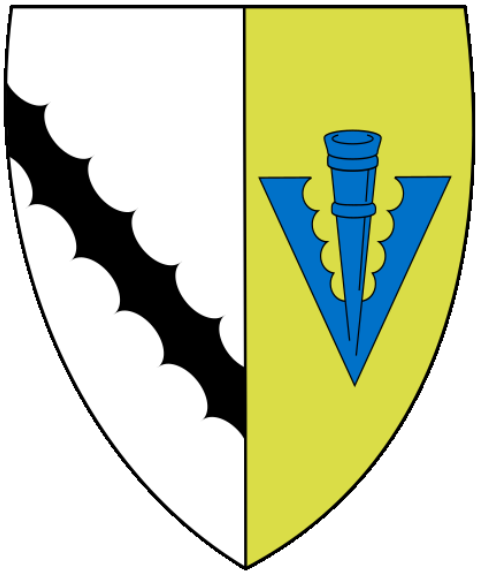
Max Carroll

Type Error Debugging in Hazel

Computer Science Tripos, Part II



Sidney Sussex College College
University of Cambridge
May 7, 2025

A dissertation submitted to the University of Cambridge in partial fulfilment for a Bachelor of Arts

Declaration of Originality

Declaration Here.

Proforma

Candidate Number:	**Candidate Number Here**
College:	**Sidney Sussex College**
Project Title:	**Type Error Debugging in Hazel**
Examination:	**Computer Science Tripos, Part II – 05/2025**
Word Count:	**17488** [1]
Code Line Count:	**Code Count** [2]
Project Originator:	**The Candidate**
Supervisors:	**Patrick Ferris, Anil Madhavapeddy**

Original Aims of the Project

Aims Here. Concise summary of proposal description.

Work Completed

Work completed by deadline.

Special Difficulties

Any Special Difficulties encountered

Acknowledgements

Acknowledgements Here.

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

[2] *Calculation method here*

Contents

Chapter 1

Introduction

Software bugs are an inherent part of programming, often leading to unexpected behaviour and system failures. Debugging these errors is a *time-consuming* process taking between 20-60% of active work time [**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*?

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

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

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

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

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

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

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).

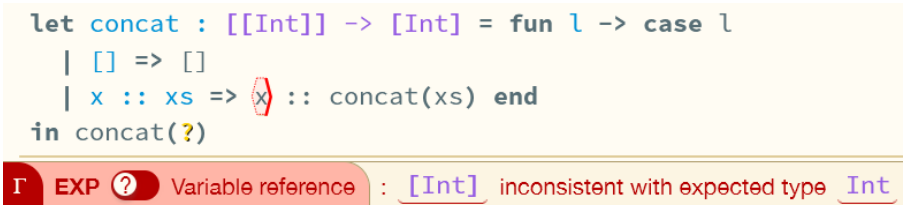


Figure 1.1: A Static Type Error from the Hazel Editor

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.

Chapter 2

Preparation

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

2.1 Background Knowledge

2.1.1 Static Type Systems

A *type system* is a lightweight formal mathematical method which categorises values into *types* and expressions into types that evaluate to values of the same type. It is effectively a static *approximation* to the runtime behaviour of a language. The following sections expect basic knowledge formal methods of type systems in terms of judgements (appendix A reviews this). Note that I will use *partial functions* to represent typing assumption contexts.

Dynamic Type Systems

Dynamic typing has purported strengths allowing rapid development and flexibility, evidenced by their popularity [**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:

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

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

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

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

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

Compound type casts can be broken down during evaluation upon usage of such constructs. For example, applying v to a *wrapped*[3] functions decomposes the cast into casting the applied argument and then the result:

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

Or if f has the dynamic type:

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

The direction of the casts reflects the *contravariance* [**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.

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

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

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

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

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

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

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

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

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

Where $\blacktriangleright_\to$ is a pattern matching function to extract the argument and return types from a function type, used to account for if $\Gamma \vdash e_1 : ?$, where we treat $?$ then as a dynamic function $? \blacktriangleright_\to ? \to ?$. Intuitively, $e_1(e_2)$ has type τ_2' if e_1 has type $\tau_1' \to \tau_2'$ or $?$ (then treated as $? \to ?$), and e_2 has type τ_1 which is consistent with τ_1' and hence is assumed that it can be passed into the function.

But, for evaluation to work the static type information needs to be encoded into casts to be used in the dynamic internal language, for which the evaluation semantics are defined. This is done via *elaboration*, similarly to Harper and Stone's approach to defining (globally inferred) Standard ML [**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:

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

If e_1 elaborates to d_1 with type $\tau_1 \sim \tau_2 \to \tau$ and e_2 elaborates to τ_2' with $\tau_2 \sim \tau_2$ then we place a cast[8] on the function d_1 to $\tau_2 \to \tau$ and on the argument d_2 to the function's expected argument

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

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

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

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

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*

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]

[9] Highlighted in red as an error.

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

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

The core calculus [**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]

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

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

External Language

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

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

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

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

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

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

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

Internal Language

The internal language is non-bidirectionally typed and requires an extra *hole context* Δ mapping each hole *metavariable* u to it's *checked type* τ [13] and the type context Γ under which the hole was typed. Each metavariable context notated as $u :: \tau[\Gamma]$, notation borrowed from contextual modal type theory (CMTT) [**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]:

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

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

Elaboration

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

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

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

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

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

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

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

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

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

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

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

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

[16] Potentially leading to incorrect cast insertion.

Final Forms

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

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

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

Dynamics

A small-step contextual dynamics [**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 values are found by checking the holes *environment*, effectively making a *delayed substitution*. See the following rule:

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

When substituting a matching hole u, we replace it with d and apply substitutions from the environment σ of u to d, after first substituting any occurrences of u in the hole's environment σ. This corresponds to *contextual substitution* in CMTT [**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.

[17] Between function types

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

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

As Hazel is a *pure language*[20] and as holes act as closures, then performing hole substitution is *commutative* with respect to evaluation. That is, filling incomplete parts of a program *before* evaluation gives the same result as filling *after* evaluation then resuming evaluation.

2.1.3 The Hazel Implementation

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

Language Features

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

- **Lists** – Linked lists, in the style of ML. By use of the dynamic type, Hazel can represent *heterogeneous* lists, which may have elements of differing types.
- **Tuples & Labelled Tuples**[22] – Allowing compound terms to be constructed and typed [**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.

Evaluator

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

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

- **Final Form Checker** – Returns whether a term is either: a evaluable expression, a value, or an indeterminate term. Using the evaluator abstraction with this means there is no need to maintain a separate syntactic value checker, instead it is derived directly from the evaluation transitions. Yet, it is still *syntactic* as the implementation does not actually *perform* any evaluation steps which the abstraction presents it with.
- **Evaluator** – Maintains a stack machine to actually perform the reduction steps and fully evaluate terms.
- **Stepper** – Returns a list of *possible*[24] evaluation steps. This allows the evaluation order to be user-controlled in a stepper environment.

Each evaluation method is transformed into an evaluator by being applied to an OCaml *functor* [**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

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

[26] For OCaml developers and functional programmers.

Primarily, the OCaml search procedure for ill-typed witnesses Seidel et al. [**SearchProc**] and the Hazel core language [**HazelLivePaper**] were researched over the preceding summer.

Tools and Source Code

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

2.3 Requirement Analysis

2.4 Software Engineering Methodology

Do the theory first.

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

Talking with devs over slack.

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

2.5 Legality

MIT licence for Hazel.

Chapter 3

Implementation

This project was conducted in *two* major phases:

1. I constructed a core mathematical theory for *type slicing* and *cast slicing* formalising what these ideas actually were and considered the changes to the system presented by Seidel et al. for the *type error witnesses search procedure* to work in Hazel.
2. I implemented the theories, making it suitable for implementation and extending it to the majority of the Hazel language. Further, suitable deviations from the theory, made upon critical evaluation, are detailed throughout.

3.1 Type Slicing Theory

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

The first two criteria attempt to give insight on the structure of the typing derivations, and hence how types are decided. While the third criterion gives a complete picture of the regions of code which contribute to the a term's type.

I would like to stress that the second and third criterion were *very challenging* to formalise, requiring non-obvious mathematical machinery: *context typing slices* (section 3.1.2), *checking contexts* (definition 49), and *type-indexed slices* (section 3.1.3).

3.1.1 Expression Typing Slices

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

Term Slices

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

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

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

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

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

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

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

Type slices υ:[1]

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

Expression slices ς:

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

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

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

Precision forms a partial order [**PartialOrder**], that is, a reflexive, antisymmetric, and transitive relation. Respectively:

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

We also have a *bottom* (least) element, $\square \sqsubseteq \varsigma$ (for all ς). This relation is trivially extended to include *complete terms*[2] e which are *top* elements of their respective chains: if $e \sqsubseteq \varsigma$ then $e = \varsigma$. Sometimes I refer to this relation as ς_1 is a term slice *of* ς_2 iff $\varsigma_1 \sqsubseteq \varsigma_2$.

Lattice Structure: For any *complete term* 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* $\varsigma_1 \sqcap \varsigma_2$. Full details on this are found in the appendix. Note that not every pair of slices has a join: consider 1 and 2. But, every pair does have a meet as for all ς_1, ς_2, $\square \sqsubseteq \varsigma_1$ and $\square \sqsubseteq \varsigma_2$.

Typing Assumption Slices

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

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

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

[2] With no gaps.

[3] Relevant to the given criterion.

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

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

Equally, joins and meet can be extended extensionality:

Definition 3 (Typing Assumption Slice Joins and Meets)**.** *For typing slices* γ_1, γ_2*, and any variable* x*: 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, the slices γ of complete typing assumptions Γ form a bounded lattice. Again, not every pair of slices has a join, as not every type slice has a 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:

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

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

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

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

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

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

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

Definition 6 (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 & Context Slices

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

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

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

Type contexts – mapping types to types:

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

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

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

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

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

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

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

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

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

Joins and meets can be defined via extensionality as before, and we can still form bounded lattices.

