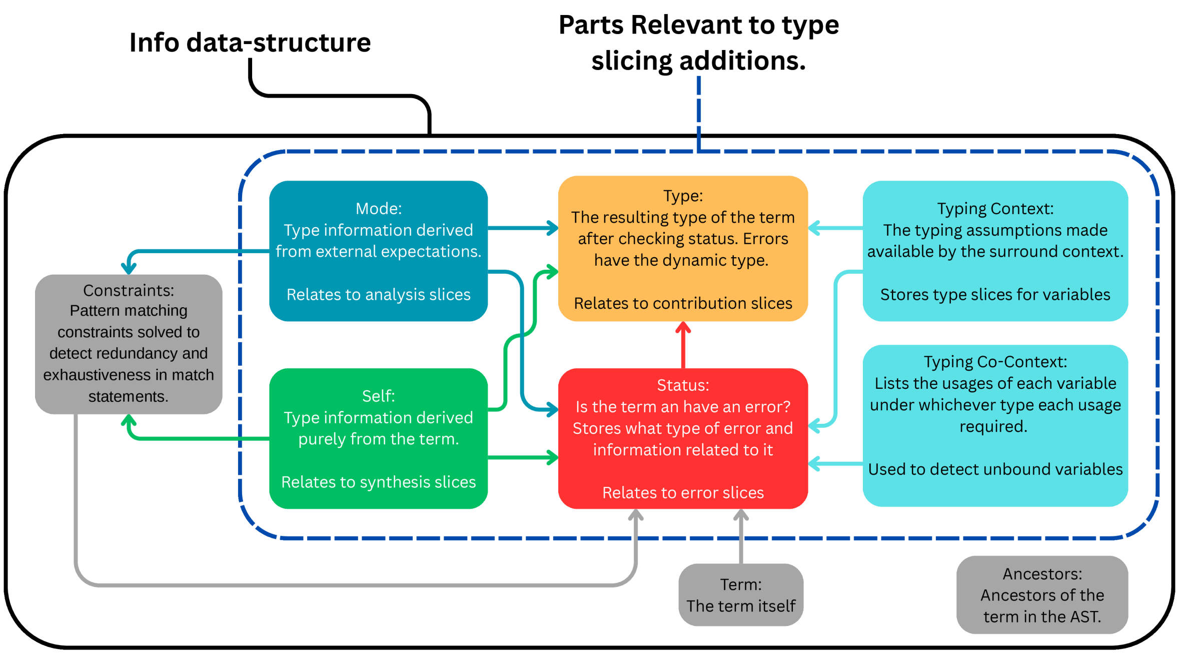
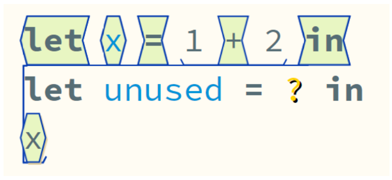

Info data-structure
Parts Relevant to type slicing additions.
Mode: Type information derived from external expectations. Relates to analysis slices
Type: The resulting type of the term after checking status. Errors have the dynamic type. Relates to contribution slices
Typing Context: The typing assumptions made available by the surround context. Stores type slices for variables
Constraints: Pattern matching constraints solved to detect redundancy and exhaustiveness in match statements.
Self: Type information derived purely from the term. Relates to synthesis slices
Status: Is the term an have an error? Stores what type of error and information related to it. Relates to error slices
Typing Co-Context: Lists the usages of each variable under whichever type each usage required. Used to detect unbound variables
Term: The term itself
Ancestors: Ancestors of the term in the AST.

Figure 3.9: Info.t data-structure

— Typing (Co-)Context —

The typing context and co-contexts are modified to use type slices. 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. Therefore, it requires using *unstructured* code slices. It is useful in practice allowing slices calculated during binding to be retrieved usage, see fig. 3.10.



```
let x = 1 + 2 in
let unused = ? in
x
```

Figure 3.10: Type slice for variable `x`: includes it's binding and slice.

— *Status and Type* —

The status, mode, and self are combined to determine a term's actual type, being dynamic if there is an error. When the expectations (mode, self) are inconsistent, the inconsistent slice information parts are tagged to the error status; section 4.8.1 retrospectively considers what slice information to be extracted here. Additionally, contribution slices can extract the static parts of synthesis slices here.

— 3.3.4 — Integration

To support the full Hazel language, type slices needed to implement many functions, for example: type substitution[6], type normalisation, weak-head normalisation, tracking sum types, various structural matching functions etc. Additionally almost every usage of types in the codebase had to be refactored to use type slices (which are so easily pattern matched upon) while ensuring slices correctly maintained.

— 3.3.5 — User Interface & Examples

The type slices of the expression at the cursor (in red) are highlighted, see fig. 3.11. Error slices distinguish between the synthesis and analysis parts with blue and red, see the evaluation examples section 4.9.2.

— 3.4 — Cast Slicing Implementation

To implement cast slicing, replace casts between *types* by casts between *type slices*. Type slices are already type-indexed and retain all type information so can be used equivalently.

— 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. For example, a list literal recursively elaborate it's terms and joins their slices, casting to the join. Ensuring all the type slice information is retained and/or reconstructed during elaboration was a meticulous and error-prone process.

— 3.4.2 — Cast Transitions

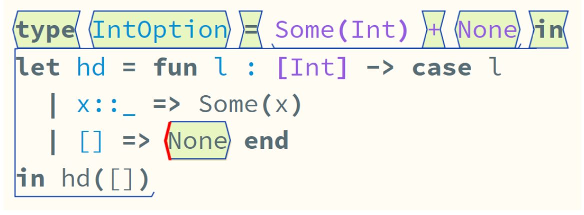
Section 2.1.2 gave an intuitive overview of how casts are treated at runtime. Type-indexed slices allows cast slices to be decomposed in exactly the same way.

However, as Hazel only checks consistency between casts between *ground types*, there are two rules where new[7] casts are *inserted* (ITGround, ITExpand). The new types are created via a *ground matching* relation taking the topmost compound constructor of types, e.g. ground functions fig. 3.12. Relevant portions of the appendix are fig. B.8, fig. B.10, fig. B.11.

As we store type slices incrementally, the part of the slice corresponding *only* to the outer type constructor is the outer slice tag.

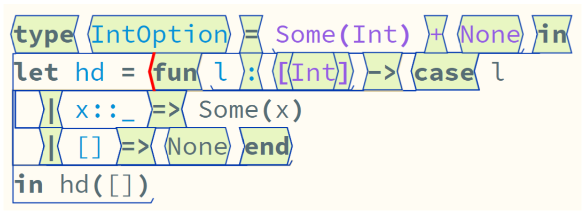
[6]Note: this is the only feature which does not currently retain type slices fully.

[7]As opposed to being derived from decomposition.



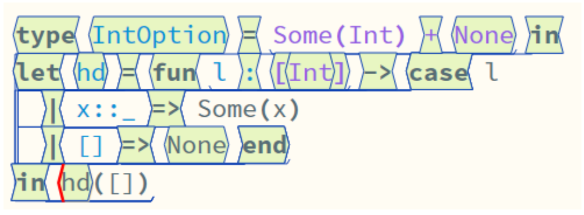
```
type IntOption = Some(Int) + None in
let hd = fun l : [Int] -> case l
  | x::_ => Some(x)
  | [] => None end
in hd([])
```

(a) `None` synthesises `IntOption` due to it's type definition.



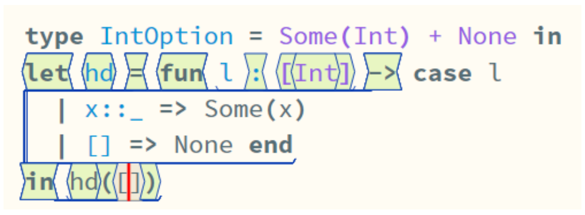
```
type IntOption = Some(Int) + None in
let hd = fun l : [Int] -> case l
  | x::_ => Some(x)
  | [] => None end
in hd([])
```

(b) The function synthesises `[Int]`→ `IntOption` due to it's `[Int]` annotation and that the match branches synthesis `IntOption`. Both branches provide the same type information, only one branch (the last) is highlighted.



```
type IntOption = Some(Int) + None in
let hd = fun l : [Int] -> case l
  | x::_ => Some(x)
  | [] => None end
in hd([])
```

(c) The variable usage of **hd** synthesises `[Int]`→ `IntOption` similarly, whose slice is retrieved from the typing context.



```
type IntOption = Some(Int) + None in
let hd = fun l : [Int] -> case l
  | x::_ => Some(x)
  | [] => None end
in hd([])
```

(d) The list input is expected to be an `[Int]` as it is applied to **hd** which is a function annotated with input type `[Int]`.

Figure 3.11: Type Slices

$$\tau_1 \to \tau_2 \blacktriangleright_{\text{ground}} \,?\, \to \,?$$

Figure 3.12: Ground Matching Functions

— 3.4.3 — Unboxing

When we know a final form's (section 2.1.2) type, we may need to extract parts of the term according to the type during evaluation. For example, extracting the elements of a tuple. But due to casts and holes, this is not trivial [**LivePatternMatching**].

Slicing does not concern unboxing, but indeterminate evaluation (section 3.5) unveiled bugs within which had to be fixed. I raised and fixed these, eventually being merged into the main branch. Appendix G details this, after covering the required context on the unboxing implementation.

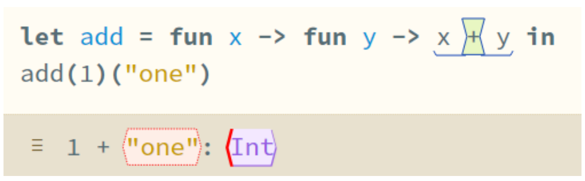
— 3.4.4 — User Interface & Examples

Type slices within casts can be selected from the evaluation result and displayed. This require reworking some of the dependencies of Hazel's model-view-update architecture to make sure the cursor has access to the code editor cell when inside the evaluation result cell.

— 3.5 — Indeterminate Evaluation

Dynamic errors include evaluation traces, which can aid in debugging [**TraceVisualisation**], yet static type errors lack such traces. Seidel et al. [**SearchProc**] offer an algorithm to search for these traces for static errors in OCaml by lazily, non-deterministically narrowing input holes to least specific values based on their usage context.

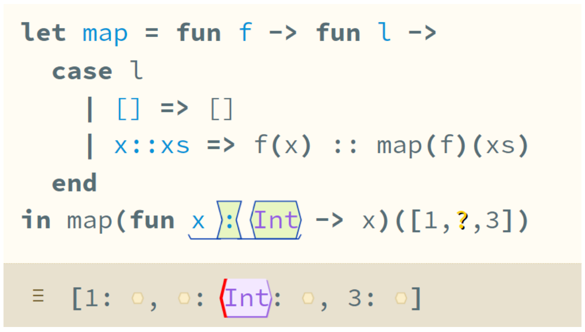
This section creates a framework for non-deterministic evaluation of indeterminate expressions by lazily performing hole substitutions using type information from dynamic casts. Unlike Seidel, this supports more language features (all of Hazel), any number of inputs (holes), and



```
let add = fun x -> fun y -> x + y in
add(1)("one")
```

```
≡ 1 + "one": Int
```

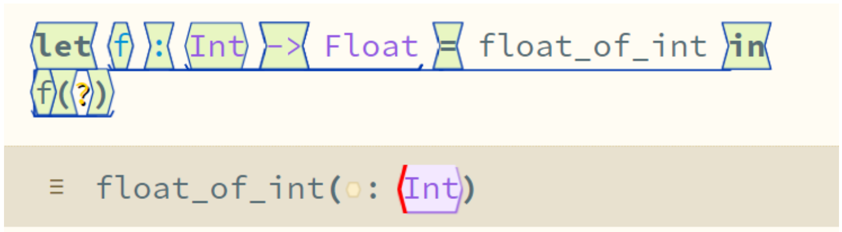
(a) A simple cast error blaming the plus operator for requiring the integer cast.



```
let map = fun f -> fun l ->
  case l
    | [] => []
    | x::xs => f(x) :: map(f)(xs)
  end
in map(fun x : Int -> x)([1,?,3])
```

```
≡ [1: ○, ○: Int: ○, 3: ○]
```

(b) A hole cast to `Int` due to a mapped function annotated with an `Int` input.



```
let f : Int -> Float = float_of_int in
f(?)
```

```
≡ float_of_int(○: Int)
```

(c) A decomposed cast. The input of the function takes only slice for the argument part `Int` of the function type `Int` → `Float`.

Figure 3.13: Cast Slicing Examples

exhaustive generation of these inputs. It is a general evaluation method, not limited to cast error searches. Specifics relating to cast errors and implementing search orderings, are covered in section 3.6.

— 3.5.1 — Resolving Non-determinism

To model infinite non-determinism I create a monadic DSL with an explicitly tree/forest-based representation. The forest model allows for varying low level search traversals. The module type of combinators is in `Nondeterminism.Search`; it's underlying parametric type is `t('a)` with `'a` being the type of the solutions. Section 3.6.3 discusses the actual implementations of this interface, giving four searching procedures.

In addition the functions from Section 2.1.4, I add standard `map` and `join` functions, and various other functions, e.g. `ifte` modelled after Mercury's if-then-else construct [**Mercury**].

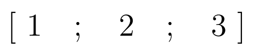
— *Abstracting Search Order: Forest Model* —

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

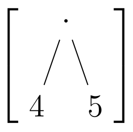
Instead, we can use a model based on forests (lists of trees) [**Bunches**]. Choice, similarly to streams, is performed by concatenating forests. Finally, to build tree structures, a `wrap` combinator wraps every tree up into a single tree with a new root, see fig. 3.14. Effectively, this encodes a notion of cost for search paths.

