

Number 1006



UNIVERSITY OF
CAMBRIDGE

Computer Laboratory

Type error debugging in Hazel

Max Carroll

July 2026

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500
<https://www.cl.cam.ac.uk/>

© 2026 Max Carroll

This technical report is based on a dissertation submitted May 2025 by the author for the degree of Bachelor of Arts (Computer Science Tripos) to the University of Cambridge, Sidney Sussex College.

Some figures in this document are best viewed in colour. If you received a black-and-white copy, please consult the online version if necessary.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

<https://www.cl.cam.ac.uk/techreports/>

ISSN 1476-2986

DOI <https://doi.org/10.48456/tr-1006>

Abstract

This report seeks to improve the debugging of type errors in Hazel, a gradually typed functional language supporting incomplete programs, by explaining static type errors more completely and by combining the benefits of static and dynamic type errors. Three methods are developed. *Type slicing* highlights the code that determines an expression’s synthesised or analysed type, explaining not just where a static error occurred but why; it generalises beyond errors, applying to all expressions. *Cast slicing* propagates type slices through evaluation, attaching to each runtime cast a slice of the source code that gave rise to it, recovering the source context of dynamic type errors. A *type error witness search procedure*, built on indeterminate evaluation, automatically finds inputs witnessing a dynamic counterpart to a static error, linking the two classes of errors. All three methods are given mathematical foundations over a core bidirectionally and gradually typed calculus of functions, holes, and casts; the implementation in the Hazel programming environment extends considerably beyond this formalised fragment, covering nearly the full language. The methods are evaluated on a corpus of ill-typed Hazel programs, with the best search strategy failing to find existing dynamic witnesses for only 2% of the corpus.

Acknowledgements

I thank my supervisors, Anil Madhavapeddy and Patrick Ferris, for their guidance throughout this work. I am especially grateful to Patrick for his day-to-day supervision, and for writing the best-effort OCaml-to-Hazel transpiler `hazel_of_ocaml` [11], from which the corpus of Hazel programs [10] used in the evaluation was constructed.

Contents

1	Introduction	8
1.1	Related Work	9
2	Preparation	10
2.1	Background Knowledge	10
2.1.1	Type Systems	10
2.1.2	The Hazel Calculus	12
2.1.3	The Hazel Implementation	14
2.1.4	Bounded Non-Determinism	15
3	Implementation	17
3.1	Type Slicing Theory	17
3.1.1	Expression Typing Slices	17
3.1.2	Context Typing Slices	18
3.1.3	Type-Indexed Slices	19
3.1.4	Criterion 1: Synthesis Slices	20
3.1.5	Criterion 2: Analysis Slices	20
3.1.6	Criterion 3: Contribution Slices	21
3.2	Cast Slicing Theory	22
3.3	Type Slicing Implementation	22
3.3.1	Hazel Terms	22
3.3.2	Type Slice Data-Type	22
3.3.3	Static Type Checking	25
3.3.4	Integration	26
3.3.5	User Interface & Examples	27
3.4	Cast Slicing Implementation	27
3.4.1	Elaboration	27
3.4.2	Cast Transitions	27
3.4.3	Unboxing	28
3.4.4	User Interface & Examples	28
3.5	Indeterminate Evaluation	28
3.5.1	Resolving Non-determinism	29
3.5.2	A Non-Deterministic Evaluation Algorithm	30
3.5.3	Hole Instantiation & Substitution	30
3.5.4	Determining the Types for Hole Substitutions	32
3.6	Search Procedure	33
3.6.1	Detecting Relevant Cast Errors	33
3.6.2	Searching Methods	34
3.7	Implementation Availability	35
4	Evaluation	36
4.1	Success	36
4.2	Evaluation Criteria	36
4.3	Methodology	36