Typing Assumption Contexts & Context Slices

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

Definition 9 (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.*

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

[5] The conjectures are heavily suspected but not formally proven yet.

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

Definition 10 (Typing Assumption Context Slice Precision)**.** *If f' and f are typing assumption context slices, then $f' \sqsubseteq f$ if and only if, for all typing context slices γ, that $f'(\gamma) \sqsubseteq f(\gamma)$.*

Again, such functions are monotone:

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

Joins and meets are also defined extensionally:

Definition 11 (Typing Assumption Context Slice Joins & Meets)**.** *For typing assumption context slices f_1 and f_2 and any typing assumption slice γ:*

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

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

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

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

Context Typing Slices

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

Definition 12 (Expression Context Typing Slice Composition & Application)**.** *For expression context typing slices c^f, $c'^{f'}$, and expression typing slices ς^γ:*

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

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

3.1.3 Type-Indexed Slices

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

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

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

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

This section will only consider *context slices*, but term slices are type-indexed in exactly the same way, see the appendix.

The main property that we want these indexed-slices to maintain is that a slice can be *reconstructed* from their sub-parts. *Joining* the sub-slices should produce the slice for the entire type. As sub-slices come from different regions of code, we pair sub-slices with contexts which place the two sub-slices inside the same context, making them join-able.

Definition 13 (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:

$$\overline{p} = p$$

$$\overline{p_1 * \mathcal{S}_1 \to p_2 * \mathcal{S}_2} = p_1 \circ \overline{\mathcal{S}_1} \sqcup p_2 \circ \overline{S_2}$$

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

Definition 14 (Type-Indexed Context Typing Slice Composition)**.** *For type-indexed context typing slices $\mathcal{S}$ and $\mathcal{S}'$. If $\mathcal{S} = p$ and $\mathcal{S}' = p'$:*[6]

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

If $\mathcal{S} = p$ and $\mathcal{S}' = p'_1 * \mathcal{S}'_1 \to p'_2 * \mathcal{S}'_2$:[7]

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

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

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

3.1.4 Criterion 1: Synthesis Slices

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

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

[6] In shorthand, the overline for a base type slice is often omitted.

[7] In shorthand, brackets will be omitted.

Definition 15 (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 2 (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.*

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

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

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

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

$$c \blacktriangleright_{\to} c \to c$$

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

This approach correctly calculates the synthesis slices.

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

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

3.1.5 Criterion 2: Analysis Slices

A similar idea can be devised for type analysis. The way to represent these are *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 `Int` due to the annotation. Therefore, the hole's analysis slice will be:

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

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

In other words, if the context slice was type synthesised, then the marked term would still be *required* to check against `Int`. However, the overall synthesised type may differ.[8]

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

Checking Context

For this, we only want to consider the smallest context scope[9] that enforced the type checking. For example, the below term has 3 annotations, but only the inner one enforces the `Int` type on the integer 1:

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

I refer to this minimum context scope as the *minimally scoped checking context*; checking contexts will always synthesise a type. To give another example, an integer argument's type is enforced by the annotation on the function:

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

Definition 16 (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 17 (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 possible minimally scoped checking contexts can be constructed syntactically via rules. These coincide with the definition above, but is more amenable to performing proofs on. Given in the appendix **ref**.

Sub-terms of a program which are analysed will have unique checking contexts that are also part of the program:

Proposition 3 (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 $\mathcal{F}(\Gamma') = \Gamma$.*

Analysis slices are then a slice of any *minimally scoped checking context*.

Definition 18 (Analysis Slice)**.** *For a term e analysing against τ: $\Gamma \vdash e \Leftarrow \tau$, and a minimally scoped checking context $\mathcal{C}^{\mathcal{F}}$ for e. An analysis slice is a slice c^{f} of $\mathcal{C}^{\mathcal{F}}$ 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* $\Gamma \vdash e \Leftarrow \tau$.

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

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

Conjecture 5 (Minimum Analysis Slices)**.** *The minimum analysis slice analysing e in a checking context $\mathcal{C}^{\mathcal{F}}$ is an analysis slice p such that for any other any other analysis slice p' is at least as specific, $p \sqsubseteq p'$. This minimum always uniquely exists.*

This can again be represented by a judgement read as, *e which type checks against τ in checking context $\mathcal{C}$ has (type-indexed) analysis slice $\mathcal{S}$*:[10]

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

There are two situations which enforce *checking contexts*, of which application is most interesting, calculating the minimum indexed synthesis slice of the context and retrieving just the sub-slice relevant to typing the *argument*:[11]

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \dashv \mathcal{S}_1 \qquad \tau_1 \blacktriangleright_{\to} \tau_2 \to \tau \qquad \mathcal{S}_1 \blacktriangleright_{\to} \mathcal{S}_2 \to \mathcal{S} \qquad \Gamma \vdash e_2 \Leftarrow \tau_2}{\Gamma; e_1(\bigcirc) \vdash e_2 \Leftarrow \tau_2 \dashv (c^{\gamma} \mapsto c(\bigcirc)^{\gamma_2 \mapsto \gamma \sqcup \gamma_2}) \ \$ \ S_2}$$

Where the resulting slice takes sub-slices ρ of $\mathcal{S}_2$ and places them in a context $\rho(\bigcirc)$. That is, if $\mathcal{S}_2$ represented type $\texttt{Int} \to \texttt{Bool}$, the result is a context slice also representing $\texttt{Int} \to \texttt{Bool}$ with the context for the $\texttt{Int}$ part being $\rho(\bigcirc)$ with $\rho \sqsubseteq e_1$ being the sub-slice of the $\texttt{Int}$ part of $\mathcal{S}_2$ etc. The appendix defines this operation (and it's converse).

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

3.1.6 Criterion 3: Contribution Slices

This criterion aims to highlight all regions of code which *contribute* to the given type (either synthesised or analysed). Where *contribute* means that there exists a type that the sub-term could change it's type to to change the type of the overall term (or make it ill-typed). For example in the following term:

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

The terms which *contribute* to the *underlined* lambda term checking successfully against $\texttt{Int} \to \texttt{?}$ is everything *except* the x term, plus the annotation enforcing the type. Highlights related to synthesis are in a darker shade:

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

Notice that, the only sub-terms which *can* have their type changed without affecting the overall type must be expected to be dynamic. Therefore, this criterion just omits all dynamically annotated regions of the program.[12]

These can be calculated by just taking the analysis slice and only the static parts of the synthesis slice.

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

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

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

3.2 Cast Slicing Theory

Cast slicing propagates type slice information during evaluation, by tagging casts types with type slices. The primary reason in using type-indexed slices to calculate type slices was to allow type-slices to be decomposable during evaluation, while retaining only the code relevant to the part of the type involved in the cast. We therefore change the syntax of casts to $\langle \mathcal{S} \Rightarrow \mathcal{S} \rangle$. The updated hazel elaboration and dynamics rules must using slices can be found in the appendix, but are relatively trivial.

3.3 Type Slicing Implementation

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

3.3.1 Hazel Terms

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

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

- **Expressions**: The primary encompassing term including: constructors[13], holes, operators, variables, let bindings, functions & closures, type functions, type aliases, pattern match expressions, casts, explicit fix[14] expression.
- **Types**: Unknown type[15], base types, function types, product types, record types, sum types, type variables, universal types, recursive types.
- **Patterns**: Destructuring constructs for bindings, used in functions, match statements, and let bindings, including: Holes, variables (to bind values to), wildcard, constructors, annotations (enforcing type requirements on bound variables).
- **Closure Environments**: a closure mapping variables to their assigned values in it's syntactic context.

For example fig. 3.1 shows a let binding expression whose binding is a tuple pattern (in blue) binding two variables `x`, `y` annotated with a type (in purple).

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

Figure 3.1: Let binding a tuple with a type annotation.

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

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

[14] Fixed point combinator for recursion.

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

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 3.2: Initial Type Slice Data-Type

However, this did not model the structure of type slices particularly well. Slices are generally incrementally constructed. Synthesis slices build slices from the slices of sub-term, so the list data-type is efficient due to it's persistent nature. However, analysis slices add ids to *all* sub-slices. Hence, analysis slices had *quadratic* space complexity in the depth of the type.

Incremental Slices

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

Instead of tagging an id in an analysis slice to all subslices, we tag it only once as a *global slice*, and *lazily* tag it to sub-slices upon usage. That is, when de-constructing types via function matching etc.

We get the following type:

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

The invariant that a slice has at only one incremental and/or global slice is maintained by splitting into three types (`empty_term`, `incr_term`, `term`). While, regular unsliced types are maintained to provide easier interoperability with the rest of the code-base (and 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 ($x :> y$ means x a subtype of y):

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

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

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

Equally, as functions are bifunctors, contravariant in their input and covariant in their output:

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

This function subtyping property significantly reduces work in defining functions on this type as seen below.

Utility Functions Functions on slices often only concern the underlying type, e.g. checking if a slice is a *list* type. (**ref to sec**). In which case it is more convenient write a $\texttt{typ\_term} \to \alpha$ and $\texttt{typslice\_term} \to \alpha$ function directly on the possible empty slice types.

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

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

Other utility functions include mapping functions, wrapping functions, unpacking functions, matching functions etc.

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, only the right branch is used in the slice. For contribution slices, both must be retained. See how the part of the left branch slice is omitted when already known from the right branch in fig. 3.3 (slices highlighted in green). Note that we still needed some information from the left hand side (the first tuple element).

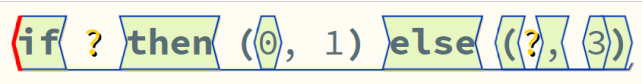


Figure 3.3: Type Slice Joins

3.3.3 Static Type Checking

The Hazel implementation is *bidirectionally typed*. During type checking, the typing *mode* in which to check the term is specified with `Mode.t` type: *synthesising*[16] (`Syn`) or *analysing* (`Ana(Typ.t)`).

The type checker associates each term with a type information object `Info.t`, stored in a map by term id with efficient access. `Info.t` is demonstrated in **??** with arrows representing dependencies (e.g. a terms type depending on the mode, self, and status).