The DSL is extended with a `wrap : t('a) => t('a)` combinator. As `wrap` is abstract, the underlying implementation does not actually need to use a forest data structure.[8] Finally, a `run` function produces a lazy list of solutions in the tree with ordering specified by the search method.

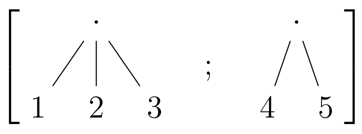
[8]Therefore, DFS is still implemented with regular streams.



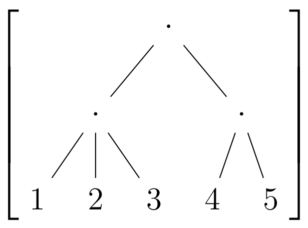
(a) `let x = return(1) <||> return(2) <||> return(3)`



(b) `let y = wrap(return(4) <||> return(5))`



(c) `let z = wrap(x) <||> y`



(d) `wrap(z)`

Figure 3.14: Forests Defined Using `wrap`

— Recursive Functions —

As OCaml is strict, defining infinite choices via recursion can lead to non-termination during definition. I define a shorthand lazy application function `apply(f, x)` by `return(x) >>= f`, represented infix by `|>-`. Provided that bind lazily applies `f`,[9] recursive functions can be written directly resulting in infinite choices without OCaml's strictness leading to infinite recursion.

— 3.5.2 — A Non-Deterministic Evaluation Algorithm

This section demonstrates one indeterminate evaluation algorithm evaluating terms to concrete values in fig. 3.15 with an accompanying code extract fig. 3.16.

Instantiation is implemented by a *non-deterministic* function `instantiate`, discussed in detail in section 3.5.3:

```
Instantiation.instantiate : Exp.t => S.t(Exp.t)
```

Evaluation steps are performed by a (deterministic) function `take_step`, classifing it's input as either (concrete) values, indeterminate values, steppable expressions. The step evaluator re-uses the Hazel stepper logic, which had existing bugs to be fixed: causing non-termination of fixed points[10] and misclassifying concrete values as indeterminate values:

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

To ensure that the search tree has finite branching factor, possibly infinite choices must be wrapped, e.g. evaluation steps. The actual implementation additionally threads state tracking number of instantiations and trace length throughout the algorithm.

— 3.5.3 — Hole Instantiation & Substitution

This section details the semantics of hole instantiations, including Hazel-specific issues.

— 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*?

[9] Which it usually does, consider streams as an example.
[10] Due to incorrect management of closures; the main branch stepper is still broken as of May 9th.

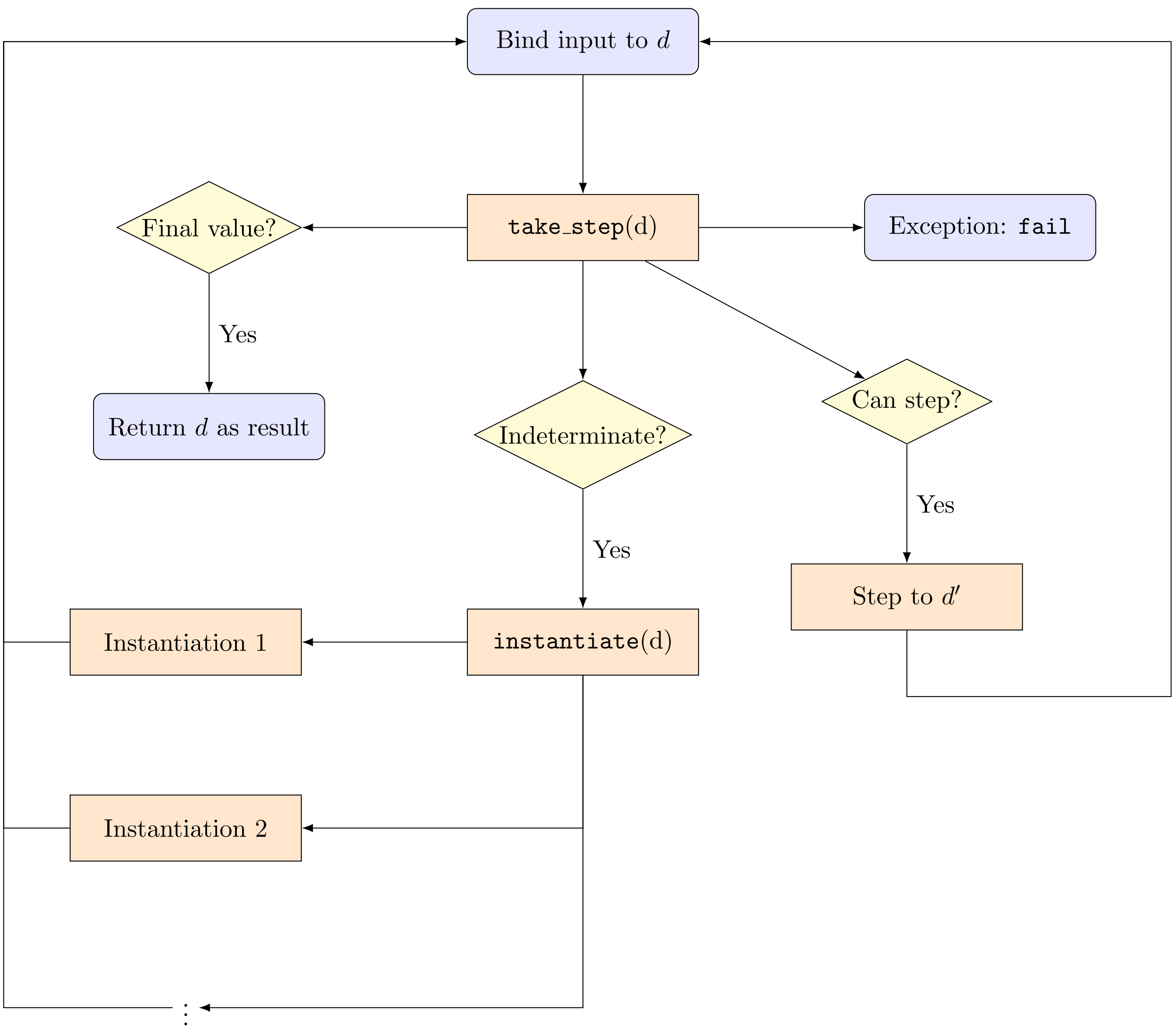


Figure 3.15: Block diagram of indeterminate evaluation to values

When attempting to evaluate the indeterminate term some transitions rules require a sub-term to be concrete (e.g. a function during application). We chose to instantiate the hole that blocks the *first* blocked transition rule. If latter holes were instantiated, the term might *still* be unevaluable due to this first hole.

This is implemented using Hazel's evaluator abstraction (`EV_MODE`), separating this logic from the transition semantics. Therefore, hole choosing logic will automatically update to future changes in the transition semantics.

— *Synthesising Terms for Types* —

Suppose we know which hole to instantiate and to which type (section 3.5.4). How do we refine these holes fairly and lazily, to the *least specific* value that allows evaluation to continue?

Base types must be instantiated directly to their (possibly infinite set of) values, for example:

— *Booleans* — `return(true) <||> return(false)`.

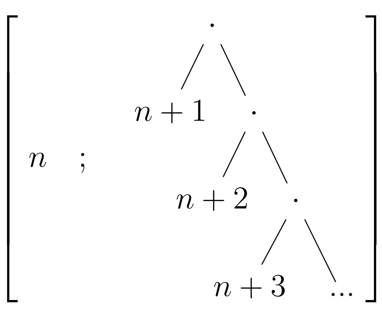
— *Integers* — A recursive definition using lazy application `|>-`, see fig. 3.17:

— *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. 3.18:

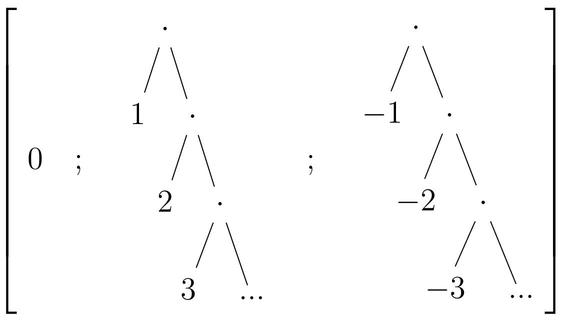
```
module Make = (S: Search) => {
  module Instantiation = Instantiation.Make(S);
  open S;
  open S.Infix;

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

Figure 3.16: Indeterminate Evaluation to Values

```
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)
```



(a) `ints_from(n)`



(b) `ints`

Figure 3.17: Enumerating Integers

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

— *Lists* — A list is either the empty list, [ ] or a cons $?_1 :: ?_2$. To retain the correct dynamic type information, $?_2$ must be cast back to the list type.

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

— *Functions* — Constant functions have least specific values $\lambda\_.\ ?$. The function may then be applied to any value, and it's result synthesised after application. This can synthesise any return value, hence errors in the usage of the function will be detected. But, when the *input*

```
let chars = ... // Choice every single letter string to be considered
let rec strings = () => return("")
  <||> wrap(chars >>= chr =>
            (() |>- strings) >>= str =>
            chr ++ str))
```

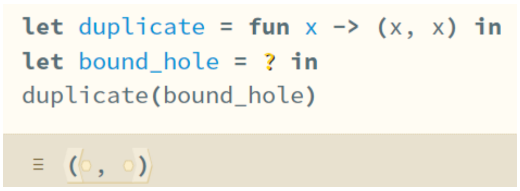
Figure 3.18: Enumerating Strings

has an erroneous type but is not yet caught,[11] these will be lost.[12]

— *Maintaining Correct Casts* — Holes have a dynamic type at runtime. Therefore, the hole's context *expects* a dynamic expression. Therefore, we must cast every instantiation back to the dynamic type.

— *Substituting Holes* —

Holes can be bound to variables in the execution environment, and may also be duplicated, before they are required to be instantiated, see fig. 3.19. Every occurrence must be substituted.



```
let duplicate = fun x -> (x, x) in
let bound_hole = ? in
duplicate(bound_hole)

≡ (◦, ◦)
```

Figure 3.19: Duplicated Holes

Hole substitution was described as part of the Hazel calculus section 2.1.2. But, unexpectedly, the main Hazel branch does not yet implement it. A full implementation of metavariables and delayed closures is complex. Therefore, as hole closures are not required for hole instantiation,[13] I use existing term ids identify holes, ensuring these are maintained and propagated correctly, so that duplicated holes retain the same ID.[14]

Substitutions must also be performed within closures, eagerly evaluating the results to ensure that closures bind variables to values.

— 3.5.4 — Determining the Types for Hole Substitutions

If we know which hole to instantiate, how do we know which type to instantiate it to? This logic is highly specific to Hazel's cast semantics, with many Hazel-specific issues arising.

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, this is *not* always the case. For efficiency reasons, Hazel treats casts over compound data lazily, e.g. casts around tuples will only by pushed inside upon usage of a component of the tuple. Treating casts eagerly is a significant change to the Hazel semantics, so was opted against. Section 4.8.4 evaluates the consequence of this choice.

[11] Due to partial annotations.
[12] This is rare, and could be mitigated by generating the identity function where possible.
[13] Instantiations do not contain variable references.
[14] All of elaboration and dynamics required these checks.

— Pattern Matching —

Does a hole even have only one possible type? Dynamic pattern matching actually allows terms to be matched against *non-uniform* types. See fig. 3.20, having branches having patterns of multiple types. Hence instantiating to any of the types might allow progress.