CONTENTS	6
4.4 Hypotheses	37
4.5 Program Corpus Collection	37
4.5.1 Methodology	37
4.5.2 Statistics	38
4.6 Performance Analysis	38
4.6.1 Slicing	38
4.6.2 Search Procedure	38
4.7 Effectiveness Analysis	38
4.7.1 Slicing	38
4.7.2 Search Procedure	38
4.8 Critical Analysis	40
4.8.1 Slicing	40
4.8.2 Static-Dynamic Error Correspondence	40
4.8.3 Categorising Programs Lacking Type Error Witnesses	41
4.8.4 Improving Code Coverage	43
4.9 Holistic Evaluation	43
4.9.1 Interaction with Existing Hazel Type Error Localisation	43
4.9.2 Examples	44
5 Conclusions	47
5.1 Further Directions	47
5.1.1 UI Improvements, User Studies	47
5.1.2 Cast Slicing	47
5.1.3 Property Testing	47
5.1.4 Non-determinism, Connections to Logic Programming	47
5.1.5 Symbolic Execution	47
5.1.6 Let Polymorphism & Global Inference	47
Bibliography	49
A Overview of Semantics and Type Systems	55
B Hazel Formal Semantics	57
B.1 Syntax	57
B.2 Static Type System	57
B.2.1 External Language	57
B.2.2 Elaboration	58
B.2.3 Internal Language	59
B.3 Dynamics	60
B.3.1 Final Forms	60
B.3.2 Instructions	60
B.3.3 Contextual Dynamics	61
B.3.4 Hole Substitution	62
C Slicing Theory	63
C.1 Expression Typing Slices	63
C.1.1 Term Slices	63
C.1.2 Typing Assumption Slices	65
C.1.3 Expression Typing Slices	66
C.2 Context Typing Slices	67
C.2.1 Contexts	67

CONTENTS	7
C.2.2 Context Slices	68
C.2.3 Typing Assumption Contexts & Context Slices	69
C.2.4 Context Typing Slices	70
C.3 Type-Indexed Slices	71
C.3.1 Type-Indexed Context Typing Slices	71
C.3.2 Type-Indexed Expression Typing Slices	72
C.3.3 Function Matching	72
C.3.4 Global Application	72
C.4 Checking Contexts	73
C.5 Criterion 1: Synthesis Slices	74
C.6 Criterion 2: Analysis Slices	75
C.7 Criterion 3: Contribution Slices	75
C.8 Cast Syntax	77
C.9 Elaboration	77
C.10 Dynamics	78
D Extended Pattern Matching Instantiation	80
E Supplementary Results and Corpus Data	81
F On Representing Non-Determinism	83

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

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 [27]. Additionally, the errors only *state* the expected types, but with no explanation for *why*.

In dynamic typing, type errors are found later only appearing during runtime with specific inputs. Additionally, they don't generally specify any source code context which caused them. However, such an error is accompanied by an evaluation trace, which can be *more intuitive* [42], demonstrating concretely why values are ill-typed programs go wrong.

Aims: This report seeks to improve user understanding of type errors by explaining static type errors more *completely*, and *combining* the benefits of static and dynamic type errors. I consider three directions to achieve this, implementing three features for use in the Hazel language [1]:

1. Can we explain *static type errors* more *completely*, highlighting the code that determines *why* the error expects it's inconsistent type?

Being more *complete*, this would alleviate the issue of static errors being incorrectly localised, while also helping build understanding of *why* the errors occur.

Solution: I devise a *novel* method: **type slicing**. Including formal mathematical foundations built upon the formal *Hazel calculus* [23]. Additionally, it generalises to highlight all code relevant to typing any expressions (not just errors).

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

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

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

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

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

Solution: I implement a **type error witness search procedure**, which discovers inputs (witnesses) to expressions which cause a *dynamic type error*. This is based on research by Seidel et al. [31] who devised and evidenced the usefulness of a similar procedure for a subset of OCaml.

Hazel [1] is a functional, locally inferred, and gradually typed research language that allows writing *incomplete programs* under active development at the University of Michigan. Being gradually typed, it is a natural setting for this work, allowing both static and dynamic code

```

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

```

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

Note: ? represents a hole, an *incomplete* part of the program.

Figure 1.1: Example: Concatenation @ has been used instead of cons ::, but $f(x)$ is incorrectly localised as the error. Type slicing will give a full context highlighting the terms involved in the error (the annotation, $f(x)$ and @ operator). The search procedure could find a witness (e.g. [0]) concretely evaluating to a dynamic error. The cast in the dynamic error will contain a slice highlighting the source code that makes @ take lists as input. The results are demonstrated in the evaluation section 4.9.2

to coexist. The evaluation 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 and may be linked automatically.