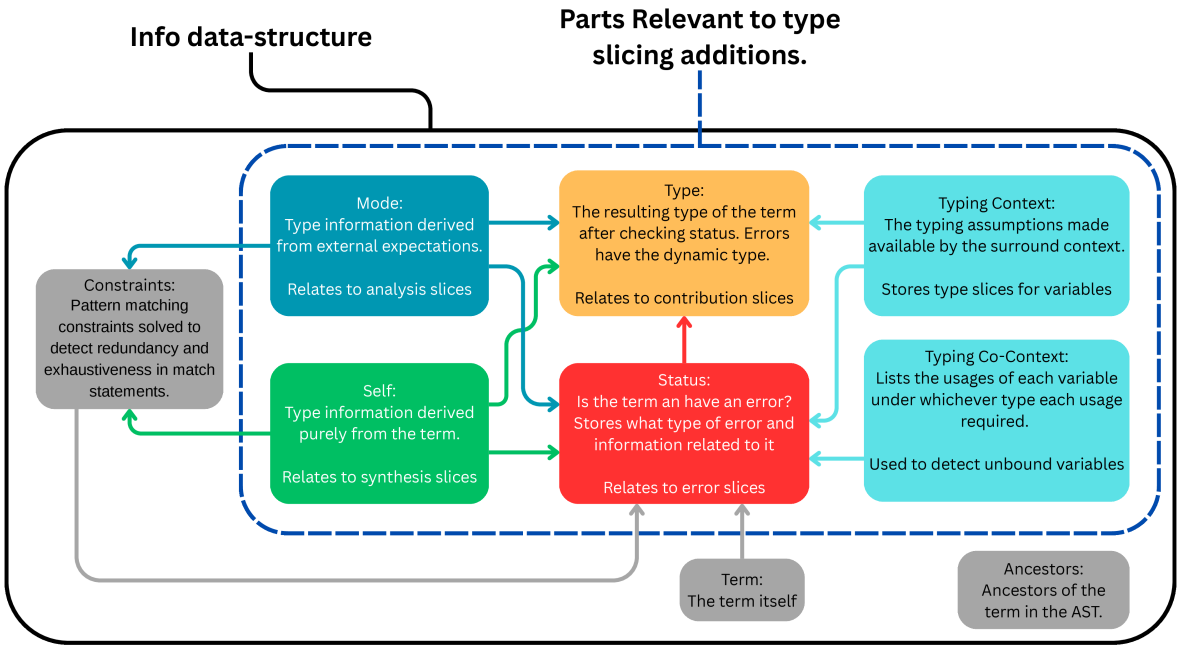


Figure 3.4: `Info.t` data-structure

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

Self and Mode

Code relating to synthesis slices and analysis slices is factored into `Self.t` and `Mode.t`. Additionally, some type checking logic that could be factored but had not yet been on the Hazel dev branch were done so. Despite this, the actual implementation of slicing still requires full knowledge of how the type checker works in order to actually add the *correct* IDs to the slice. Two examples of this slice logic are shown in **??**.

`Self.t` uses incremental slices, while `Mode.t` uses global slices. These global slices are then retained upon type decomposition (e.g. function matching)..

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 a useful addition in practice allowing slices calculated when binding variables to be shown at their use site, see fig. 3.5.

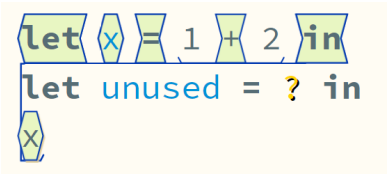


Figure 3.5: Variable Slice

Status and Type

The mode and self are combined to determine the final type. Contribution slices extract the static parts of synthesis slices here. When the synthesised and analysed types are inconsistent, the slice information can be used to construct minimised errors slices (explaining just the inconsistent parts), see section 4.8.1 which demonstrates their retrospective design and effectiveness.

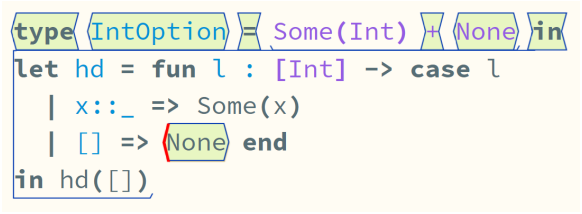
3.3.4 Extensions

To support the full Hazel language, type slices needed to implement many functions, for example: type substitution[17], type normalisation, weak-head normalisation, tracking sum types, various structural matching functions etc. Additionally almost the entire codebase using type needed to be rewritten to use type slices (which cannot so easily be pattern matched), ensuring the slices are maintained correctly throughout.

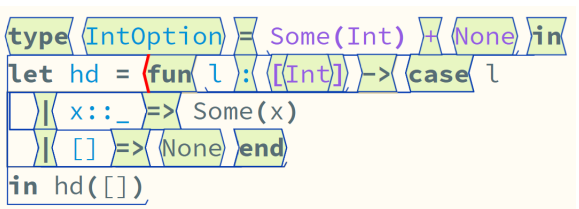
3.3.5 User Interface & Examples

The type slices of the expression at the cursor (in red) are highlighted (in green), see fig. 3.6. Error slices distinguish between the synthesis and analysis parts with blue and red, see the evaluation examples section 4.9.2.

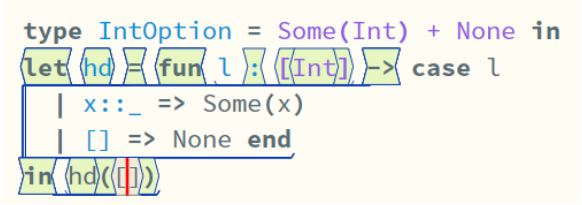
[17]Note: this is the only feature which does not retain type slices fully correctly currently.



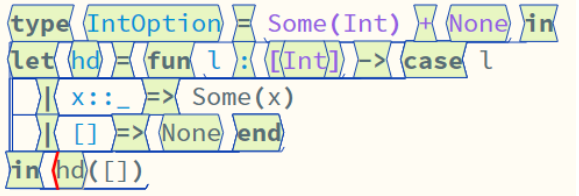
(a) `None` synthesises `IntOption` due to it's type definition.



(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.



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



(d) The variable usage of `hd` synthesises `[Int]`→ `IntOption` similarly (and also due to the binding).

Figure 3.6: Type Slices

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 and from the type as can be determined directly from the term.

For example, list literals recursively elaborate it's terms and join their slices, inserting a cast to this join.

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

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[18] casts are *inserted* (ITGround, ITExpand). The new types are both created via a *ground matching* relation which takes only the topmost compound constructor,

[18] As opposed to being derived from decomposition.

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

Figure 3.7: Ground Matching List

for example ground functions fig. 3.7. Relevant portions of the appendix are fig. B.8, fig. B.10, fig. B.11.

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

3.4.3 Unboxing

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

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

- A list literal: `[1,2,3]`.
- A list with casts wrapped around it: `[1,2,3]⟨[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 Bug

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

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

- *Branch matches?* Execute the branch.
- *Branch does not match?* Try the next branch.
- *Branch indeterminately matches?* Cannot assume the branch doesn't match so must stop evaluation here. The match statement is then indeterminate.

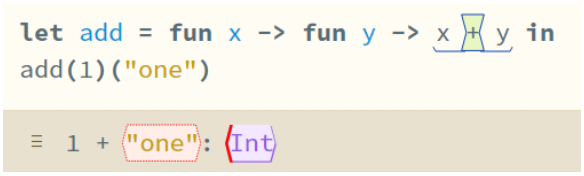
Figure 3.8 demonstrates a concrete example which would get stuck in Hazel, but does *not* need to. I reported and fixed this, with my PR merged into the dev branch.

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

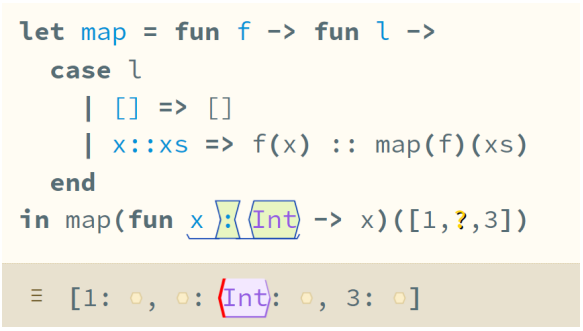
Figure 3.8: Pattern Matching Bug

3.4.4 User Interface & Examples

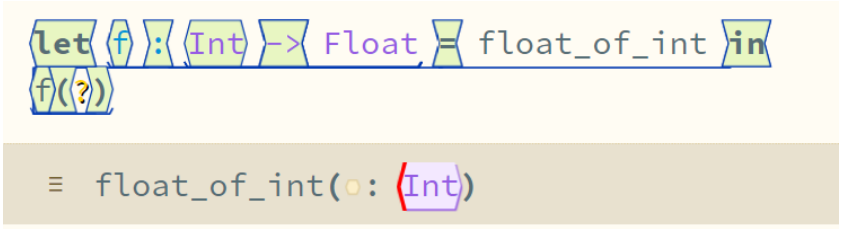
Type slices within casts can be selected from the evaluation result and displayed. This require reworking some of the dependencies of Hazel's model-view-update architecture to make sure the cursor has access to the code editor cell when inside the evaluation result cell.



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



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



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

Figure 3.9: Cast Slicing Examples

3.5 Indeterminate Evaluation

Dynamic errors include evaluation traces, aiding debugging [**TraceVisualisation**]. Static type errors lack such traces, as ill-typed programs do not run. Seidel et al. [29] offer an OCaml algorithm using lazy, non-deterministic narrowing of holes to least specific values based on context (e.g., instantiating a hole in + as an integer).

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

exhaustive generation of these inputs. Further, it is a general evaluation method, not limited to cast error searches. Specifics relating to cast errors, are covered in section 3.6.

This section covers the following, answering each question:

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

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

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

3.5.4 Discusses hole instantiation and substitution. What does lazy instantiation actually entail, when exactly should a hole be instantiated? Which hole[19] should be instantiated in order to continue evaluation to make progress? How should holes be substituted with their narrowed values; the same hole may exist in multiple locations within the expression?