```
case ?
  | 0 => true
  | 0. => true
  | "zero" => true
  | _ => false
end
```

Figure 3.20: Dynamic/Type-case Match Statement

We can collect each of these possible types from the elaborated casts inserted on the *branches*, non-deterministically rewrapping them around the scrutinee.

— Extended Match Expression Instantiation (Pattern Instantiation) —

An interesting extension was partially implemented which improves code coverage and additionally detects errors within patterns. It instantiates holes in a match expression according also to the *structure* of each pattern, allowing the instantiation to prioritise searching along each branch. We instantiate the scrutinee with the least specific versions which match the patterns on each branch, e.g. `?::?` for `x::xs`. However, difficulties come when the scrutinee is a compound term, or when least specific matches only indeterminately matching earlier branches. Further details addressing these issues can be found in appendix H.

— 3.5.5 — User Interface

The default evaluation method returns every possible indeterminate or concrete values. The possibilities can be cycled through via some arrows buttons.

— 3.6 — Search Procedure

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

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

3.6.2 What exactly are cast errors? How can they be detected? Which ones are actually *relevant*, causing evaluation to get stuck?[15]

3.6.3 How can different search methods be implemented: DFS, BFS, Bounded DFS, Interleaved DFS?

— 3.6.1 — Abstract Indeterminate Evaluation

We reuse the previous framework (fig. 3.15), but abstract the the return logic to a higher-order `logic` function. This also allows returning other types, for example, the integer *size* of values. Or return only specific classes of expressions, e.g. those with cast errors. An 'expert' search abstraction is also provided which allows custom instantiation to be injected.

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

— 3.6.2 — 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. But these failures won't stop evaluation until necessary. Therefore, we should consider only the casts which are directly *causing* a term to get stuck, this is implemented similarly to choosing which hole to instantiate (section 3.5.3).

— *Cast Laziness* — Only casts between *ground* types are checked for consistency. Due to cast laziness (section 3.5.4), some compound terms will be cast between inconsistent types, but *not* placed within a *cast failure*. As before, this issue is ignored due to requiring large changes to Hazel semantics, the evaluation finds it to be a rare occurrence.

— *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.

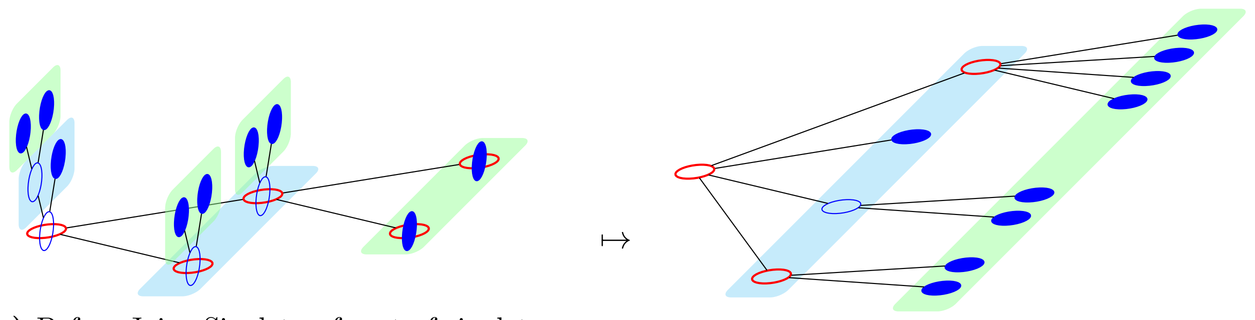
— 3.6.3 — Searching Methods

I implement *four* different search methods, implementing the non-determinism signature specified in section 3.5.1.

— *Depth First Search (DFS)* — Modelling DFS by streams is a typical method [**ListOfSuccess**]: implementing choice and conjunction via appending, and `wrap` being identity function (no internal tree structure needed).

— *Breadth First Search (BFS)* — Breadth first search represents forests by sequences of sequences [**BFSCombinators**], with the nth inner sequences representing every solution in the nth level of the trees in the forest. Then, `choice` concatenates each level, and `wrap` conses the empty list, pushing every solution one level down.

The other monadic operators are complex, but `join : t(t('a)) => t('a)` can be visualised in fig. 3.21: for simplicity showing joining trees of trees, which is extended to forests by choicing the trees in each inner forest.



(a) Before Join: Singleton forest of singleton forests (list of lists of lists of lists). Rectangles are the inner lists of the forests.

(b) After Join: Flattened into a single forest (list of lists) by rotating vertical trees into horizontal plane. Solid circles are the solutions actually stored in the BFS lists.

Figure 3.21: BFS Join

This join and mapping can be defined lazily; the standard definition for `bind` from the paper [**Bunches**] does not work easily in OCaml due to its strictness.

— *Bounded Depth First Search (BDFS)* — BFS is fair, avoiding non-termination (for finite branching factor), but it has *exponential* space complexity in the depth of the level being explored [**NorvigAI**]. In comparison, DFS only requires space linear in the depth explored.

Bounded DFS, performs successive depth-bounded depth first searches, retaining low space complexity of DFS while avoiding unfairness. Previous solutions are repeatedly explored, but this does not increase the already exponential time complexity of the search.

Spivey [**Bunches**] represent this by functions `Int => ['a, Int]`, calculating solutions within an integer depth bound. Solutions are tagged with their remaining depth budgets, which allows keeping only the fringe solutions (zero budget) upon each iteration, so the same solution will not appear twice.

I augment this into an `Bound.t => (Bool, ['a, Bound.t])`, implemented as a functor allowing customised depth bound increases. The boolean is false only if no search path reaches the depth bound, in which case the trees have been fully searched and the algorithm can be terminated.

— *Interleaved Streams (IDFS)* — Streams, but with choice and conjunction via interleaving also ensures fairness. This works even for trees with infinite branching factor. However, interleaving has a linear space complexity, resulting in exponential space complexity as with BFS.

— 3.6.4 — User Interface

The user can switch the search procedure on via a switch in the settings.

— 3.7 — Repository Overview

— 3.7.1 — Branches

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

— 3.7.2 — Hazel Architecture

— 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.

— 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 (section 4.5):

— 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) via the *bechamel* library [**Bechamel**].

— *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.[3]

[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).

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

— *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 these?*

This section provides *critical* arguments on usefulness or effectiveness, which are *evidenced* by quantitative data. Differing implementations and *subsequent improvements*[4] 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.[5]

— 4.4 — Hypotheses

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

— *Search method space requirements:* — The space requirements for DFS and 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 [**SmallScopeHypothesisOrigination**] 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 [**SmallScopeHypothesis**] and answer-set programs [**SmallScopeHypothesisAnswerSet**]. 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 [**HazelCorpus**].

— 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 [**OCamlCorpus**]. 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 [**HazelOfOCaml**]. 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.[6]

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

[4]Which were then implemented and analysed.

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

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

	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

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[7] shown in fig. 4.1.

— 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, capable of interactive use 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, these results are found given a 30s time limit. Micro-benchmarking 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[8] than bounded DFS (BDFS) and DFS, while DFS is the fastest[9].

However, full averages are not a fair test as the sets of programs averaged over are different, for example, BFS only happens to succeed 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.

— 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

[7]When using normal, deterministic, evaluation.

[8]Although, IDFS efficiently keeps most memory allocated short lived, being in the *minor* heap.

[9]Having the least overhead.

Averages	unit	DFS	BDFS	IDFS	BFS
		Implementations			
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

Ratios vs. DFS	BDFS	IDFS	BFS
	Implementations		
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

expressions after type checking.

Figure 4.4 shows that both type and cast slices are generally small. In particular, the proportion of the context[10] 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. As hypothesised, both of which are smaller on average (fig. 4.5) than type slices. Therefore, casts can more precisely point to which part of an expressions type caused them.

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

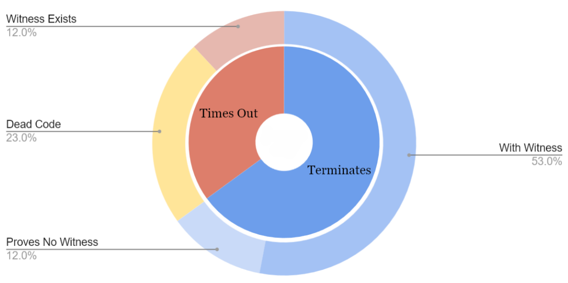
[10]Calculated by close approximation by the *program size*. As each program in the corpus is just one definition. Calculating the context itself is non-trivial.

Averages		**Subdivisions**			
		Ok		**Errors**	
	unit	from	to	from	to
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)					

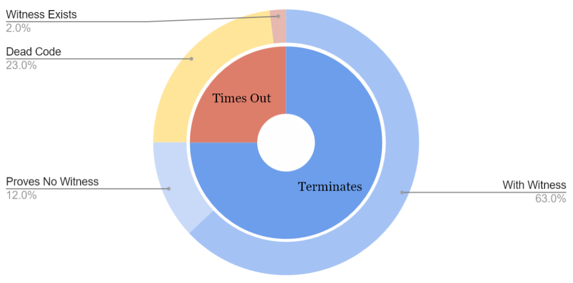
Figure 4.5: Effectiveness: Cast Slices

— 4.7.2 — Search Procedure

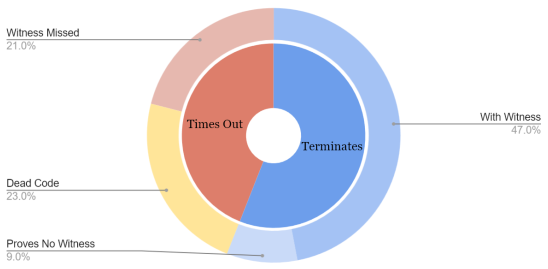
— Witness Coverage —



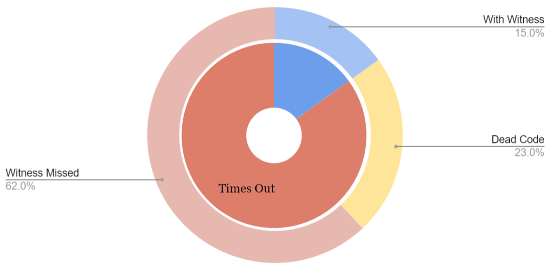
(a) DFS



(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 for BDFS 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[11]; the bugs were excluded from the results. This gives only 2% of cases where BDFS failed to actually find an *existing* witness; DFS and IDFS also meet the goal of 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 linear correlation (Pearson correlation coefficient = 0) 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 often highlights large, verbose slices, including many irrelevant branches, mostly justifying subsumption in subterms, which never relates directly to type errors. Therefore, we use synthesis and analysis slices to explain static errors more concisely. Users can still select subterms to inspect their type consistency if needed.

Type slicing theory (section 3.1) requires highlighted code to form valid expressions or contexts, though some highlighted parts, like unused or dynamic bindings, don't affect types and can be omitted. This motivated the use of unstructured (ad-hoc) slices (**??**).

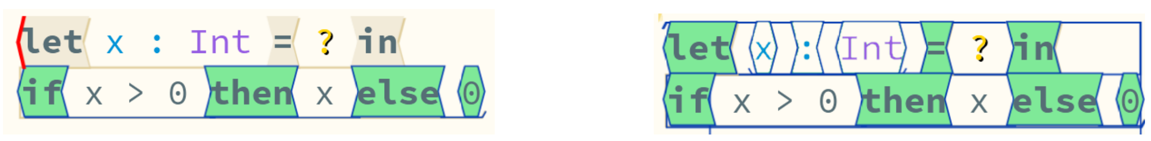
In fig. 4.8, a bound integer `x` is only used in one branch. Since the other branch can already determine the type of the conditional, the `x` and `let` are excluded from the slice. Though the whole program is selected,[12] the let expression is omitted. A contribution slice would include everything,[13] making it even more verbose.

— *Error Slices* —

Type errors arise from inconsistencies between a term's analysis and synthesis slices, or across synthesising branches. Understanding the error requires comparing all the slices involved.

Some type parts may agree, hence another form of type joining was introduced to isolate only the inconsistent parts. For instance, in fig. 4.9, differences like `(Int, Int)` vs `(Int, String)` highlight only the mismatched sub-slices.

[11] Being also present on the main branch, not just in my code.
[12] See the red cursor.
[13] Both branches must be shown, as `x` needs to type-check.



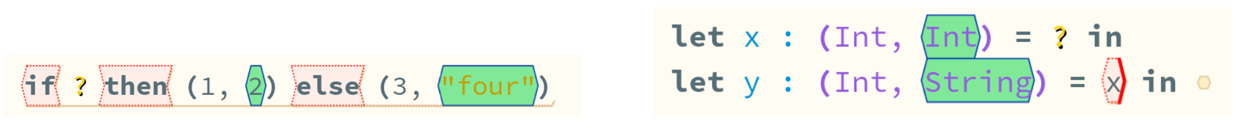
```
let x : Int = ? in
if x > 0 then x else 0
```

```
let x : Int = ? in
if x > 0 then x else 0
```

(a) Ad-hoc Slice: Let expression omitted

(b) Ordinary Slice

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



```
if ? then (1, 2) else (3, "four")
```

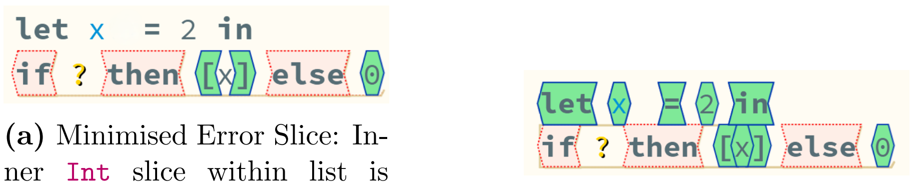
```
let x : (Int, Int) = ? in
let y : (Int, String) = x in
```

(a) Partially Inconsistent Branches

(b) Partially Inconsistent Expectations

Figure 4.9: Error Slices

Compound type inconsistencies necessarily differ at the outermost constructor (e.g., `List` vs `Int`), being the *primary* cause of the error. Deeper inconsistencies are not those causing the type error. Therefore, we could extract only the slice on the outermost constructor. These minimised slices (fig. 4.10) are significantly (avg. 3x) smaller than full combined slices (fig. 4.4). This ratio may grow with program complexity.[14] This is now the default UI for type errors (with colours to distinguish the two parts).