1.1. Related Work

There has been extensive research into the field of programming languages and debugging, attempting to understand *what* is needed [36], *how* developers fix bugs [20], and a plethora of compiler improvements and tools. This report adds to this body of research by focusing on the Hazel language, which is itself an active research system being taken in various directions but is of particular note as a *teaching language* [19] for students; these features can additionally help with building understanding of bidirectional type systems.

To my knowledge the ideas of *type slicing* and *cast slicing* are novel. However, they were inspired, but differing substantially, to *program slices*, originally explored by Weiser [75], slices in expression-based languages [41], *dynamic program slicing* [71] and *type error slicing* [55, 52], which similarly relate to type systems.

The *type witness search procedure* is based upon Seidel et al. [31], but with significant differences, which will be explained throughout.

Preparation

This chapter gives the technical background used by the report: the type theory for understanding Hazel’s core semantics, the relevant parts of the Hazel implementation, and the bounded non-determinism used by the witness search procedure.

2.1. Background Knowledge

2.1.1. 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 (chapter 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 [48, 12]. Of particular relevance here, execution traces are known to help provide insight to errors [42], 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* [68]. 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*¹ cast expressions which maintain and enforce runtime type constraints alongside a dynamic type written $?$.

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

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

Then *injections* meeting *projections*, $v\langle\tau_1 \Rightarrow ? \Rightarrow \tau_2\rangle$ represent 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:

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

Compound type casts will be decomposed during evaluation. For example, applying v to a function wrapped in a cast decomposes the cast into casting the applied argument and then the result:

$$(f\langle\tau_1 \rightarrow \tau_2 \Rightarrow \tau'_1 \rightarrow \tau'_2\rangle)(v) \mapsto (f(v\langle\tau'_1 \Rightarrow \tau_1\rangle))\langle\tau_2 \Rightarrow \tau'_2\rangle$$

Or if f has the dynamic type:

$$(f\langle? \Rightarrow \tau'_1 \rightarrow \tau'_2\rangle)(v) \mapsto (f(v\langle\tau'_1 \Rightarrow ?\rangle))\langle? \Rightarrow \tau'_2\rangle$$

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 cast.

¹Directly represented in the language syntax as expressions.

Gradual Type Systems

A *gradual type system* [33, 49] combines static and dynamic typing. Terms may be annotated as dynamic, marking regions of code omitted from type-checking but still *interoperable* with static code. For example, the following type checks:

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

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

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

- The *external language* – where static type checking is performed which allows annotating expressions with the dynamic type.
- The *internal language* – where evaluation and runtime type checking is performed via cast expressions.² The example above would reduce to a *cast error*:

`10⟨Int⇒?⇒String⟩ ++ "str"`

For type checking, a *consistency* relation $\tau_1 \sim \tau_2$ is introduced. This is a weakening of the type equality requirements in normal static type checking, allowing *consistent* types to be used instead. Where every type τ is consistent with the dynamic type `?`.

$$\frac{}{\tau \sim ?} \quad \frac{}{\tau \sim \tau} \quad \frac{\tau_1 \sim \tau_2}{\tau_2 \sim \tau_1} \quad \frac{\tau_1 \sim \tau'_1 \quad \tau_2 \sim \tau_2}{\tau_1 \rightarrow \tau_2 \sim \tau'_1 \rightarrow \tau'_2}$$

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

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau'_2 \quad \tau_1 \blacktriangleright \rightarrow \tau_2 \rightarrow \tau \quad \tau_2 \sim \tau'_2}{\Gamma \vdash e_1(e_2) : \tau'_2}$$

Where $\blacktriangleright \rightarrow$ extracts the argument and return types from a function type, used to account for if $\Gamma \vdash e_1 : ?$, where we treat `?` then as a dynamic function $? \blacktriangleright \rightarrow ? \rightarrow ?$. Intuitively, $e_1(e_2)$ has type τ'_2 if e_1 has type $\tau'_1 \rightarrow \tau'_2$ or `?` (then treated as $? \rightarrow ?$), and e_2 has type τ_1 which is consistent with τ'_1 and hence is assumed that it can be passed into the function.