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

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

As always, a UI is implemented in section 3.5.7.

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 3.6.4 discusses the actual implementations of this interface, giving four searching procedures.

Monadic Non-determinism

section 2.1.4 described a high level monadic framework for nondeterminism. I extend this with some extra functions:

- Standard `map` and `join` functions. Mapping transforms all possible solutions, but does not change if a candidate *is* a solution.
- `once : t('a) => option('a)`, extracts any one solution, if one exists. This can be used to efficiently model *don't care* non-determinism, where if one possibility fails, then we know all others fail also.

[19] There may be multiple.

- `run : t('a) => Sequence.t('a)` produces a lazy list (Jane Street's `Base.Sequence`) of all solutions.
- `guard : bool => t(unit)`, represents success. When chaining binds, if any guard binding fails, then the whole computation fails.
- `ifte : m('a) => ('a => m('b)) => m('b)`. This corresponds to Mercury's interpretation of an if-then-else construct [**Mercury**]. It is put to good use for computations which explain the failure of the conditional.

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, monadic non-determinism can be represented by forests [**Bunches**]. A forest is a list of trees, and can be defined as a monad. Choice, similarly to streams, is performed by concatenating forests. Finally, in order to build tree structure, a `wrap` combinator can wrap a forest as a tree whose root leads branches to each tree in the original forest, see fig. 3.10.

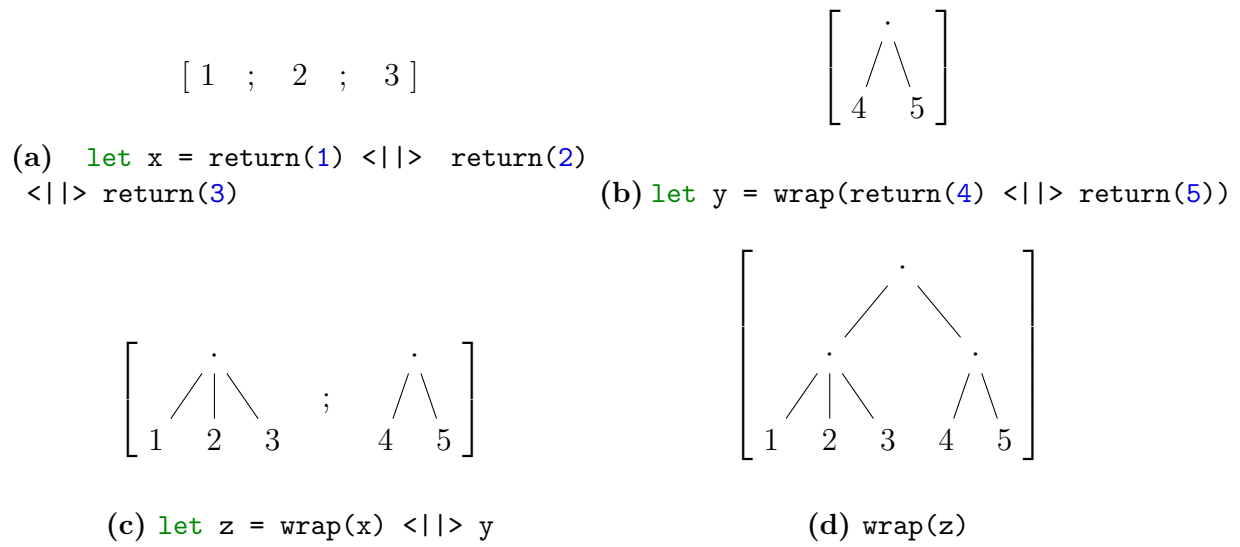


Figure 3.10: Forests Defined Using `wrap`

Therefore, I extend the DSL with a `wrap : t('a) => t('a)` combinator. Here, `wrap` is abstract, the underlying implementation does not actually need to use a forest data structure.[20]

These trees can be traversed in various orders, e.g. breadth-first search (BFS)[**BFSCombinators**], or bounded DFS. With this in mind, `run` now represents performing the search on the tree, returning the solutions in sequence.

In essence, `wrap` allows encoding some notion of *cost*[21] to solutions, which can then affect the search order.

[20] Therefore, DFS can still be efficiently implemented with regular streams.

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

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`,[22] 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 how we can indeterminately evaluation to return all possible values. Demonstrated by fig. 3.11 and with code extract fig. 3.12.

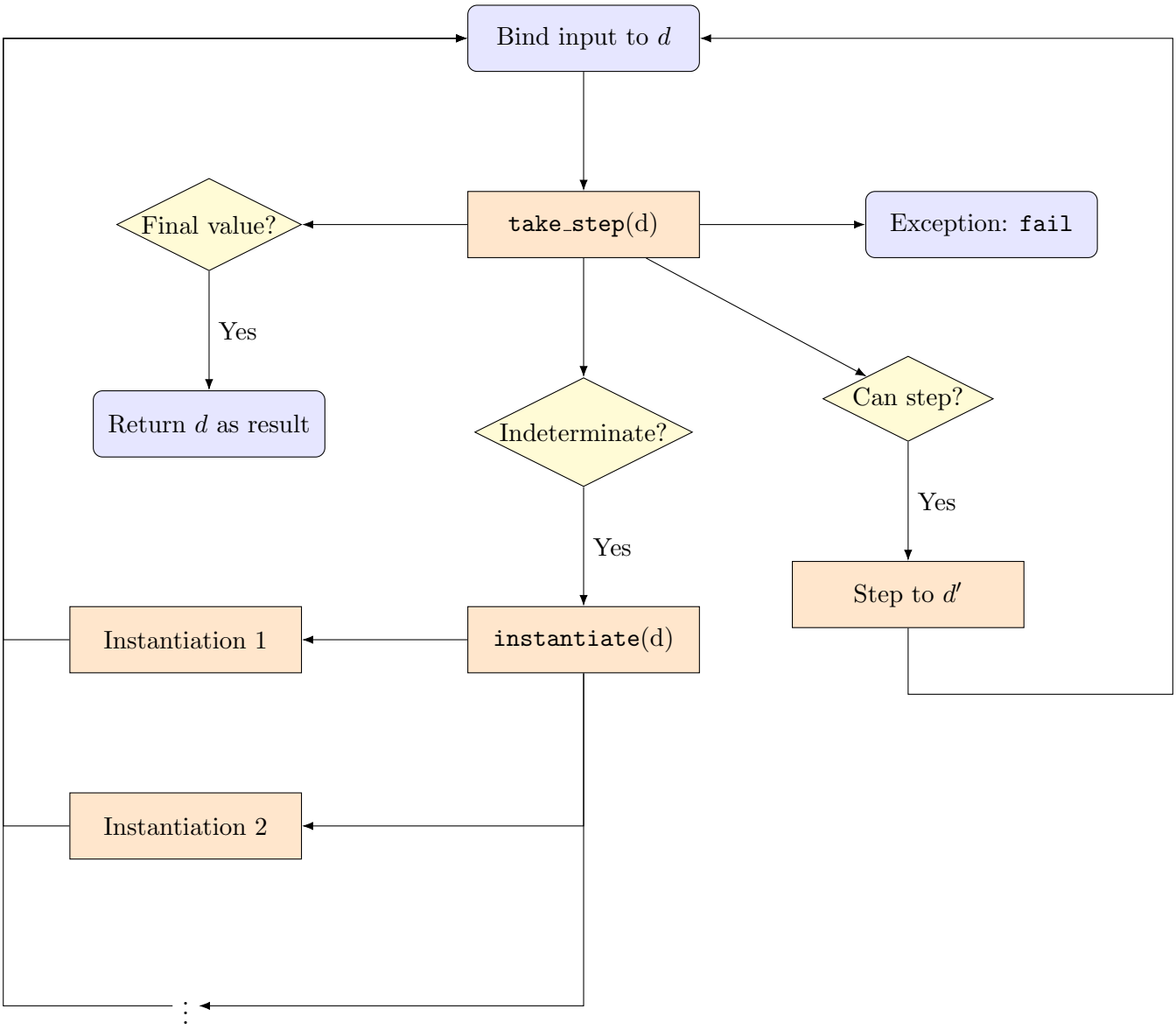


Figure 3.11: Block diagram of indeterminate evaluation to values

[22] Which it usually does, consider streams as an example.

Instantiation is implemented by a *non-deterministic* function `instantiate`, discussed in detail in section 3.5.4:

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

Classifying a term d, into values, indeterminate terms, and expressions with a possible step is done by a (deterministic) function `take_step`, discussed in section 3.5.5:

```
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. To allow for multiple searching methods, the algorithm is placed within a *functor*, which takes a searching method, `S : Search`.

```
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 3.12: Indeterminate Evaluation to Values

3.5.3 Threading Evaluation State

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

- Trace length.
- Number of and size of instantiations performed.

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

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

3.5.4 Hole Instantiation & Substitution

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

Choosing which Hole to Instantiate

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

When attempting to evaluate the indeterminate term some transitions rules require a subterm 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 choice logic will automatically update to any future changes in the transition semantics.

Synthesising Terms for Types

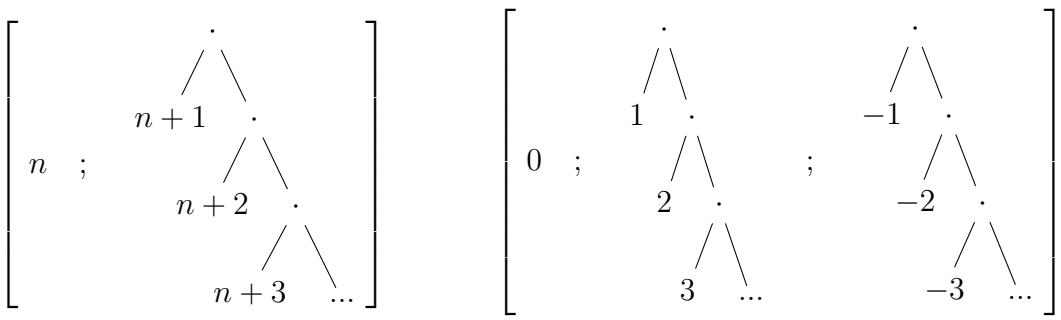
Suppose we know which hole to instantiate and to which type (section 3.5.6). How do we refine these holes fairly and lazily, to the *least specific* value that allows evaluation to continue?