```
let x  = 2 in
if ? then [x] else 0
```

```
let x  = 2 in
if ? then [x] else 0
```

(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**], where statics are recalculated on edits, even cursor movement. While slice-based type checking is fast, it's less ideal for such interactive use in large programs.

Hazel ensures efficient edits[15] via a zipper data structure [**Zipper**, **OneHoleContext**]. Extending this to the AST and typing info could enable edits to only locally update the type information. However, some edits (e.g. adding a binding) can cause non-local, more extensive, type recalculation, but, these are rare. Such a system would require a major rewrite of Hazel's statics, with benefits beyond just type slicing.

— 4.8.3 — Static-Dynamic Error Correspondence

A static type error will place a term inside a cast error during elaboration, which can be associated with a dynamic error whose cast error is dependent on (decomposed from, **??**) this original failed cast. This works well for *inconsistent expectations* static type errors.

However, for inconsistent branches, no direct cast failure occurs until both are evaluated in a shared static context. The search might find such cases, but linking them back to the static

[14]The dataset had few partially inconsistent cases, where the biggest savings occur.

[15]In constant time.

error is harder. Still, since elaboration adds casts to each branch, these can be tracked with the error.[16]

— 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 lies in unreachable code under all valid type instantiations, subdivided into:
  - Pattern Cast Failure: Error within a pattern matching branch, making the branch unreachable. These are detectable by the extended pattern directed instantiation algorithm (section 3.5.4).[17]
  - Unbound Constructor: Attemping to match an unbound constructor. Also detectable with the extended instantiation.
  - Wildcards: Erroneous code bound to the inaccessible wildcard pattern: `string`.
  - Non-Trivial: Less easily detectable. One example exhibited this, infinitely recursing for all inputs.
- Hazel Bugs: Unboxing bugs present in the main branch (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`.

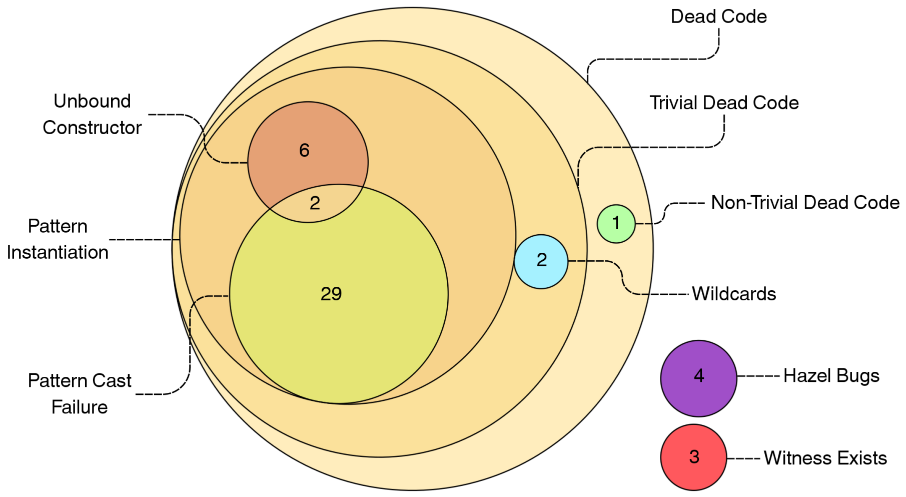


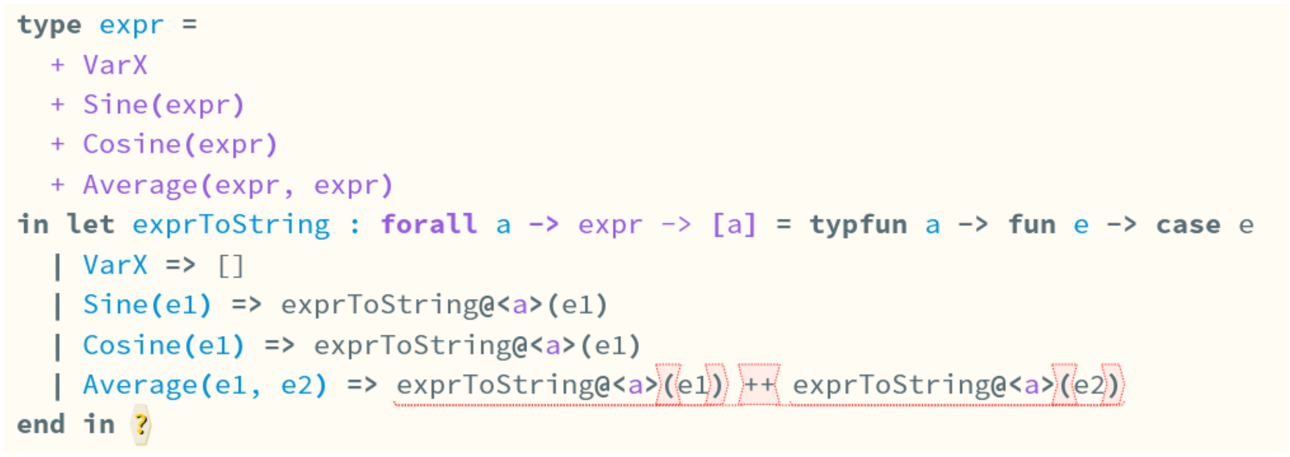
Figure 4.11: Distribution of Failed Program Classes

— Non-Termination, Unfairness, and Search Order —

DFS is fast but prone to non-termination, giving it lesser coverage than BDFS (fig. 4.6). If evaluation infinitely loops, so will DFS without exploring other instantiations. It is also unfair, it may never try some instantiations, e.g. instantiating `(?,?)` will yield `(0, 0)`, `` `Typ(...) ``, etc., but never `(1, 0)` (fig. 4.13). This affected 10% of cases.

[16]But we cannot assume the inconsistent branches caused the error; it might instead be the context using the value that is incorrect.

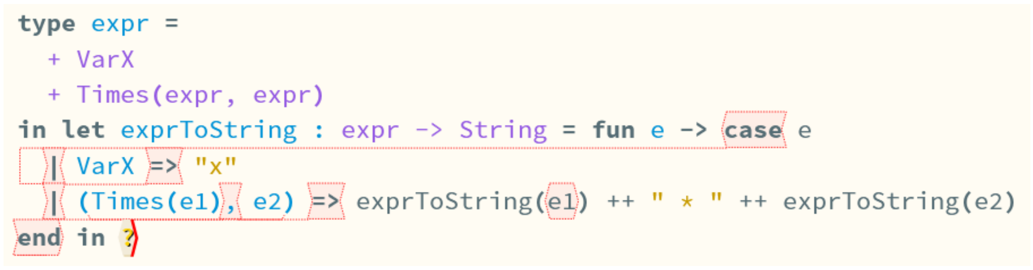
[17]The inconsistent patterns would be attempted, and subsequently reduced to expression cast failures.



```
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.

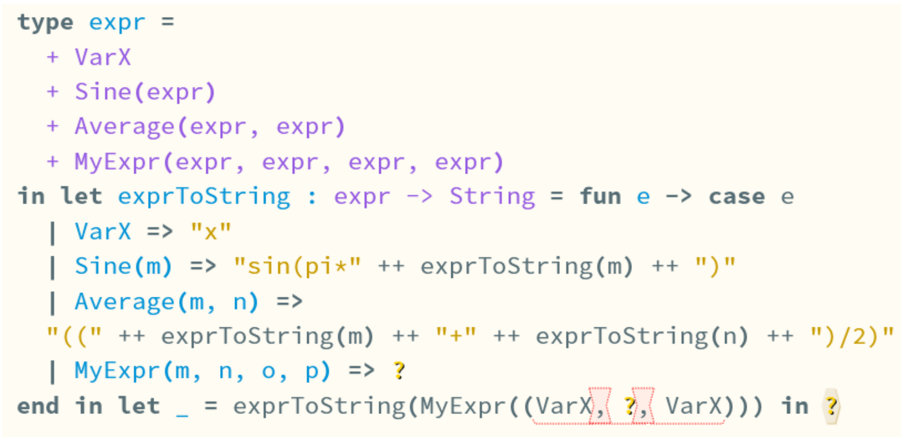
(a) Witness Exists: prog2270.typed.hazel



```
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

To address this, BDFS, BFS, and IDFS were implemented, with BDFS performing best due to its preference for evaluation and low memory use (fig. 4.2).

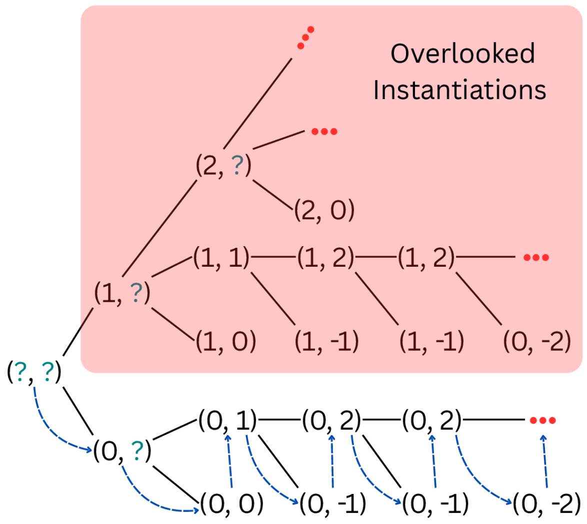


Figure 4.13: Non-exhaustive Instantiations in DFS

BFS and IDFS would do significantly better if evaluation were less frequently wrapped deeper in the search tree (reducing the *cost* of evaluation). Choosing how frequently to wrap evaluation is a delicate balancing act between finding errors deeper in evaluation vs. attempting more instantiations.

— 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. 12 (32%) of the dead code classified had a nested error within a branch, unreachable due to a pattern error[18] within the branch pattern. Even if witnesses are found for these branch errors, the nested errors remain hidden.

— Dynamically Safe Code —

Some static errors is inherently dynamically safe, having no witness, e.g. `if true then 0 else "str"`. The search procedure often proves this by running out of instantiations: BDFS proved no witness 12% of the time.

However, in general, safe code may lead to non-terminating searches, endlessly generating witnesses. So, a timeout does not necessarily imply a witness was found, or that none 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 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.

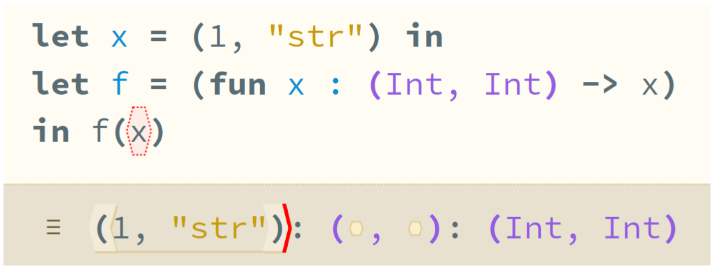
Hazel treats casts lazily, deferring cast transitions on compound data until their elements are used (e.g., tuple elements). Therefore, these elements cannot be instantiated until the

[18]Unbound constructor or inconsistent expectations.

data is deconstructed. Further, Hazel cannot detect cast errors between non-ground compound types (e.g., `[Int]` vs `[String]`), therefore if such compound type's elements are unused, the errors cannot be witnessed. These errors are uncommon, none appeared in the search corpus.

Addressing this would require eager cast semantics, as has been previously explored for dynamic and gradual type systems [**EagerCasts**, **GradualEagerCasts**].[19] Alternatively, more sophisticated instantiation could track hole types of compound data elements using the outer casts, but this still needs eager error checks.

Eager casts would catch dynamic errors earlier. For example, fig. 4.14 shows a cast inconsistency undetected (by lazy casts) at runtime until the tuple element is accessed.



```
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 especially impacts IDFS and BFS, who prioritise instantiation over evaluation. This leads to high memory use and hinders evaluation from progressing far enough to detect errors, even when valid witnesses have been instantiated.

Further, some errors only arise on very specific inputs, e.g. `(23, 31)` and `Mode.t` in fig. 4.15. Directing instantiation to maximise code coverage earlier could find such errors attempting fewer instantiations.

— 4.8.5 — Improving Code Coverage

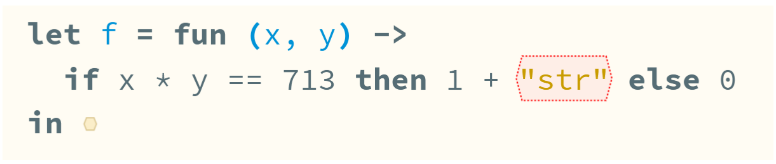
Of the program classes lacking type error witnesses (section 4.8.4), only 3 had concrete witnesses missed, each due to instantiations happening not to execute the erroneous branch.

These often require specific, interdependent inputs, so are less likely to instantiate. These errors are harder for programmers to detect or understand. Understanding the error might require recognizing input interdependencies.

Intelligently directing hole instantiations to better cover the code would help. Symbolic execution and program test generation are well-researched areas having explored this, producing constraint solvers and translation to theories for SMT solvers.

A pattern-directed instantiation as described in section 3.5.4 would discover 2 of the 3 missed witnesses by BDFS. Extended symbolic execution could deal with (`Int` and `Float`) inequalities and conditionals.

[19] Often referred to 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

Still, BDFS retains a very high coverage over the search procedure (97%), 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 [**MarkedLocalisation**] to debug the errors.

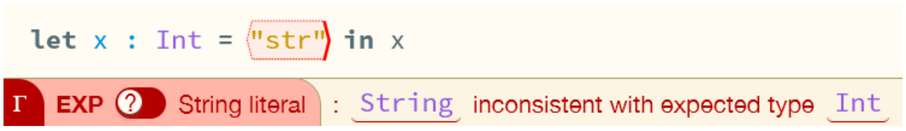
— 4.9.1 — Interaction with Existing Hazel Type Error Localisation

Hazel has three error types addressed by this project:[20] *inconsistent expectations* (analytic and synthetic types are inconsistent), *inconsistent branches* (branches or list element types are inconsistent), and *inexhaustive matches*.

Section 4.8.3 showed how witnesses can be associated to inconsistency errors. Generating examples for inexhaustive match statements was not a core aim, but witnesses for this can be found. Static generation of such examples is already a (mostly)[21] solved problem [**PatternMatchingWarnings**], and is under development for Hazel. Pattern matrices are preferable (faster), and would aid directing hole instantiation (section 4.8.5).

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* 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.

When errors arise from the programmer misunderstanding the program types, error localisation can be inaccurate (due to assuming different types to the programmer's expectations). The context inspector (fig. 4.16) clarifies what assumptions the system makes while slicing and witnesses can explain *why*.



```
let x : Int = "str" in x
```

Γ EXP ? String literal : String inconsistent with expected type Int

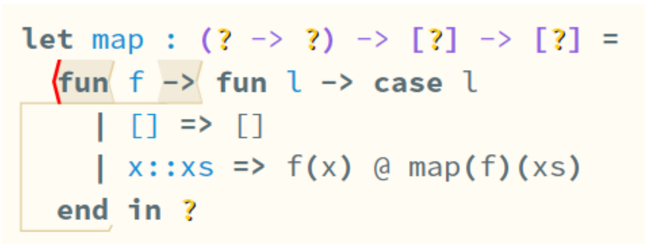
Figure 4.16: Selected Static Error described by Hazel Context Inspector

When the programmer and system agree on the types, bidirectional typing generally localises the error(s) well [**BidirectionalTypes**, **MarkedLocalisation**]. However, there is not always enough static information to even recognise errors. The search procedure tests such code for such type errors automatically.