But, for evaluation to work the static type information needs to be encoded into casts to be used in the dynamic internal language, for which the evaluation semantics are defined. This is done via *elaboration*, similarly to Harper and Stone's approach to defining (globally inferred) Standard ML [58] by elaboration to an explicitly typed internal language XML [65]. The *elaboration judgement* $\Gamma \vdash e \rightsquigarrow d : \tau$ reads 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:

$$\frac{\Gamma \vdash e_1 \rightsquigarrow d_1 : \tau_1 \quad \Gamma \vdash e_2 \rightsquigarrow d_2 : \tau'_2 \quad \tau_1 \blacktriangleright \rightarrow \tau_2 \rightarrow \tau \quad \tau_2 \sim \tau'_2}{\Gamma \vdash e_1(e_2) : \tau \rightsquigarrow (d_1 \langle \tau_1 \Rightarrow \tau_2 \rightarrow \tau \rangle)(d_2 \langle \tau'_2 \Rightarrow \tau_2 \rangle) : \tau}$$

If e_1 elaborates to d_1 with type $\tau_1 \sim \tau_2 \rightarrow \tau$ and e_2 elaborates to τ'_2 with $\tau'_2 \sim \tau_2$ then we place a cast³ on the function d_1 to $\tau_2 \rightarrow \tau$ and on the argument d_2 to the function's expected

²i.e. the proposed *dynamic type system* above.

³This cast is required, as if $\tau_1 = ?$ then we need a cast to realise that it is even a function. Otherwise $\tau_1 = \tau_2 \rightarrow \tau$ and the cast is redundant.

argument type τ_2 to perform runtime type checking of arguments. Intuitively, casts must be inserted whenever type consistency is used, but deciding which casts to insert is non-trivial [28].

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* [34] takes on a more algorithmic definition of typing judgements, being more intuitive to implement. They also allow some amount of local type inference [59].

This is done in a similar way to annotating logic programs [76, p. 123], by specifying the *mode* of the type parameter in a typing judgement, distinguishing when it is an *input* (type checking) and when it is an *output* (type synthesis).

We express this with two judgements:

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

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

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

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

When designing such a system care must be taken to ensure *mode correctness* [73]. 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:

$$\frac{\Gamma \vdash e_1 \Leftarrow \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 \Leftarrow \tau_1}{\Gamma \vdash e_1(e_2) \Leftarrow \tau_2}$$

We try to *check* e_2 with input τ_1 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:

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 \Leftarrow \tau_1}{\Gamma \vdash e_1(e_2) \Leftarrow \tau_2}$$

Where τ_1 is now known, being *synthesised* from the premise $\Gamma \vdash e_1 \Rightarrow \tau_1 \rightarrow \tau_2$. 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.

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 [23] 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.⁴ The intuition should be clear from the previous gradual and bidirectional typing sections.

⁴The difficulties combining gradual and bidirectional typing are largely orthogonal to adding holes.

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.

Notating $\langle\!\!\rangle^u$ or $\langle\!e\!\rangle^u$ for empty and non-empty holes respectively, where u is the *metavariable* or name for a hole. Internal expression holes, $\langle\!\!\rangle_\sigma^u$ or $\langle\!e\!\rangle_\sigma^u$, 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.⁵

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

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

External Language

Holes synthesise the *dynamic type*, a natural choice made possible by the use of gradual types:

$$\text{SNEHole} \frac{\Gamma \vdash e \Rightarrow \tau}{\Gamma \vdash \langle\!e\!\rangle^u \Rightarrow ?} \quad \text{SEHole} \frac{}{\Gamma \vdash \langle\!\!\rangle^u \Rightarrow ?}$$

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:

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

Of course, e should type check against τ if it can synthesise a consistent type. The goal of type consistency is to allow type treating terms as if they were of any consistent type.

Internal Language

The internal language is explicitly typed with typing judgement, $\Delta; \Gamma \vdash d : \tau$. Where Δ is a *hole context*, mapping each hole *metavariable* u to its *checked type* τ ⁶ and its type context Γ under which the hole was typed.