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. 3.13:

```
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 3.13: Enumerating Integers

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

Figure 3.14: Enumerating Strings

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. 3.14:

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

Lists: A list is either the empty list, [ ] or a cons $?_1 :: ?_2$. Note that, 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, if the *input* has an erroneous type but is uncaught as the function is dynamic, potential errors arising from this cannot be found.[23] Extensions to program synthesis (section 5.1.5) would require more sophisticated function instantiation.

Maintaining Correct Casts

Every hole has a dynamic type at runtime. Therefore, that hole's context *expects* it to have the dynamic type. 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. 3.15. Therefore, instantiations must be substituted for every occurrence of the same hole.

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

≡ (○, ○)
```

Figure 3.15: Duplicated Holes

[23] This can occur in dynamic functions with inconsistent branches. But this is extremely rare, and can be fixed for most cases by generating the identity function where function hole's expected type allow.

Consistent hole substitution was described as part of the Hazel calculus section 2.1.2. Unexpectedly, the main Hazel branch did not yet implement it. A full implementation of metavariables and delayed closures is complex. Therefore, as hole closures are not required for hole instantiation,[24] I instead use the existing term ids to represent metavariables, and ensure these ids are maintained and propagated correctly throughout statics, elaboration, and evaluation.[25]

We can substitute terms for every hole with the given ID throughout the entire expressions. Substitutions within closures *include* eagerly evaluating the resulting binding to ensure the invariant that closures bind variables to values.

3.5.5 One Step Evaluator

The step evaluator uses the evaluation abstraction to create an *evaluation context* from a term, to return:

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

This information allows indeterminate evaluation methods to treat different steps differently. For example one might wish to return only the substitution steps, to produce a compressed execution trace.

3.5.6 Determining the Types for Holes

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

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

Directly from Casts

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

Cast Laziness

However, this is *not* always the case. For efficiency reasons, Hazel treats casts over compound data-types 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 discusses the consequence of this choice.

[24] Currently, there is no need to instantiate to a variable reference.

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

Pattern Matching

Does a hole even have only one possible type?

The introduction of (dynamic) pattern matching actually allows terms to be matched against *non-uniform* types if the scrutinee is dynamic. In section 3.5.6, if `term` is an integer 0 then it returns 0, but if it's the string "one" then it returns 1. Hence instantiating `code` to either an int or string might allow progress.

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

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

Extended Match Expression Instantiation (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, sometimes this instantiation is not specific enough to explicitly not match with previous statements, hence getting stuck. Proposed solutions can be found in **??**.

3.5.7 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:

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

?? What exactly are cast errors? How can they be detected? Which ones are actually *relevant*, causing evaluation to get stuck?[26]

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

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

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

Figure 3.16: Instantiating a Scrutinee

3.6.1 Abstract Indeterminate Evaluation

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

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

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

3.6.2 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. The aim of this project was to allow give dynamic traces *explaining* a static type error. Clearly these failure must be ignored.

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

Therefore, we consider only the casts which are *causing* a term to be indeterminate (therefore getting stuck), this is implemented similarly to choosing which hole to instantiate (section 3.5.4).

Cast Laziness Only casts between *ground* types are checked for consistency. Due to cast laziness (section 3.5.6), some are cast between inconsistent types, but *not* placed within a *cast failure*.

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

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

3.6.3 Explaining Cast Errors

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

Show how streams & append represent depth first search

Figure 3.17: Depth First Search as Streams

3.6.4 Searching Methods

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

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

Depth First Search

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

Breadth First Search

Breadth first search needs to take specific account of the tree structure. Sequences of bags[27] can be used to represent solutions at each *level* of the tree [**BFSCombinators**]. Therefore, forcing each element of the stream gives the solutions at successive depths, i.e. iterating through these bags returns the solutions in a breadth first order.

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

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

Merge trees, show that it maintains BFS.

Figure 3.18: Breadth First Choice

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

For a bag `b` of elements `x1`, `x1` ..., `xn`. The tree which produces these solutions in the first level, $[b]$[28], is equivalent to `return(x1) <||> return(x2) <||> ... <||> return(xn)`.

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

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

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

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

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

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

[28] Using list notation also for sequences

Figure 3.19: Monadic BFS Requires Exponential Space

However, as `f` is mapped to the entire bag `b`, it is difficult to maintain laziness in OCaml. We need partial applications for f... **TODO - can't actually work out how to do this in OCaml?!?!**

Iterative Deepening Depth First Search

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

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

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

To represent iterative deepening [**SearchAlgebra**], we can use functions `int => list(('a, int))`, calculating every[29] solution found within an integer depth bound `d`, alongside it's remaining depth budget `r`. Tracking the remaining depth budget simplifies `bind` and allows only solutions on the fringe (zero budget) to be output upon each iteration, so the same solution will not appear twice.

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

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

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

Interleaved Streams

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

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

[29] Again, relying on finite branching factor.

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

Vs.

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

Figure 3.20: Associativity of Iterative DFS `bind`

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

3.6.5 User Interface

3.7 Repository Overview

3.7.1 Branches

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

3.7.2 Hazel Architecture

Chapter 4

Evaluation

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

4.1 Success

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

4.2 Goals

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

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

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

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

[1] That is, reasonable coverage and performance.

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

4.3 Methodology

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

Quantitative Analysis

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

The time and space usage of the search procedure implementations were micro-benchmarked for each ill-typed program in the corpus. Up to 100 runs were taken per program with estimated time, major and minor heap allocations were estimated using an ordinary linear regression (OLS) to $r^2 > 0.95$[3] via the *bechamel* library [**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.[4]

Qualitative Analysis

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

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

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

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

4.4 Hypotheses

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

[3] TODO: get values

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

[5] Which were then implemented and analysed.

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

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

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

The Small Scope Hypothesis: This hypothesis [**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.[7]

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

[7] 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

4.5.2 Statistics

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

4.6 Performance Analysis

4.6.1 Slicing

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

4.6.2 Search Procedure

Only the annotated ill-typed corpus containing inconsistency errors are used in evaluating the search procedure. After all, any well-typed program cannot have a dynamic type error.

As the search procedure may be non-terminating, the results are found under a 30s time limit. Each search implementation gives terminates on a different set of programs with potentially different witnesses and traces. The succeeding *effectiveness* analysis considers the search procedures results under these constraints.

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

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

[8] When using normal, deterministic, evaluation.

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

[10] Having the least overhead.

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

Figure 4.2: Benchmarks: Search Implementations

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

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

4.7 Effectiveness Analysis

4.7.1 Slicing

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

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

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

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

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

4.7.2 Search Procedure

Witness 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

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

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

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

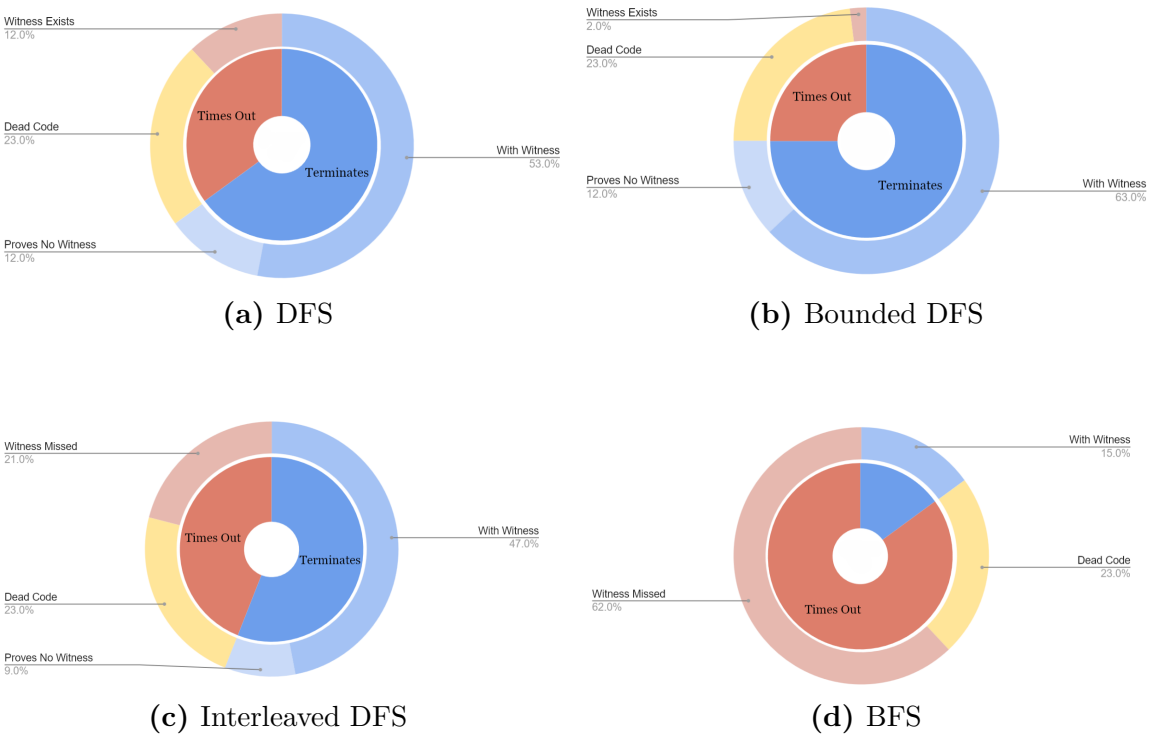
Figure 4.5: Effectiveness: Cast Slices

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

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

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

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



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

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

Figure 4.6: Search Procedure Coverage

Witness & Trace Size

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

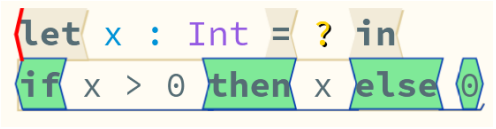
However, there was no correlation (Pearson correlation [**PearsonCorrelation**]) between witness sizes and trace sizes, even when normalised by the original deterministic evaluation trace lengths. This is likely because most errors are in the base cases, so few large witnesses are even found, with the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