Alternatively, annotations can be added to locate the errors. However, this is time-consuming, many annotations may be required to detect the error statically. It can make sense to use the search procedure to check for errors more quickly. Figure 4.17 shows a mostly annotated map function, still not providing sufficient type information to detect the error statically.

[20]Other errors: syntax errors, unbound variables, label errors etc. are out of scope

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

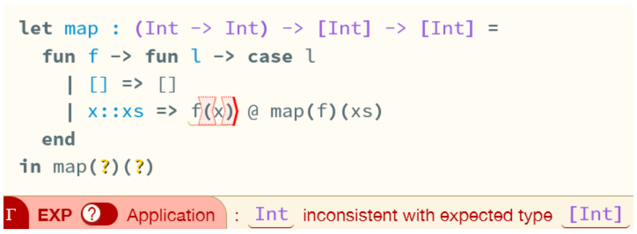


```
let map : (? -> ?) -> [?] -> [?] =
  fun f -> fun l -> case l
    | [] => []
    | x::xs => f(x) @ map(f)(xs)
  end in ?
```

Figure 4.17: Partially Annotated Program misses Error: Concatenation (`@`) used instead of cons (`::`)

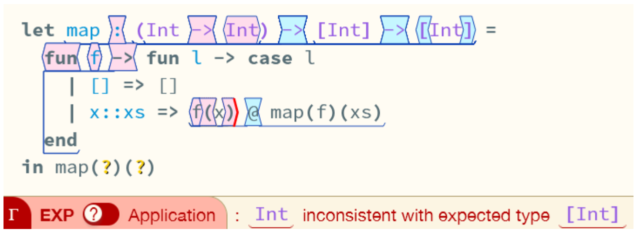
— 4.9.2 — Examples

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[22] is expected (being an argument of list concatenation). See fig. 4.18.



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

EXP ? Application : Int inconsistent with expected type [Int]



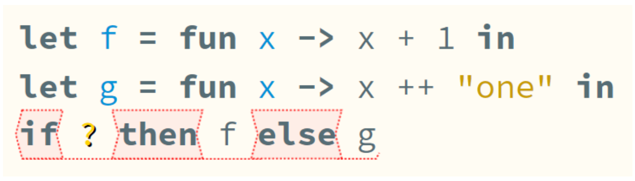
```
let map : (Int -> Int) -> [Int] -> [Int] =
  fun f -> fun l -> case l
    | [] => []
    | x::xs => f(x) @ map(f)(xs)
  end
in map(?)(?)
```

EXP ? Application : Int inconsistent with expected type [Int]

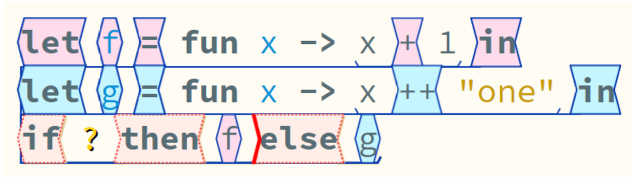
(a) Error **(b)** With Error Slice

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, and conflicting addition and string concatenation operators.



```
let f = fun x -> x + 1 in
let g = fun x -> x ++ "one" in
if ? then f else g
```



```
let f = fun x -> x + 1 in
let g = fun x -> x ++ "one" in
if ? then f else g
```

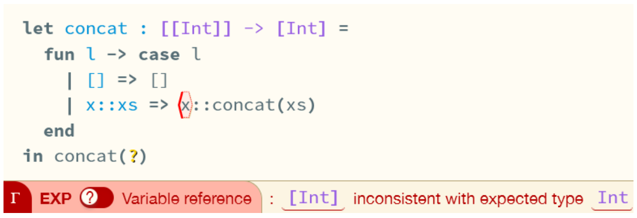
(a) Error **(b)** With Error Slice

Figure 4.19: Example: Type Slice of Inconsistent Branches

Not every static type error slice is concise or obvious. I demonstrate this by returning to the example from the introduction (fig. 4.20). The user mistakes list cons `::` for list concatenation `@`. The type slice does highlight the `::` and annotation enforcing `x` as an `Int`, but they might not understand why if they still expect `::` to perform concatenation. Additionally, there is considerable noise from the synthesis slice distracting from this. A type witness demonstrates concretely how this program goes wrong: that the lists are not being concatenated. The user can cycle through increasingly larger witnesses[23]. Finally, the cast slice to `Int` concisely retrieves relevant part of the original type slice.

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

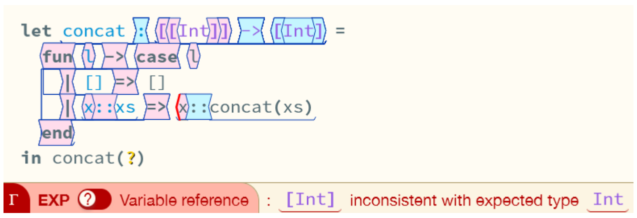
[23]Looking at larger witnesses can be useful to spot pattern with what is happening overall, i.e. that the lists are being consed, not concatenated.



```
let concat : [[Int]] -> [Int] =
  fun l -> case l
    | [] => []
    | x::xs => x::concat(xs)
  end
in concat(?)
```

Γ EXP ? Variable reference : [Int] inconsistent with expected type Int

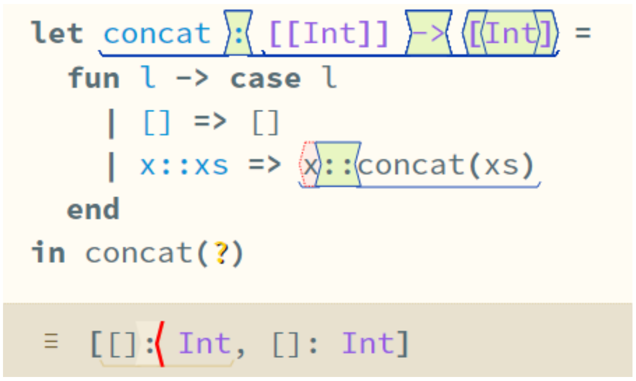
(a) Error: Mistaken Cons for Concat



```
let concat : [[Int]] -> [Int] =
  fun l -> case l
    | [] => []
    | x::xs => x::concat(xs)
  end
in concat(?)
```

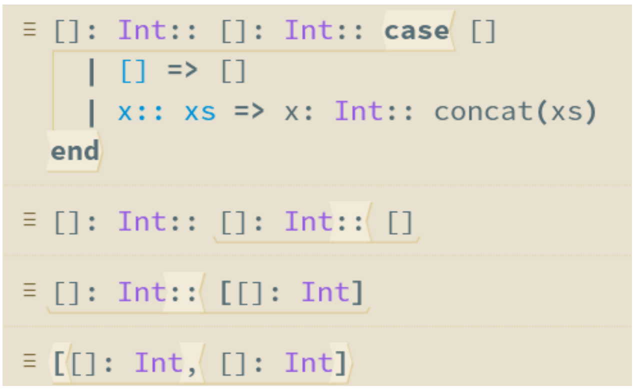
Γ EXP ? Variable reference : [Int] inconsistent with expected type Int

(b) Verbose Type Slice



```
let concat : [[Int]] -> [Int] =
  fun l -> case l
    | [] => []
    | x::xs => x::concat(xs)
  end
in concat(?)

≡ [[]: Int, []: Int]
```

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



```
≡ []: Int:: []: Int:: case []
    | [] => []
    | x:: xs => x: Int:: concat(xs)
  end

≡ []: Int:: []: Int:: []

≡ []: Int:: [[]: Int]

≡ [[]: Int, []: Int]
```

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

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

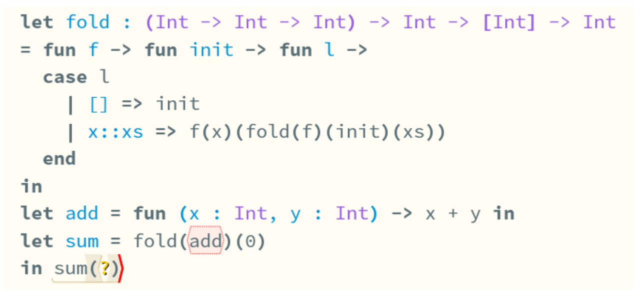
Another, more subtle, example is confusing curried and non-curried functions: fig. 4.21 implements a `fold` function taking curried functions, but tries to apply an uncurried `add` function to sum int lists. This example shows how a single error can result in *multiple* cast errors, each having a more 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

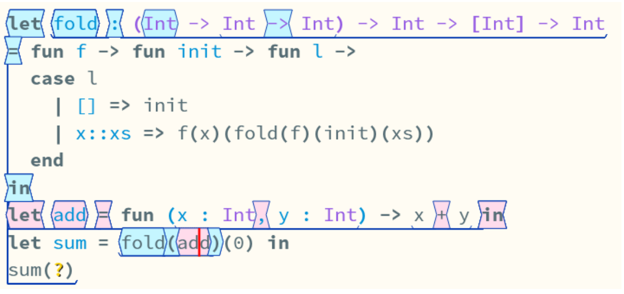
As designing an intuitive UI was not a core goal, using the features can feel awkward or unintuitive. Below are proposed improvements, which the architecture of both Hazel and the new features can easily support:

- Display type slices only *upon request* using the Hazel context inspector, showing the *analysing* and *synthesising* types of the selected expression, see fig. 4.16.
- Allow users to deconstruct type slices to query specific parts, e.g. select the just the sub-slice explaining the *return type* of a function or just the function arrow part. This could be done by selecting the type parts in the context inspector. This interaction would help users really understand how code comes together to define it's types.
- Visualise graphs of cast dependencies, showing the *execution* context leading to a cast error, summarising more concisely than full evaluation traces.



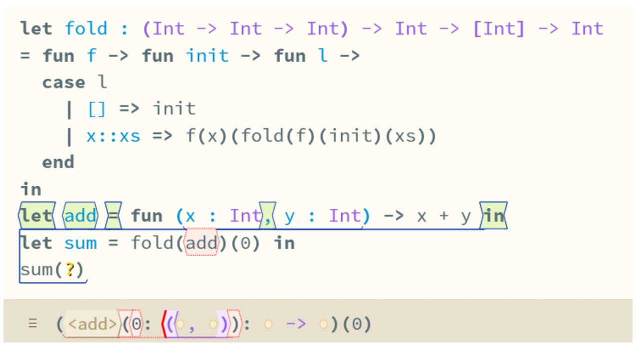
```
let fold : (Int -> Int -> Int) -> Int -> [Int] -> Int
= fun f -> fun init -> fun l ->
  case l
    | [] => init
    | x::xs => f(x)(fold(f)(init)(xs))
  end
in
let add = fun (x : Int, y : Int) -> x + y in
let sum = fold(add)(0)
in sum(?)
```

(a) Confused uncurried for curried `add`



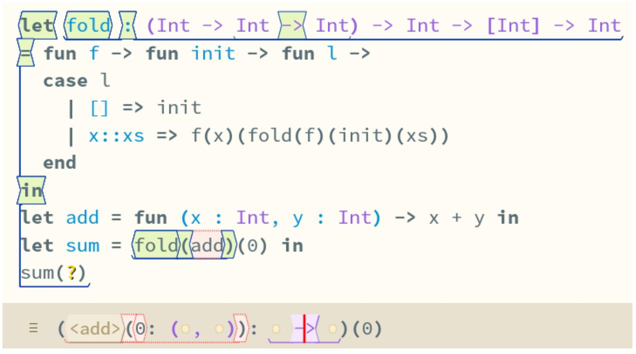
```
let fold : (Int -> Int -> Int) -> Int -> [Int] -> Int
= fun f -> fun init -> fun l ->
  case l
    | [] => init
    | x::xs => f(x)(fold(f)(init)(xs))
  end
in
let add = fun (x : Int, y : Int) -> x + y in
let sum = fold(add)(0) in
sum(?)
```

(b) Somewhat Verbose Type Slice



```
let fold : (Int -> Int -> Int) -> Int -> [Int] -> Int
= fun f -> fun init -> fun l ->
  case l
    | [] => init
    | x::xs => f(x)(fold(f)(init)(xs))
  end
in
let add = fun (x : Int, y : Int) -> x + y in
let sum = fold(add)(0) in
sum(?)

≡ (<add>(0: ((◦, ◦)): ◦ -> ◦)(0)
```

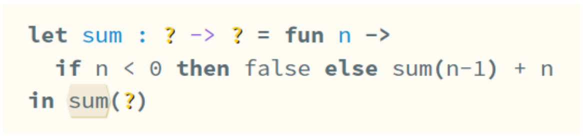
(c) Witness (`[0]`) expects input to `add` to be a tuple



```
let fold : (Int -> Int -> Int) -> Int -> [Int] -> Int
= fun f -> fun init -> fun l ->
  case l
    | [] => init
    | x::xs => f(x)(fold(f)(init)(xs))
  end
in
let add = fun (x : Int, y : Int) -> x + y in
let sum = fold(add)(0) in
sum(?)

≡ (<add>(0: (◦, ◦)): ◦ -> ◦)(0)
```

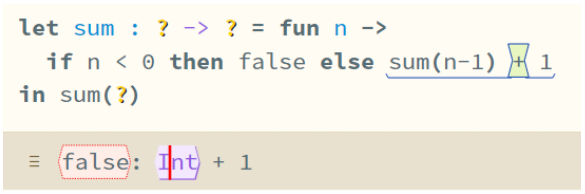
(d) Witness (`[0]`) expects output of `add(0)` to be a function

Figure 4.21: Example: Subtle Currying Error



```
let sum : ? -> ? = fun n ->
  if n < 0 then false else sum(n-1) + n
in sum(?)
```

(a) Error



```
let sum : ? -> ? = fun n ->
  if n < 0 then false else sum(n-1) + 1
in sum(?)

≡ false: Int + 1
```

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

Figure 4.22: Example: Searching for Error in Dynamic Code

- Provide a UI for the search procedure's execution traces and instantiations, integrated with Hazel's trace visualiser. This could include trace compression for better readability (e.g., skipping irrelevant function calls).
- Implement key bindings to cycle through indeterminate evaluation paths more quickly.