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

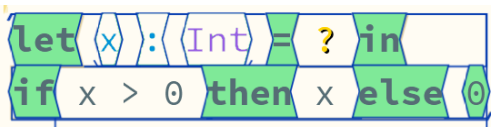
Figure 4.7: Witness & Trace Sizes

4.8 Critical Analysis

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



(a) Ad-hoc Slice: Let expression omitted



(b) Ordinary Slice

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

4.8.1 Slicing

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

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

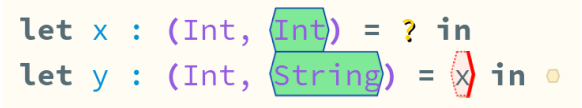
Figure 4.8 binds a integer `x`, but this variable is only used in one branch, where the type information for the `if` is already known from the other branch, so not included in the slice (highlighted in green). So, even though we selected[14] the whole program, the let expression is omitted from the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

Error Slices

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

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

(a) Partially Inconsistent Branches

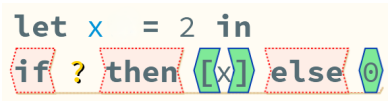


(b) Partially Inconsistent Expectations

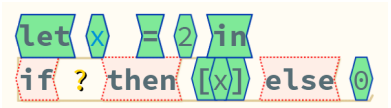
Figure 4.9: Error Slices

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

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



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



(b) Ordinary Error Slice

Figure 4.10: Minimised Error Slices

4.8.2 Structure Editing

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

Parsing and structural edit actions are made efficient[17] by use of a zipper data structure [**HuetZipper**, **OneHoleContext**]. It might be possible to extend this zipper idea into the abstract syntax tree and it's statics (typing information), allowing local changes (and type changes) to propagate in the AST zipper. However, some local changes can propagate type changes *non-locally* (e.g. inserting a new binding) which would require extensive recalculation of the typed AST. These non-local updates would be relatively rare. This is very complex and would require an entire rewrite of the Hazel statics, but would be beneficial for Hazel in general, not just type slicing.

4.8.3 Static-Dynamic Error Correspondence

Section 3.6.3 noted that a static error will elaborates a cast failure, 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, *inconsistent branches* errors, static errors caused by synthesised branches being inconsistent, do not directly cause cast errors. Such errors will only be detected if both branches are used within a static context during the same evaluation run. The search procedure might find such cases if they exist, but cannot so easily associate it back to the static error. However, elaborating these terms does insert casts: on each branch to the dynamic type; these could be associated together with the error and tracked accordingly.

4.8.4 Categorising Programs Lacking Type Error Witnesses

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

[16] The corpus has few *partially* inconsistent errors.
[17] Constant time.

- Witness Exists: BDFS failed to find an existing witness.
- Dead Code: The error was in unreachable code by any type consistent instantiations. These then subclassified into:
  - Pattern Cast Failure: An error was within a pattern matching branch. This additionally makes the branch unreachable. These cast failures would be found by an extended pattern directed instantiation algorithm (section 3.5.6).[18]
  - Unbound Constructor: A pattern matching branch matching an unbound constructor. These unbound errors are also detectable with extended pattern directed instantiation.
  - Wildcards: Error within trivially dead code bound to the inaccessible wildcard pattern: `let _ = ... in ....`
  - Non-Trivial: Dead code which is less easily detectable. There was only one example which was due to infinite recursion for all inputs.
- Hazel Bugs: Evaluation errors encountered stemming from *unboxing bugs* in the main branch of Hazel. These were excluded from the statistics.

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

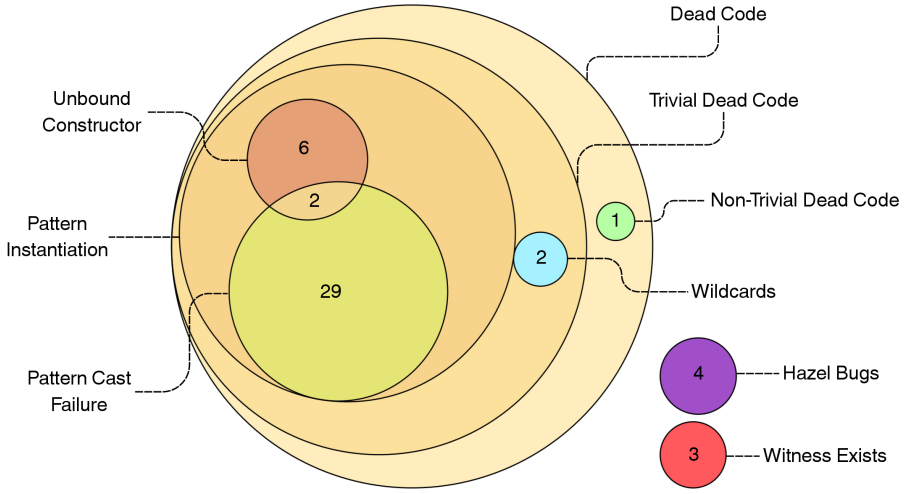


Figure 4.11: Distribution of Failed Program Classes

Non-Termination, Unfairness, and Search Order

DFS is fast, but has significant insurmountable issues with non-termination, resulting in it having worse coverage than BDFS (fig. 4.6). Any time evaluation goes into an infinite loop, no more instantiations can be explored. Similarly, DFS will never backtrack to try previous instantiations, for example instantiating a pair of integers `(?,?)` would only try `(0,?)`, `(0, 0)`,

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

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

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

(a) Witness Exists: prog2270.typed.hazel

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

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

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

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

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

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

Figure 4.12: (Paraphrased) Failure Examples

`(0, 1)`, `(0, -1)`, `(0, 2)` etc. but never `(1, 0)` for example (fig. 4.13). This was a common issue, occurring in 10% of the search corpus. Bounded DFS, BFS, and interleaved DFS, were implemented in response to this. With BDFS performing best due to preference of evaluation over instantiation (and it's lesser memory footprint: fig. 4.2).

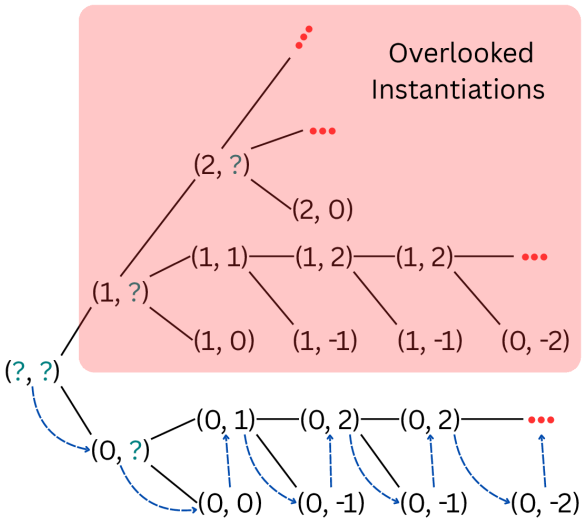


Figure 4.13: Non-exhaustive Instantiations in DFS

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

Dead Code & Nested Errors

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

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

Dynamically Safe Code

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

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

[19] Unbound constructor or inconsistent expectations.

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.

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

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

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

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

Figure 4.14: Inconsistent Lazy Casts: No Cast Failure

Combinatorial Explosion

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

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

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

4.8.5 Improving Code Coverage

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

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

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

Figure 4.15: Witness Requiring Very Specific Instantiations

These errors often require rather specific, interdependent, inputs to be missed by the current search procedure. Therefore, it is likely that programmers are *also* more likely to not notice errors (in dynamic code), or not understand the errors (in static code).[21]

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

One form of this, purely structural pattern-directed instantiation, discussed in section 3.5.6, would have discovered 2 of the 3 missed witnesses by BDFS. In order to extend this structural pattern-directed instantiation to worth also with base types (integers), requires more efficient ways to representing constraints. To extend this to include floats, inequalities, list lengths, etc. requires full-blown symbolic execution [**SymbolicExecutionSurvey**].

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

4.9 Holistic Evaluation

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

4.9.1 Interaction with Existing Hazel Type Error Localisation

Hazel has three types of errors which can be addressed to varying extent by this project:[22] *inconsistent expectations* (where a term's analytic and synthetic types are inconsistent), *inconsistent branches* (where branches are synthetic and inconsistent, this also applies to values in a list literal), and *inexhaustive match* warnings.

Section 4.8.3 showed how dynamic witnesses can be associated to both classes of inconsistency errors, with inconsistent branch errors being more difficult to detect in general. Generating examples for inexhaustive match statements was not a core aim of this project, but indeterminate evaluation can do this automatically (see the `inexhaustive-matches` branch **TODO**). Faster, static generation of counter-examples to match statements is already a (mostly)[23] solved problem in static languages [**PatternMatchingWarnings**], and is under development for Hazel.[24]

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

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

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

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

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.