— 5 —
Conclusions

This project aimed to enhance type error debugging in Hazel by implementing two novel features: *type slicing* and *cast slicing*. These features were successfully formalised and implemented, with further variations explored and incorporated.

All core goals and extensions set out in section 2.3 were achieved except for the low priority extensions, of which extended pattern instantiation was partially implemented.

Further, type slicing proved more expressive than initially planned, applying to *all* expressions, *not just errors*. Therefore, it can also be used to learn and build understanding on how Hazel's type system works, not just for debugging errors.

Additionally, an indeterminate evaluation framework (section 3.5) was built, greatly generalising the search procedure to allow multiple search orderings (DFS, BFS, IDFS, BDFS, and extensible to future methods). And, searching for different classes of results, e.g. concrete values, or derived results like sizes of expressions or variable substitution steps.

— 5.1 — Further Directions

This section presents some *extensions* to improve the features as justified by the evaluation (section 4.8 & 4.9) alongside other interesting applications.

— 5.1.1 — UI Improvements, User Studies

The current UI can be improved as suggested in Section 4.9.3. Once these are implemented, a user study could assess the real world effectiveness of type slicing and cast slicing in improving understanding of the type system, and locating errors.

— 5.1.2 — Cast Slicing

Cast slicing only propagates type slice information throughout evaluation. As demonstrated, this is useful for debugging. But, it provides no slicing on execution properties of the program. Extended slicing methods could, for example, provide a minimal program which evaluates to the *same* cast around the *same* value. This could build on traditional *dynamic slicing* methods [**DynProgSlice**, **FunctionalProgExplain**], and find use also in debugging *semantic* errors.

— 5.1.3 — Property Testing

Indeterminate evaluation provides a way to generate inputs to function. So, it could be used for property testing, generating inputs to functions and testing expressions. A framework similar to SmallCheck [**SmallCheck**] or QuickCheck [**QuickCheck**] could be implemented. Being part of the evaluator, intermediate expressions and execution properties could also be tested.

— 5.1.4 — Non-determinism, Connections to Logic Programming

Non-deterministic evaluation could be harnessed to implement non-deterministic constructs (e.g. a choice operator) for Hazel.

Treating holes as unknowns and instantiating lazily is reminiscent of *free logic variables* in functional logic programming languages [**FunctionalLogicProgramming**], e.g. in *Curry* [**CurryLang**]. Adding unification [**UnificationSurvey**] would allow full logic programming in Hazel. A needed-narrowing evaluation strategy [**NeededNarrowing**] would be an significantly more efficient than the current lazy non-deterministic instantiation. Equally, unification and needed narrowing could be applied to generating type witnesses.

— 5.1.5 — Symbolic Execution

The search procedure cannot always find witnesses, usually when branches containing the error are missed during the search. This problem has been extensively researched for automated test generation and program verification, often using symbolic execution [**SymbolicExecutionSurvey**]. Constraints along execution paths can be modelled via satisfiability modulo theories (SMTs) [**SMTs**], for which there exist many efficient solvers [**SMTSolver**].

Indeterminate evaluation can be consider a form of simple symbolic execution, only considering constraints enforced by casts. Section 3.5.4 considered directing instantiation using patterns within match statements.

— *Polymorphism* —

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

— 5.1.6 — Let Polymorphism & Global Inference

Types in globally inferred languages are often more subtle, as there are fewer annotations asserting the programmers intent. Extension to type slicing would be useful in understanding why expressions have their globally inferred types. Further, Seidel et al. provides evidence that the witness search procedure is particularly useful in such languages, aiding in error localisation.

— *In Hazel* —

Global inference is difficult or impossible to combine with complex type systems. Global inference for high rank polymorphism, like Hazel's System-F style polymorphism, is undecidable [**SystemFUndecidable**]).

However, a form of let-polymorphism via principal type schemes [**PrincipleTypeSchemes**] is possible. Intersection this with gradual typing has been explored by Garcia and Cimini [**GradualTI**] and Miyazaki et al. [**DTI**], 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 are relatively easily extended to a let-polymorphic Hazel in the style of Miyazaki et al. However, type slicing now has to consider non-local constraints alongside code which sourced them, differing significantly from bidirectional type slicing. Somewhat similar ideas, but limited only to errors, have been explored in *constraint-based type error slicing* by Haack and Wells [**HaackErrSlice**].

— 5.2 — Reflections

TODO

— 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 [**PracticalFoundations**]. 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 $x_1 : \tau_1, \dots, x_n : \tau_2$. For example the SLTC[2] [**TAPL**] has a typing rule for lambda expression and application as follows:

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

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.

— 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*.

— B.1 — Syntax

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

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

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

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

$$
\text{SNEHole}\ \frac{\Gamma \vdash e \Rightarrow \tau}{\Gamma \vdash \llparenthesis e \rrparenthesis^u \Rightarrow ?}
\qquad
\text{SAsc}\ \frac{\Gamma \vdash e \Leftarrow \tau}{\Gamma \vdash e : \tau \Rightarrow \tau}
$$

$\boxed{\Gamma \vdash e \Leftarrow \tau}$ 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

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

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

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

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

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

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

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

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

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

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

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

Figure B.5: Elaboration judgements

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

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

Figure B.7: Identity substitution and substitution typing

— B.2.3 — Internal Language

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

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

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

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

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

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

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

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

Figure 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

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

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

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

Figure B.9: Final forms

— B.3.2 — Instructions

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

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

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

$$\text{ITCast}\;\frac{\tau \text{ ground}}{d\langle \tau \Rightarrow ? \Rightarrow \tau \rangle \longrightarrow d} \qquad \text{ITCastError}\;\frac{\begin{array}{c} \tau_1 \neq \tau_2 \\ \tau_1 \text{ ground} \qquad \tau_2 \text{ ground} \end{array}}{d\langle \tau_1 \Rightarrow ? \Rightarrow \tau_2 \rangle \longrightarrow d\langle \tau_1 \Rightarrow ? \not\Rightarrow ? \rangle \tau_2}$$

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

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'

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

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.

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

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

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

Figure B.13: Hole substitution

— C —
On Representing Non-Determinism

— High Level Representation —

Other high level representations, besides monads, include:

— Logic Programming DSL: — Languages like Prolog [**Prolog**] and Curry [**CurryLang**] 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 [**NondetDSL**] created a tagless final style [**TaglessFinalDSL**] domain specific language within OCaml. This approach fully abstracts the search procedure from the non-deterministic algorithm constructs.

— Delimited Continuations: — Delimited continuations [**DelimitedControl**, **ShiftReset**] 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 [**EffectsExamples**, **HandlersInAction**], 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]

— 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 [**TaglessFinalDSL**, **NondetDSL**], is very complex. However, this would allow more flexibility in writing non-deterministic evaluation, with some optimisations made automatically by the DSL.

[1] An important dependency of Hazel.

— 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`.

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

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

— E — Slicing Theory

— E.1 — Expression Typing Slices

— E.1.1 — Term Slices

— Syntax —

Extending core Hazel syntax with patterns $pt = \_ \mid x$ where $\_$ is the wildcard pattern (binding the argument to nothing).

Definition 11 (Term Slice Syntax)**.** *Pattern expression slices p:*

$$p ::= \square_{pat} \mid \_ \mid x$$

Type slices υ:

$$\upsilon ::= \square_{typ} \mid ? \mid b \mid \upsilon \to \upsilon$$

Expression slices ς:

$$\varsigma ::= \square_{exp} \mid c \mid x \mid \lambda p : \upsilon.\ \varsigma \mid \lambda p.\ \varsigma \mid \varsigma(\varsigma) \mid (\!|\,|\!)^u \mid (\!|\varsigma|\!)^u \mid \varsigma : \upsilon$$

Note: labels for gaps are generally omitted, being determined from their position.

— Precision Relation —

Definition 12 (Term Precision)**.** *Pattern slices p:*

$$\frac{}{\square_{pat} \sqsubseteq p} \qquad \frac{}{x \sqsubseteq x}$$

Type slices υ:

$$\frac{}{\square_{typ} \sqsubseteq \upsilon} \qquad \frac{}{\upsilon \sqsubseteq \upsilon} \qquad \frac{\upsilon_1' \sqsubseteq \upsilon_1 \qquad \upsilon_2' \sqsubseteq \upsilon_2}{\upsilon_1' \to \upsilon_2' \sqsubseteq \upsilon_1 \to \upsilon_2}$$

Expression slices ς:

$$\frac{}{\square_{exp} \sqsubseteq \varsigma} \qquad \frac{}{\varsigma \sqsubseteq \varsigma} \qquad \frac{p' \sqsubseteq p \qquad \upsilon' \sqsubseteq \upsilon \qquad \varsigma' \sqsubseteq \varsigma}{\lambda p' : \upsilon'.\ \varsigma' \sqsubseteq \lambda p : \upsilon.\ \varsigma}$$

$$\frac{p' \sqsubseteq p \qquad \varsigma' \sqsubseteq \varsigma}{\lambda p'.\ \varsigma' \sqsubseteq \lambda p.\ \varsigma} \qquad \frac{\varsigma_1' \sqsubseteq \varsigma_1 \qquad \varsigma_2' \sqsubseteq \varsigma_2}{\varsigma_1'(\varsigma_2') \sqsubseteq \varsigma_1(\varsigma_2)} \qquad \frac{\varsigma' \sqsubseteq \varsigma \qquad \upsilon' \sqsubseteq \upsilon}{\varsigma' : \upsilon' \sqsubseteq \varsigma : \upsilon}$$

Proposition 5 (Precision is a Partial Order)**.** *Term precision forms a partial order on term slices. That is:*

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

Proof. TODO typeset $\square$

— Lattice Structure —

Proposition 6 (Term Slice Bounded Lattice)**.** *For any term t, the set of slices ς_1 and ς_2 of t forms a bounded lattice:*

- *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, $\square \sqsubseteq \varsigma \sqsubseteq t$.*

Proof. TODO typeset □

— E.1.2 — Typing Assumption Slices

— Typing Assumptions as Partial Functions —

Definition 13 (Typing Assumptions)**.** *A typing assumption function Γ is a partial function from the set of variables $\mathcal{X}$ to types $\mathcal{T}$. A partial function is either defined $\Gamma(x) = \tau$ or undefined $\Gamma(x) = \bot$. It's domain* $\mathrm{dom}(\Gamma)$ *is largest the set of variables $S \subseteq \mathcal{X}$ for which Γ is defined: $\forall x \in S.\ \Gamma(x) \neq \bot$.*

— Typing Assumption Slices —

Definition 14 (Typing Assumption Slices)**.** *A typing assumption slice γ is a partial function from the set of variables $\mathcal{X}$ to type slices υ.*

— Precision —

Definition 15 (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:*

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

Proposition 7 (Precision is a Partial Order)**.** *Typing assumption precision forms a partial order on typing assumption slices.*

Proof. **Typeseting TODO** □

— Lattice Structure —

Definition 16 (Typing Assumption Slice Joins and Meets). *For typing slices γ_1, γ_2, and any variable x:*

- *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, $(\gamma_1 \sqcup \gamma_2)(x) = \gamma_1(x) \sqcup \gamma_2(x)$.*

Proposition 8 (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.*

Proof. Typsetting TODO $\square$

— E.1.3 — Expression Typing Slices

Definition 17 (Expression Typing Slices). *An expression typing slice ρ is a pair ς^γ of expression slice ς and typing assumption slice γ.*

— Precision —

Definition 18 (Expression Typing Slice Precision). *For expression typing slices $\varsigma_1^{\gamma_1}$, $\varsigma_2^{\gamma_2}$:*

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

Proposition 9 (Precision is a Partial Order). *Expression typing precision forms a partial order on expression typing slices.*

Proof. **Typesetting TODO** $\square$

— Lattice Structure —

Definition 19 (Expression Typing Slice Joins and Meets). *For expression typing slices $\varsigma_1^{\gamma_1}$, $\varsigma_2^{\gamma_2}$:*

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

Proposition 10 (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.*

— Type Checking —

Definition 20 (Interpreting Term Slices as Terms). *By replacing gaps in terms with holes, the dynamic type, or wildcard patterns, slices can be interpreted as terms. For gaps:*

$$\frac{}{[\![\square_{typ}]\!] = ?} \qquad \frac{}{[\![\square_{pat}]\!] = \_} \qquad \frac{u \text{ is fresh}}{[\![\square_{exp}]\!] = (\!|\!)^u}$$

Patterns:

$$\frac{}{[\![\_]\!] = \_} \qquad \frac{}{[\![x]\!] = x}$$

Types:

$$\frac{}{[\![\tau]\!] = \tau} \qquad \frac{[\![\upsilon_1]\!] = \tau_1 \qquad [\![\upsilon_2]\!] = \tau_2}{[\![\upsilon_1 \to \upsilon_2]\!] = \tau_1 \to \tau_2}$$

Expressions:

$$\frac{}{[\![e]\!] = e} \qquad \frac{[\![p]\!] = pt \quad [\![\upsilon]\!] = \tau \quad [\![\varsigma]\!] = e}{[\![\lambda p : \upsilon.\ \varsigma]\!] = \lambda pt : \tau.\ e} \qquad \frac{[\![p]\!] = pt \quad [\![\varsigma]\!] = e}{[\![\lambda p.\ \varsigma]\!] = \lambda pt.\ e}$$

$$\frac{[\![\varsigma_1]\!] = e_1 \quad [\![\varsigma_2]\!] = e_2}{[\![\varsigma_1(\varsigma_2)]\!] = e_1(e_2)} \qquad \frac{[\![\varsigma]\!] = e \quad [\![\upsilon]\!] = \tau}{[\![\varsigma : \upsilon]\!] = e : \tau}$$

Definition 21 (Interpreting Typing Assumption slices by Typing Assumptions)**.** *Translated by extension, for typing assumption slice γ:*

$$[\![\gamma]\!](x) = [\![\gamma(x)]\!] \qquad \textit{if } \gamma(x) \neq \bot$$

$$[\![\gamma]\!](x) = \bot \qquad \textit{if } \gamma(x) = \bot$$

Definition 22 (Expression Typing Slice Type Checking)**.** *For expression typing slice ς^γ and type τ. $\gamma \vdash \varsigma \Rightarrow \tau$ iff $[\![\gamma]\!] \vdash [\![\varsigma]\!] \Rightarrow \tau$ and $\gamma \vdash \varsigma \Leftarrow \tau$ iff $[\![\gamma]\!] \vdash [\![\varsigma]\!] \Leftarrow \tau$.*

— E.2 — Context Typing Slices

— E.2.1 — Contexts

Definition 23 (Contexts Syntax)**.** *Pattern contexts – mapping patterns to patterns:*

$$\mathcal{P} ::= \bigcirc_{pat}$$

Type contexts – mapping types to types:

$$\mathcal{T} ::= \bigcirc_{typ} \mid \mathcal{T} \to \tau \mid \tau \to \mathcal{T}$$

Expression contexts – mapping patterns, types, or expression to expressions:[1]

$$\mathcal{C} ::= \bigcirc_{exp} \mid \lambda \mathcal{P} : \tau.\ e \mid \lambda pt : \mathcal{T}.\ e \mid \lambda pt : \tau.\ \mathcal{C} \mid \lambda \mathcal{P}.\ e \mid \lambda pt.\ \mathcal{C} \mid \mathcal{C}(e) \mid e(\mathcal{C}) \mid e : \mathcal{T} \mid \mathcal{C} : \tau$$

Definition 24 (Context Substitution)**.**

$$\frac{}{\bigcirc_{pat}\{pt\} = pt} \qquad \frac{}{\bigcirc_{typ}\{\tau\} = \tau} \quad \frac{}{\bigcirc_{exp}\{e\} = e}$$

$$\frac{\mathcal{T}\{\tau_1\} = \tau_1'}{(\mathcal{T} \to \tau_2)\{\tau_1\} = \tau_1' \to \tau_2} \qquad \frac{\mathcal{T}\{\tau_2\} = \tau_2'}{(\tau_1 \to \mathcal{T})\{\tau_2\} = \tau_1 \to \tau_2'}$$

$$\frac{\mathcal{P}\{pt\} = pt'}{(\lambda \mathcal{P} : \tau.\ e)\{pt\} = \lambda pt' : \tau.\ e} \qquad \frac{\mathcal{P}\{pt\} = pt'}{(\lambda \mathcal{P}.\ e)\{pt\} = \lambda pt'.\ e}$$

$$\frac{\mathcal{T}\{\tau\} = \tau'}{(\lambda pt : \mathcal{T}.\ e)\{\tau\} = \lambda pt : \tau'.\ e} \qquad \frac{\mathcal{T}\{\tau\} = \tau'}{(e : \mathcal{T})\{\tau\} = e : \tau'}$$

$$\frac{\mathcal{C}\{e\} = e'}{(\lambda pt : \tau.\ \mathcal{C})\{\tau\} = \lambda pt : \tau.\ e'} \qquad \frac{\mathcal{C}\{e\} = e'}{(\lambda pt.\ \mathcal{C})\{\tau\} = \lambda pt.\ e'} \qquad \frac{\mathcal{C}\{e_1\} = e_1'}{(\mathcal{C}(e_2))\{e_1\} = e_1'(e_2)}$$

$$\frac{\mathcal{C}\{e_2\} = e_2'}{(e_1(\mathcal{C}))\{e_2\} = e_1(e_2')} \qquad \frac{\mathcal{C}\{e\} = e'}{(\mathcal{C} : \tau)\{e\} = e' : \tau}$$

The input and output classes of contexts $\mathcal{C}$ will be notated $\mathcal{C} : \texttt{X} \to \texttt{Y}$ for **`Pat`** *(patterns),* **`Typ`** *(types),* **`Exp`** *(expressions).*

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

Definition 25 (Context Composition). *Defined analogously as context substitution, but substituting contexts syntactically, provided the input and output classes match. If $\mathcal{C}_1 : \mathtt{X} \to \mathtt{Y}$ and $\mathcal{C}_2 : \mathtt{Y} \to \mathtt{Z}$ then $\mathcal{C}_2\{C_1\} = \mathcal{C}_2 \circ \mathcal{C}_1 : \mathtt{X} \to \mathtt{Z}$. Equivalence of contexts can be defined syntactically and coincides exactly with an extensional definition.*

Proposition 11 (Context composition is associative). *For all $\mathcal{C}_1 : \mathtt{X} \to \mathtt{Y}$, $\mathcal{C}_2 : \mathtt{Y} \to \mathtt{Z}$, and $\mathcal{C}_3 : \mathtt{Z} \to \mathtt{W}$ then:*

$$(\mathcal{C}_3 \circ \mathcal{C}_2) \circ C_1 = \mathcal{C}_3 \circ (\mathcal{C}_2 \circ C_1)$$

Proof. **Typesetting todo.** □

— E.2.2 — Context Slices

Syntax extended analogously to term slices. Use c to represent context slices.

— Precision —

Definition 26 (Context Precision). *If $c : \mathtt{X} \to \mathtt{Y}$ and $c' : \mathtt{X} \to \mathtt{Y}$ are context slices, then $c' \sqsubseteq c$ if and only if, for all terms t of class $\mathtt{X}$, that $c'\{t\} \sqsubseteq c\{t\}$.*

Proposition 12 (Precision is a Partial Order). *Context precision forms a partial order on context slices.*

Proof. **Typesetting TODO** □

Proposition 13 (Context Filling Preserves Precision). *For context slice $c : \mathtt{X} \to \mathtt{Y}$ and term slice ς of class $\mathtt{X}$. Then if we have slices $\varsigma' \sqsubseteq \varsigma$, $c' \sqsubseteq c$ then also $c'\{\varsigma'\} \sqsubseteq c\{\varsigma\}$.*

— Lattice Structure —

Definition 27 (Context Slice Joins & Meets). *For context slices $c_1 : \mathtt{X} \to \mathtt{Y}$ and $c_2 : \mathtt{X} \to \mathtt{Y}$ and any term t of class $\mathtt{X}$:*

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

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

Definition 28 (Purely Structural Contexts). *Least specific slices of some context, containing only gaps $\square$ and the mark $\bigcirc$. The purely structural context of $\mathcal{C}$ is the unique one that is a slice of $\mathcal{C}$*

$$\mathcal{P}_s ::= \bigcirc_{pat}$$

$$\mathcal{T}_s ::= \bigcirc_{typ} \mid \mathcal{T}_s \to \square \mid \square \to \mathcal{T}$$

$$\mathcal{C}_s ::= \bigcirc_{exp} \mid \lambda\mathcal{P} : \square.\ \square \mid \lambda\square : \mathcal{T}.\ \square \mid \lambda\square : \square.\ \mathcal{C} \mid \lambda\mathcal{P}.\ \square \mid \lambda\square.\ \mathcal{C} \mid \mathcal{C}(\square) \mid \square(\mathcal{C}) \mid \square : \mathcal{T} \mid \mathcal{C} : \square$$

Proposition 14 (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 of $\mathcal{C}$ and top element $\mathcal{C}$.*

Proof. **Type Setting TODO** □

— E.2.3 — Typing Assumption Contexts & Context Slices

Definition 29 (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.*

— Precision —

Definition 30 (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 $f'(\gamma) \sqsubseteq f(\gamma)$.*

Proposition 15 (Precision is a Partial Order). *Typing assumption context precision forms a partial order on typing assumption context slices.*

Proof. **Typesetting TODO** □

Proposition 16 (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 $f'(\gamma') \sqsubseteq f(\gamma)$.*

— Lattice Structure —

Definition 31 (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 2 (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 $\mathcal{F}$.*

— E.2.4 — Context Typing Slices

Definition 32 (Expression Context Typing Slice). *An expression context typing slice p is a pair c^f of a context slice c and typing assumption context slice f.*

— Precision —

Definition 33 (Expression Context Typing Slice Precision). *For expression context typing slices $c_1^{f_1}$, $c_2^{f_2}$:*

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

Proposition 17 (Precision is a Partial Order). *Typing assumption context precision forms a partial order on typing assumption context slices.*

Proof. Analogous to expression typing slices. □

— Composition —

Definition 34 (Expression Context Typing Slice Composition & Application). *For expression context typing slices c^f, $c'^{f'}$, and expression typing slices ς^γ:*

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

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

Proposition 18 (Expression Context Slice Composition is Associative). *For all p_1, p_2, and p_3 then:*

$$(p_3 \circ p_2) \circ p_1 = p_3 \circ (p_2 \circ p_1)$$

Proof. **Typesetting TODO** □

— Lattice Structure —

Definition 35 (Expression Context Typing Slice Joins and Meets). *For expression typing slices $c_1^{f_1}$, $c_2^{f_2}$:*

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

Proposition 19 (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}.*

Proof. **typsetting TODO** □

— Interpreting Context Typing Slices by Contexts and Typing Assumption Contexts —

A $[\![c]\!]$ function can be analogously defined as to expressions, replacing the gaps by $\_$, $?$, $(\!|\,|\!)^u$.

— E.3 — Type-Indexed Slices

— E.3.1 — Type-Indexed Context Typing Slices

Definition 36 (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:

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

Definition 37 (Type-Indexed Context Typing Slice Composition). *For type-indexed context typing slices $\mathcal{S}$ and $\mathcal{S}'$. If $\mathcal{S} = p$ and $\mathcal{S}' = p'$:*

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

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

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

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

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

Proposition 20 (Type-Indexed Composition Preserves Full Slice Composition). *For type-indexed slices $\mathcal{S}$ and $\mathcal{S}'$:*

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

Proof. **typsetting todo** □

— E.3.2 — Type-Indexed Expression Typing Slices

(Overloading the $\mathcal{S}$ notation)

Definition 38 (Type-Indexed Expressions Typing Slices). *Syntactically defined:*

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

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:

$$\overline{\rho} = \rho$$

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

We retain left incremental composition/application as before (but applying to leaves ρ) but do not retain global right composition.

— E.3.3 — Global Application

Global application and reverse application can be defined to allow converting between context and expression slices.

Definition 39 (Reverse Application). *For an expressions slice ρ. Reverse application $|>$ converts type-indexed context typing slices to type-indexed expressions typing slices:*

$$\rho \ |> \ p = p(\rho)$$

$$\rho \ |> \ (p_1 * \mathcal{S}_1 \to p_2 * \mathcal{S}_2) = p_1 * (\rho \ |> \ \mathcal{S}_1) \to p_2 * (\rho \ |> \ \mathcal{S}_2)$$

Proposition 21 (Validity). *If $\mathcal{S}$ is a valid type indexed context typing slice. For all expression typing slices ρ, then $\rho \ |> \ \mathcal{S}$ is a valid type-indexed expression typing slice with the same structure as $\mathcal{S}$.*

Proof. **typsetting TODO** $\square$

Definition 40 (Application). *For a function f from expression typing slices ρ to context typing slices. Application \$ converts type-indexed expression typing slices to type-indexed context typing slices:*

$$f \ \$ \ \rho = f(\rho)$$

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

The contexts p_1, p_2 are eagerly applied down to the leaves ρ, to ensure the validity property.