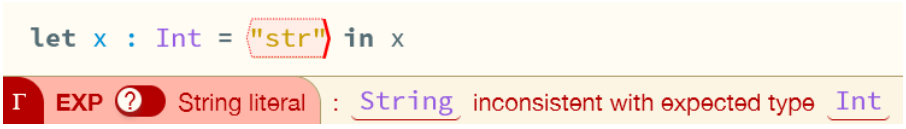


Figure 4.16: Selected Static Error described by Hazel Context Inspector

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

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

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

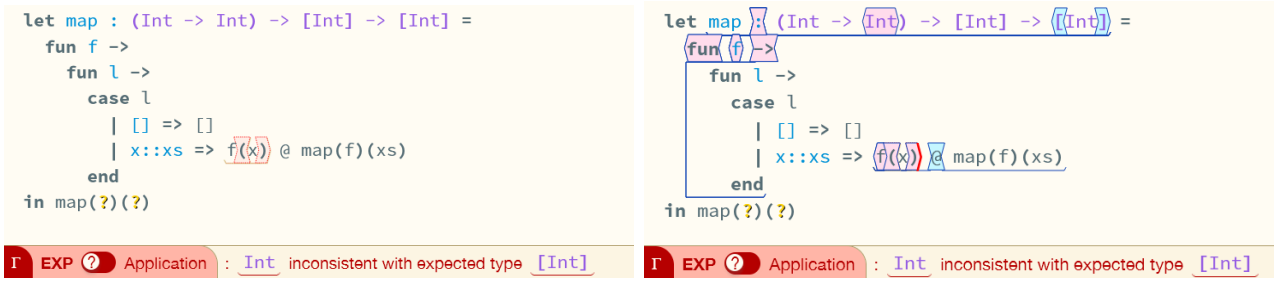
Figure 4.17: Partially Annotated Program misses Error

4.9.2 Examples

Returning to the map example, when fully annotated a static error can located at `f(x)` which has type `Int` but expects `[Int]`. The error slice shows (in pink) why `Int` is synthesised (due to

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

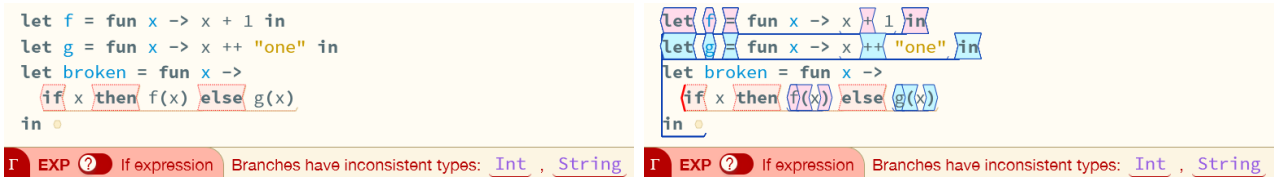
input `f` being annotated `Int -> Int` and applied) and (in blue) why a list[26] is expected (being an argument of list concatenation). See fig. 4.18.



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

Figure 4.18: Example: Type Slice of Inconsistent Expectations

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



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

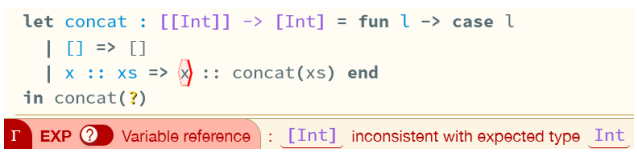
Figure 4.19: Example: Type Slice of Inconsistent Branches

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

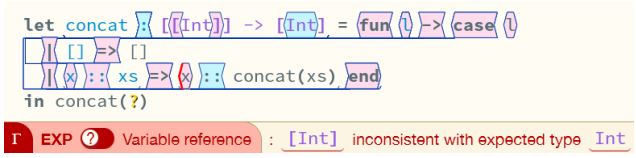
[26] A full analysis slice would also include the full `[Int]` annotation. But the inconsistency arises from the list part.

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

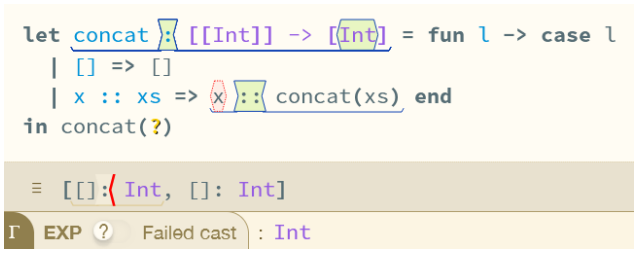
[28] Looking at larger witnesses can be useful to spot pattern with what is happening overall, i.e. that the lists are being consed, not concatenated.



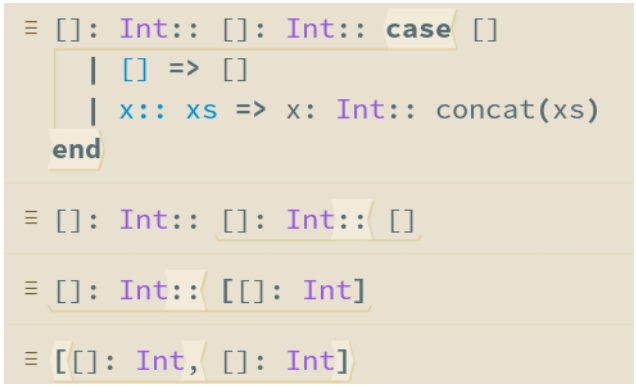
(a) Error: Mistaken Cons for Concat



(b) Verbose Type Slice



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



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

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

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

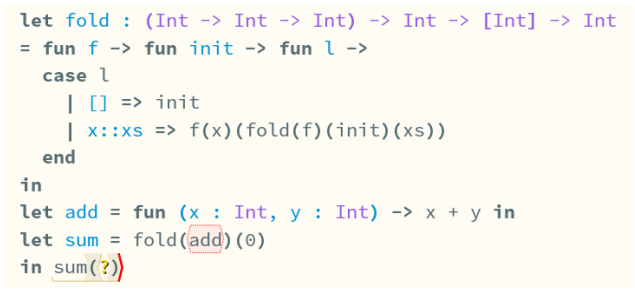
Finally, when code is dynamic, errors might not be statically found, and type slices have less information to work with. Figure 4.22 considers finding a simple error in a dynamic function. Cast slicing still works even without type annotations, allowing errors to blame regions of the source code.

4.9.3 Usability Improvements

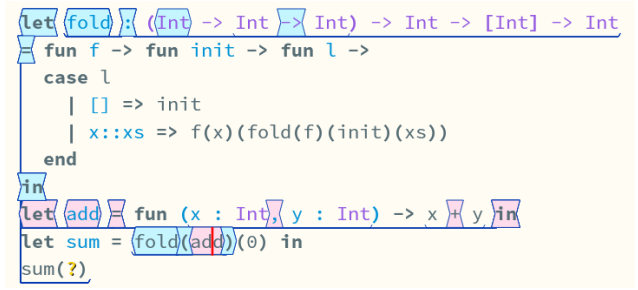
As designing an intuitive user interface was not a core goal, therefore actual use of the features can be quite awkward or unintuitive. Below, I propose various improvements, for which the architecture, of both Hazel and the newly implemented features, is sufficiently abstract and flexible to easily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

- Displaying type slices only *upon request* using the Hazel context inspector, which displays the *analysing* and *synthesising* types of the selected expression, see fig. 4.16.
- Allowing the user to deconstruct type slices to query, for example: if a term has a function type, they could select the *return type* and get only the part of the slice relevant to that, or select the function arrow to get the part that makes it a *function* (ignoring the argument

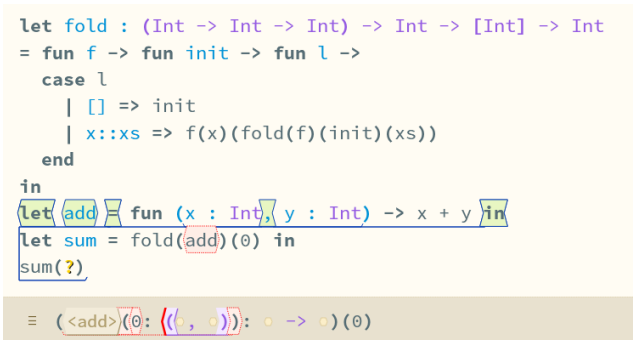
[29] Some of which were detailed in the Implementation chapter.



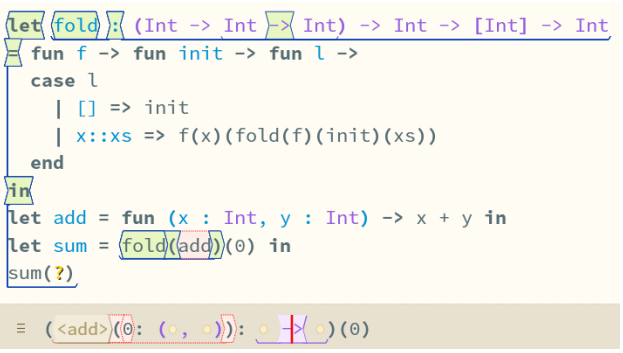
(a) Confused uncurried for curried `add`



(b) Somewhat Verbose Type Slice

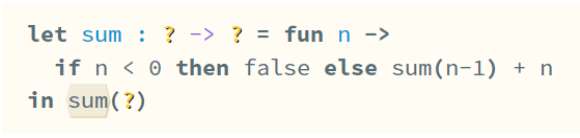


(c) Witness (`[0]`) expects input to `add` to be a tuple

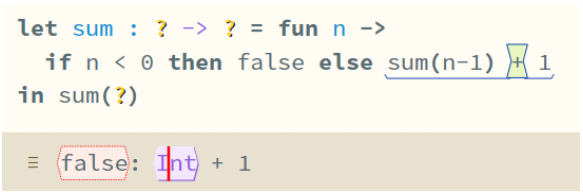


(d) Witness (`[0]`) expects output of `add(0)` to be a function

Figure 4.21: Example: Subtle Currying Error



(a) Error



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

Figure 4.22: Example: Searching for Error in Dynamic Code

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

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

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

Chapter 5

Conclusions

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

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

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

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

5.1 Further Directions

This section presents some *extensions* to improve the features as justified by the evaluation (section 4.8 & 4.9). Also, further interesting applications of the features are considered.

5.1.1 UI Improvements, User Studies

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

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

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

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

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

5.1.2 Cast Slicing

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

5.1.3 Proofs

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

5.1.4 Property Testing

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

The current algorithm generates smaller inputs first,[4] similarly to SmallCheck [**SmallCheck**]; the approach being justified by the *small scope hypothesis* (section 4.4).

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

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

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

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

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

Logic programming has historically been used for *general-purpose program synthesis*. An incomplete Hazel program could be used as a specification for it's incomplete parts, where executing the program would synthesise these. However, this approach has been found to

[3] The largest time-investment of the entire project.

[4] By default. This is also customisable, as in SmallCheck.

[5] Without refactoring the tested code.

be inefficient due to the need to solve *every* possible program, and prone to underspecified problems, among other drawbacks [**LogicProgramSynthesisDrawbacks**]. Most naturally written Hazel programs would be underspecified.

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

5.1.6 Symbolic Execution

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

The evaluation of holes can already be consider a form of simple symbolic execution. However, for indeterminate evaluation, the only constraints considered are the type of the hole. Section 3.5.6 considered constraining instantiation based on structures of pattern within match statements. Full symbolic execution would also consider more complex constraints on the *values* of base types.

Polymorphism

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

5.1.7 Let Polymorphism & Global Inference

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

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

The intersection of gradually typing and principal type schemes has been explored by Garcia and Cimini [**GradualTI**] 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 would be relatively easily extended to a let-polymorphic Hazel in the style of Miyazaki et al. However, type slicing would now have to consider non-local

constraints involved and the code which sourced them, differing significantly from the proposed theory for bidirectional typing. Somewhat similar ideas have been explored in *constraint-based type error slicing* by Haack and Wells [**HaackErrSlice**].

5.2 Reflections

TODO

Appendix A

Overview of Semantics and Type Systems

Syntax

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

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

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

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

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

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

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

Judgements & Inference Rules

A *judgement*, J, is an assertion about *expressions* in a language [**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 $x_1 : \tau_1, \ldots, x_n : \tau_2$. For example the SLTC[2] [**TAPL**] has a typing rule for lambda expression and application as follows:

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

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

Small Step Operational Semantics

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

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

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

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

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

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

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

[2] Simply typed lambda calculus.

Appendix B

Hazel Formal Semantics

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

B.1 Syntax

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

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

B.2 Static Type System

B.2.1 External Language

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

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

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

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

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

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

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

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

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

Figure B.3: Type consistency

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure B.5: Elaboration judgements

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

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

Figure B.7: Identity substitution and substitution typing

B.2.3 Internal Language

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

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

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

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

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

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

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

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

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'

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

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

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

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

Figure B.10: Instruction transitions

B.3.3 Contextual Dynamics

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

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

Figure B.11: Ground type matching

Context syntax:

$$E ::= \circ \mid E(d) \mid d(E) \mid (\!|E|\!)^u_\sigma \mid E\langle \tau \Rightarrow \tau \rangle \mid E\langle \tau \Rightarrow ? \not\Rightarrow \tau \rangle$$

$\boxed{d = E[d]}$ d is the context E filled with d' in place of $\circ$

$$\text{ECOuter}\ \frac{}{d = \circ[d]} \qquad \text{ECApp1}\ \frac{d_1 = E[d_1']}{d_1(d_2) = E(d_2)[d_1]} \qquad \text{ECApp2}\ \frac{d_2 = E[d_2]}{d_1(d_2) = d_1(E)[d_2']}$$

$$\text{ECNEHole}\ \frac{d = E[d']}{(\!|d|\!)^u_\sigma = (\!|E|\!)^u_\sigma[d']} \qquad \text{ECCast}\ \frac{d = E[d']}{d\langle \tau_1 \Rightarrow \tau_2 \rangle = E\langle \tau_1 \Rightarrow \tau_2 \rangle[d']}$$

$$\text{ECCastError}\ \frac{d = E[d']}{d\langle \tau_1 \Rightarrow ? \not\Rightarrow \tau_2 \rangle = E\langle \tau_1 \Rightarrow ? \not\Rightarrow \tau_2 \rangle[d']}$$

$\boxed{d \mapsto d'}$ d steps to d'

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

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.

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

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

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

Figure B.13: Hole substitution

Appendix C

On Representing Non-Determinism

High Level Representation

Other high level representations, besides monads, include:

Logic Programming DSL: Languages like Prolog [**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]

[1] An important dependency of Hazel.

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

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

Appendix D

Monads

A monad is a parametric type $m(\alpha)$ equipped with two operations, `return` and `bind`, where `bind` is associative and `return` acts as an identity with respect to `bind`:

A monad m, is a parameterised type with operations:

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

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

Satisfying the monad laws:

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

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

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

Figure D.1: Monad Definition

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

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

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

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

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

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

Appendix E

Slicing Theory

E.1 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 19 (Term Slice Syntax)**.** *Pattern expression slices p:*

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

Type slices υ:

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

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 20 (Term Precision)**.** *Pattern slices p:*

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

Type slices υ:

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

Expression slices ς:

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

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

Proposition 4 (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 □

Lattice Structure

Proposition 5 (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 21 (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 (Γ) 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 22 (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 23 (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:*

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

Proposition 6 (Precision is a Partial Order)**.** *Typing assumption precision forms a partial order on typing assumption slices.*

Proof. **Typeseting TODO** □

Lattice Structure

Definition 24 (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 7 (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 □

E.1.3 Expression Typing Slices

Definition 25 (Expression Typing Slices)**.** *An expression typing slice ρ is a pair ς^{γ} of expression slice ς and typing assumption slice γ.*

Precision

Definition 26 (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 8 (Precision is a Partial Order)**.** *Expression typing precision forms a partial order on expression typing slices.*

Proof. **Typesetting TODO** □

Lattice Structure

Definition 27 (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 9 (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 28 (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:

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

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

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

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

Definition 30 (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 31 (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]

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

The input to contexts is

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

Definition 32 (Context Substitution)**.**

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

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

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

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

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

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

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

Definition 33 (Context Composition)**.** *Defined analogously as context substitution, but substituting contexts, provided the input and output classes match. If $\mathcal{C}_1 : X \to Y$ and $\mathcal{C}_2 : Y \to Z$ then $\mathcal{C}_2\{C_1\} = \mathcal{C}_2 \circ \mathcal{C}_1 : X \to Z$. Equivalence of contexts can be defined syntactically and coincides exactly with an extensional definition.*

Proposition 10 (Context composition is associative)**.** *For all $\mathcal{C}_1 : X \to Y$, $\mathcal{C}_2 : Y \to Z$, and $\mathcal{C}_3 : Z \to W$ 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 34 (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 $c'\{t\} \sqsubseteq c\{t\}$.*

Proposition 11 (Precision is a Partial Order)**.** *Context precision forms a partial order on context slices.*

Proof. **Typesetting TODO** □

Conjecture 7 (Context Filling Preserves Precision)**.** *For context slice $c : X \to Y$ and term slice ς of class X. Then if we have slices $\varsigma' \sqsubseteq \varsigma$, $c' \sqsubseteq c$ then also $c'\{\varsigma'\} \sqsubseteq c\{\varsigma\}$.*

Lattice Structure

Definition 35 (Context Slice Joins & Meets)**.** *For context slices $c_1 : X \to Y$ and $c_2 : X \to Y$ and any term t of class X:*

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

Definition 36 (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 12 (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 37 (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 38 (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 13 (Precision is a Partial Order)**.** *Typing assumption context precision forms a partial order on typing assumption context slices.*

Proof. **Typesetting TODO** □

Conjecture 8 (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 39 (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 9 (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 40 (Expression Context Typing Slice)**.** *An expression context typing slice $\mathfrak{p}$ is a pair c^f of a context slice c and typing assumption context slice f.*

Precision

Definition 41 (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 14 (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 42 (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 15 (Expression Context Slice Composition is Associative)**.** *For all $\mathfrak{p}_1$, $\mathfrak{p}_2$, and $\mathfrak{p}_3$ then:*

$$(\mathfrak{p}_3 \circ \mathfrak{p}_2) \circ p_1 = \mathfrak{p}_3 \circ (\mathfrak{p}_2 \circ p_1)$$

Proof. **Typesetting TODO** □

Lattice Structure

Definition 43 (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 16 (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 44 (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:

$$\overline{p} = p$$
$$\overline{p_1 * \mathcal{S}_1 \to p_2 * \mathcal{S}_2} = p_1 \circ \overline{\mathcal{S}_1} \sqcup p_2 \circ \overline{S_2}$$

Definition 45 (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 17 (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 46 (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:

$$\overline{\rho} = \rho$$
$$\overline{p_1 * \mathcal{S}_1 \to p_2 * \mathcal{S}_2} = p_1\{\overline{\mathcal{S}_1}\} \sqcup p_2\{\overline{S_2}\}$$

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 47 (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 18 (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** □

Definition 48 (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 19 (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** □

E.4 Checking Contexts

Definition 49 (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 50 (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 20 (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:*

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

$$\frac{\Gamma \vdash \mathcal{C}^{\mathcal{F}} \text{ checks } \lambda x.\ e \text{ against } \tau \quad \tau \blacktriangleright_{\rightarrow} \tau_1 \rightarrow \tau_2}{\Gamma, x : \tau_1 \vdash \mathcal{C}^{\mathcal{F}} \circ (\lambda x.\ \bigcirc)^{\Gamma \mapsto \Gamma \backslash x:\tau_1} \text{ checks } e \text{ against } \tau_2}$$

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 21 (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 51 (Synthesis Slices)**.** *For a synthesising expression, synthesise*τ*. 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 22 (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 $\mathcal{S}$

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

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

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

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

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

Figure E.1: *Minimum synthesis slice* calculation

Conjecture 10 (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 52 (Analysis Slice)**.** *For a term* e *analysing against* τ*:* $\Gamma \vdash e \Leftarrow \tau$*, and a minimally scoped checking context* $\mathcal{C}^{\mathcal{F}}$ *for* e*. An analysis slice is a slice* $c^{\mathcal{J}}$ *of* $\mathcal{C}^{\mathcal{F}}$ *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 11 (Minimum Analysis Slices)**.** *The minimum analysis slice analysing* e *in a checking context* $\mathcal{C}^{\mathcal{F}}$ *is an analysis slice* $\mathcal{p}$ *such that for any other any other analysis slice* $\mathcal{p}'$ *is at least as specific,* $\mathcal{p} \sqsubseteq \mathcal{p}'$*. This minimum always uniquely exists.*

TODO

Figure E.2: *Minimum analysis slice* calculation

Conjecture 12 (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 53 (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:

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

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

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

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

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

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).

Appendix F

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 F.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.

Appendix G

Merges

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

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

Make sure the use of terms is consistent throughout dissertation.

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.