Proposition 22 (Validity). *If $\mathcal{S}$ is a valid type indexed expression typing slice. For all functions f from expression slices to context slices, then $f \ \$ \ \mathcal{S}$ is a valid type-indexed context typing slice with the same structure as $\mathcal{S}$.*

Proof. **typsetting TODO** $\square$

— E.4 — Checking Contexts

Definition 41 (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:*

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

Definition 42 (Minimally Scoped Checking Context)**.** *For a derivation $\Gamma \vdash e \Leftarrow \tau$, a minimally scoped expression checking context is a checking context of e such that no sub-context is also a checking context.*

Proposition 23 (Minimally Scoped Checking Contexts Forms)**.** *All minimally scoped contexts, have the following forms. Defined by a judgement: $\Gamma \vdash \mathcal{C}^{\mathcal{F}}$ checks e against τ. With the meaning: $\mathcal{C}^{\mathcal{F}}$ is a minimally scoped checking context for $\Gamma \vdash e \Leftarrow \tau$. Defined:*

$$\frac{\Gamma \vdash e \Leftarrow \tau}{\Gamma \vdash (\bigcirc : \tau)^{\mathrm{id}} \textit{ checks } e \textit{ against } \tau} \qquad \frac{\Gamma \vdash e_2 \Leftarrow \tau \qquad \Gamma \vdash e_1 \Rightarrow \tau_1 \qquad \tau_1 \blacktriangleright_{\to} \tau_2 \to \tau}{\Gamma \vdash (e_1(\bigcirc))^{\mathrm{id}} \textit{ checks } e_2 \textit{ against } \tau_2}$$

$$\frac{\Gamma \vdash \mathcal{C}^{\mathcal{F}} \textit{ checks } \lambda x.\ e \textit{ against } \tau \qquad \tau \blacktriangleright_{\to} \tau_1 \to \tau_2}{\Gamma, x : \tau_1 \vdash \mathcal{C}^{\mathcal{F}} \circ (\lambda x.\ \bigcirc)^{\Gamma \mapsto \Gamma \setminus x : \tau_1} \textit{ checks } e \textit{ against } \tau_2}$$

Proof. Verify that each rule is a checking context, this follows very directly from the Hazel typing rules. No rule has a sub-context also being a checking context by induction, with the base cases being trivial: there is only one sub-context for the base case rules, $\bigcirc$, which is by definition not a checking context. □

Proposition 24 (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 $\mathcal{F}(\Gamma') = \Gamma$.*

Proof. Every minimally scoped checking context for e' has a different structure. Hence, only one of these matches with the structure of e' in e. You simply need to verify that one always exists (induction on the typing derivation), and that $\mathcal{F}(\Gamma') = \Gamma$ when $\mathcal{C}(e') = e$. □

— E.5 — Criterion 1: Synthesis Slices

Definition 43 (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 25 (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.*

Proof. Existence follows from bounded lattice structure, uniqueness follows from the uniqueness of typing in Hazel. □

$\boxed{\Gamma \vdash e \Rightarrow \tau \dashv \mathcal{S}}$ e synthesising type τ under context Γ produces minimum type-indexed synthesis slice $\mathcal{S}$

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

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

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

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

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

Figure E.1: *Minimum synthesis slice* calculation

Conjecture 3 (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* $\rho \sqsubseteq \rho'$*.*

— E.6 — Criterion 2: Analysis Slices

Definition 44 (Analysis Slice)**.** *For a term* e *analysing against* τ*:* $\Gamma \vdash e \Leftarrow \tau$*, and a minimally scoped checking context* $\mathcal{C}^{\mathcal{I}}$ *for* e*. An analysis slice is a slice* c^{f} *of* $\mathcal{C}^{\mathcal{I}}$ *which is also a checking context for* e*. That is:*

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

Conjecture 4 (Minimum Analysis Slices)**.** *The minimum analysis slice analysing* e *in a checking context* $\mathcal{C}^{\mathcal{I}}$ *is an analysis slice* $\mathfrak{p}$ *such that for any other any other analysis slice* $\mathfrak{p}'$ *is at least as specific,* $\mathfrak{p} \sqsubseteq \mathfrak{p}'$*. This minimum always uniquely exists.*

TODO

Figure E.2: *Minimum analysis slice* calculation

Conjecture 5 (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*.*

— E.7 — Criterion 3: Contribution Slices

Definition 45 (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':*[2]

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

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:

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

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**

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

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

$$\mathrm{static}(c_1 * \mathcal{S}_1 \to Cs_2 * \mathcal{S}_2,\ c_1' * \mathcal{S}_1' \to Cs_2' * \mathcal{S}_2') = c_1 * \mathrm{static}(\mathcal{S}_1, \mathcal{S}_1') \to c_2' * \mathrm{static}(\mathcal{S}_2, \mathcal{S}_2')$$

— E.8 — Elaboration

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

— F —

Examples of Slicing Logic in Statics

— *Self* —

Every expression construct which can be synthesised has a corresponding function in `Self.re` to construct the slice from it's sub-derivations. 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 (`Int`) 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:

```
[String]
```

Figure F.1: Match Statement `Self.t`

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

— *Mode* —

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*[1] 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, we 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 (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.

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

```
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 F.2: List Literal `Mode.t`

— G —

Hazel Bugs: Unboxing

When a final form (section 2.1.2) has a type, Hazel often needs to extract parts of the term according to the type during evaluation.

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

- A list literal: `Float`.
- A list with casts wrapped around it: `Float⟨[Int]⇒[?]⟩`.
- A list cons with indeterminate tail: `1::2::?`.

Additionally, when the input is not a list at all, it returns `DoesNotMatch`, used in pattern matching. Unboxing makes use of GADTs to allow for varying output type depending on the type that the final form is being unboxed upon.

— *Hazel Unboxing Bugs* —

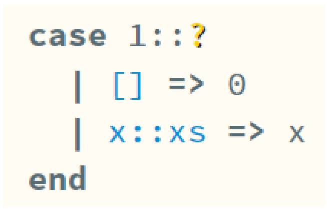
While writing the search procedure I found various unboxing bugs in Hazel. Programs exhibiting these were removed from the evaluation data, whereas some bugs were fixed by myself (hence, not removing from the evaluation data).

For example, there was the following bug, affecting pattern matching. A list cons which has an indeterminate tail would *indeterminately match* with *any* list literal pattern (of *any* length), even when it is known for certain 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.
- *Branch indeterminately matches?* Cannot assume the branch doesn't match so must stop evaluation here. The match statement is then indeterminate.

Figure G.1 demonstrates a concrete example which would get stuck in Hazel, but does *not* need to. I reported and fixed this, with my PR merged into the dev branch.



```
case 1::?
  | [] => 0
  | x::xs => x
end
```

Figure G.1: Pattern Matching Bug

— H —

Extended Pattern Matching Instantiation

This appendix contains notes on instantiating holes in a match expression according also to the structure of the patterns in a match expression.

To implement this, 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.

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 H.1: More Specific Matches

To account for this, we can take ideas from pattern matrix techniques for producing exhaustivity warnings, [**PatternMatchingWarnings**]. That is, we could generate a set of patterns which explicitly do not match any of the previous branches, then intersect those patterns with the current branch, and instantiate according to this intersection.

— I —

Merges

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

Project Proposal

Description

This project will add some features to the Hazel language [**Hazel**]. Hazel is a functional research language that makes use of gradual types to support unusual features such as: holes (code placeholders) to give type meaning to incomplete programs. Importantly for this project, all Hazel programs, even ill-typed or incomplete programs, are evaluable. This allows dynamic reasoning about ill-typed programs via evaluation traces with the potential to improve the user's understanding of *why ill-typed programs go wrong.* See example below:

```
let rec sum : Int -> Int = fun n ->
  if n == 0 then
    true    // Type error statically caught and correctly 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. [**SearchProc**]. The intended use of this is to automatically generate values (for example, function arguments) that cause ill-typed programs to 'go wrong' (lead to a cast error). More specifically, the search procedure can be thought of as evaluating a *special hole* which refines its type dynamically and non-deterministically instantiates itself to values of this type to find a value whose evaluation leads to a *general* cast error – 'general' meaning excluding trivial cast errors such as generating a value that doesn't actually have the refined expected type.

Such a search procedure is undecidable and subject to path explosion, hence the success criteria (detailed below) does not expect witnesses to be provided in general, even if they do exist. Sophisticated heuristics and methods to limit path explosion to support large code samples is not a core goal.

Formal semantics of this procedure and associated proofs is an extension goal, consisting of preservation proofs and witness generality proofs (formalising the notion of generality mentioned previously).

Secondly, *cast slicing* will track source code that contributed to any cast throughout the cast elaboration and evaluation phases. In particular, this allows a cast involved in a cast error relating to a type-error witness to point back to offending code. This is expected in some sense to be similar to blame tracking [**Blame**], error and dynamic program slicing [**ErrSlice**, **DynProgSlice**], although these are not directly relevant for this project.

Work required for the creation of an evaluation corpus of ill-typed hazel programs, requiring manual work or creation of automated translation and/or fuzzing tools, is timetabled.

Starting Point

Only background research and exploration has been conducted. This consists of reading the Hazel research papers [**Hazel**] and various other related research topics including: gradual types, bidirectional types, symbolic evaluation, OCaml error localisation and visualisation techniques.

More research, into the Hazel codebase in particular, and concrete planning is required and is timetabled accordingly.

Success Criteria

Core goals are the minimum expected goals that must be completed to consider this project a success. This corresponds to a working tool for a large portion of Hazel.

Extension goals will be timetabled in, but are relatively more difficult and not required for the project to be considered a success.

First, I give some definitions of terms:

- **Core Calculus** – The formal semantics core of Hazel as referred to by the Hazel research papers [**HazelLivePaper**].
- **Basic Hazel** – A Hazel subset consisting of the core calculus, product and sum types, type aliases, bindings, (parametric) lists, bools, int, floats, strings, and their corresponding standard operations.
- **Full Hazel** – Hazel, including **Basic Hazel** plus pattern matching, explicit impredicative system-F style polymorphism and explicitly recursive types.
- **Core Corpus** – A corpus of ill-typed Hazel programs that are similar in complexity and size to student programs being taught a functional language, *e.g. (incorrect) solutions to the ticks in FoCS*. This will include examples in **Basic** or **Full Hazel** as required.
- **Extended Corpus** – A corpus of ill-typed Hazel programs that are larger in size, more akin to real-world code.
- **Evaluation Criteria** – Conditions for the search procedure to meet upon evaluation:
  1. Must have reasonable coverage – success in finding an *existing* witness which is correct and general.
  2. Must find witnesses in an amount of time suitable for interactive debugging – in-line with build-times for a debug build of existing languages.

Core Goals

- Success criteria for Cast Slicing – Cast slicing must be *correct* (slices must include all code involved in the cast) and work for *all casts*, including casts involved in cast errors. Informal reasoning in evidence of satisfying these conditions is all that will be required.
- Success criteria for the Search Procedure – The procedure must work for **Basic Hazel**, meeting the **Evaluation Criteria** over the **Core Corpus**. Analysis of some classes of programs for which witnesses could not be generated is also expected.

Extension Goals

- Search Procedure Extensions – Support for **Full Hazel** under the same criteria as above.
- Search Procedure Performance Extensions – Meeting of the **Evaluation Criteria** over an **Extended Corpus**
- Formal Semantics – The specification of a formal evaluation semantics for the search procedure over the **Core Calculus**. Additionally, a preservation and witness generality proof should be provided.

Work Plan

21st Oct *(Proposal Deadline)* – 3rd Nov

Background research & research into the Hazel semantics, cast elaboration, type system, and codebase. Produce implementation plan for cast slicing and the search procedure for the **Core Calculus**. This includes an interaction design plan, expected to be very minimal.
Milestone 1: Plan Confirmed with Supervisors

4th Nov – 17th Nov

Complete implementation of Cast Slicing for the **Core Calculus**. Write detailed reasoning for correctness, including plan for **Basic Hazel**. Add unit testing.
Milestone 2: Cast slicing is complete for the ***Core Calculus****.*

18th Nov – 1st Dec *(End of Full Michaelmas Term)*

Complete implementation of the search procedure for the **Core Calculus**.
Milestone 3: Search Procedure is complete for the ***Core Calculus****.*

2nd Dec – 20th Dec

Extension of both cast slicing and the search procedure to **Basic Hazel**.
Milestone 4: Cast slicing & search procedure are complete for ***Basic Hazel***

21st Dec – 24th Jan *(Full Lent Term starting 16th Jan)*

Basic UI interaction for the project. Drafts of Implementation chapter. Slack time. Expecting holiday, exam revision, and module exam revision. Should time be available, the **Formal Semantics** extension will be attempted.
Milestone 5: Implementation chapter draft complete.

25th Jan – 7th Feb *(Progress Report Deadline)*

Writing of Progress Report. Planning of evaluation, primarily including decisions and design of tools to be used to collect/create the **Core Corpus** and planning the specific statistical tests to conduct on the corpus. Collected corpus and translation method will be one of:

1. Manual translation of a small ill-typed OCaml program corpus into ill-typed Hazel.
2. Manual insertion of type-errors into a well-typed Hazel corpus.
3. Collection of a well-typed Hazel corpus.
   Tools: A Hazel type fuzzer to make the corpus ill-typed.

4. Collection of a well-typed OCaml corpus.
   Tools: OCaml -¿ Hazel translator/annotator which works with well-typed OCaml. A Hazel type fuzzer.

5. Collection of an ill-typed OCaml corpus.
   Tools: OCaml -¿ Hazel translator which works with ill-typed OCaml. *This would NOT be expected to be an implicitly typed Hazel front-end which maintains desireable properties like parametricity.*

Milestone 6: Evaluation plan and corpus creation method confirmed with supervisors.
Milestone 7: Underlying corpus (critical resource) collected.

8th Feb – 28th Feb

Implementation of the required tools for evaluation as planned. Some existing code or tools may be re-used, such as the OCaml type-checker.
Milestone 8: ***Core Corpus*** *has been collected.*

1st Mar – 15th Mar *(End of Full Lent Term)*

Conducting of evaluation tests and write-up of evaluation draft including results.
Milestone 9: Evaluation results documented.
Milestone 10: Evaluation draft complete.

16th Mar – 30th Mar

Drafts of remaining dissertation chapters. If possible, collection and evaluation of **Extended Corpus** using the same tools as the **Core Corpus**.
Milestone 11: Full dissertation draft complete and sent to supervisors for feedback.

31st Mar – 13th Apr

Act upon dissertation feedback. Exam revision.
Milestone 12: Second dissertation draft complete and send to supervisors for feedback.

14th Apr – 23rd Apr *(Start of Full Easter Term)*

Act upon feedback. Final dissertation complete. Exam revision.
Milestone 13: Dissertation submitted.

24th Apr – 16th May *(Final Deadline)*

Exam revision.
Milestone 14: Source code submitted.

Resource Declaration

- Underlying Corpus of either: Well-typed OCaml programs, Ill-typed OCaml programs, Hazel programs. For use in evaluation. The required tools or manual translation to convert these into the ill-typed Hazel **Core Corpus** are detailed and allocated time in the timetable.

- Hazel source code. Openly available with MIT licence on GitHub [**HazelCode**].

- My personal laptop will be used for development, using GitHub for version control and backup of both code and dissertation. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure. A backup pc is available in case of such failure.