Elaboration

Cast insertion is performed by elaborating to the *internal language*, and must also output an additional context for holes: $\Gamma \vdash e \Leftarrow \tau \rightsquigarrow d : \tau' \dashv \Delta$ and $\Gamma \vdash e \Rightarrow \tau \rightsquigarrow d \dashv \Delta$.

The resulting hole context will record each hole's original *analysing* type along with the typing assumptions for its hole closure. Recording them instead as $?$ would lose type information.

A well-typed external expression elaborates to a well-typed internal expression *consistent* with the external type.

⁵This is required, as holes may later be substituted with terms containing variables, receiving their values from the closure environment.

⁶As originally required when typing the external language expression.

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 casts which cannot be further reduced.
- *Indeterminate Final Forms* – Terms containing holes that cannot be directly evaluated, e.g. holes or function applications where the function is indeterminate, e.g. $\llbracket \cdot \rrbracket^u(1)$.

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: $(\lambda x.x)(\llbracket \cdot \rrbracket^u)$ can evaluate to $\llbracket \cdot \rrbracket^u$.

Dynamics

A small-step contextual dynamics [29, ch. 5] is defined on the internal expressions to define a *call-by-value* evaluation order, values in this sense are *final forms*.

Like the *refined criteria* [33], Hazel presents a rather different cast semantics designed around *ground types*, that is, base types (**Bool** etc.) and least precise compound types, e.g. $\text{Bool} \rightarrow \text{Bool} \blacktriangleright_{\text{ground}} ? \rightarrow ?$. This formalisation more closely represents common dynamically typed language implementations which only use generic type tags like *fun*, corresponding to the ground type $? \rightarrow ?$. However, the idea of type consistency checking when *injections* meet *projections* remains the same, with projections/injections now being to/fro *ground types*.

Hole Substitutions

Holes are indexed by *metavariables* u , and can hence also be substituted. Hole substitution is a *meta* action $\llbracket d/u \rrbracket d'$ 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:

$$\llbracket d/u \rrbracket \llbracket \cdot \rrbracket_\sigma^u = \llbracket d/u \rrbracket \sigma d$$

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 contextual modal type theory [45].

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.

2.1.3. The Hazel Implementation

The Hazel implementation [2] is written primarily in ReasonML and OCaml with approximately 65,000 lines of code using advanced OCaml features: GADTs, functors, various preprocessors [37, ch. 9, 10, 25], and custom let operators [13]. It is under very active development, with much of the code being undocumented; this report refers to the implementation as of April 2025.

Language Features

Hazel implements the afore mentioned core calculus along with many relevant additional features:⁷

- **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.

⁷Additionally, Hazel supports other features outside the scope of this report.

- **Tuples & Labelled Tuples**⁸ – Allowing compound terms to be constructed and typed [54, ch. 11.7-8].
- **Sum Types** – Representing a value as one of many labelled variants, each of possibly different types [54, ch. 11.10].
- **Type Aliases** – Binding a name to a type, used to improve code readability or simplify complex type definitions.
- **Pattern Matching** – Checks a value against a pattern and deconstructs it accordingly, binding its sub-structures to variable names.
- **Explicit Polymorphism** – System F style parametric polymorphism [54, ch. 23]. Where explicit type functions bind arbitrary types to names, which may then be used in annotations. Polymorphic functions are then applied to a type, uniformly giving the corresponding monomorphic function. Implicit and ad-hoc polymorphic functions can still be written as dynamic code, without use of type functions, but are untyped.
- **Iso-Recursive Types** – Types defined in terms of themselves, allowing the representation of data with potentially infinite or *self-referential* shape [54, ch. 22-23], for example linked lists or trees.

2.1.4. Bounded Non-Determinism

Input generation for a witness search procedure [31] is a *non-deterministic* algorithm [77]. 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:

$$<||> : \tau \rightarrow \tau \rightarrow \tau$$

$$\text{fail} : \tau$$

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

$$x <||> (y <||> z) = (x <||> y) <||> z \quad \text{fail} <||> x = x = x <||> \text{fail}$$

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. Chapter F reasons and details other options considered: continuations, effect handlers, tagless final DSLs, direct implementation.

⁸Merged towards the end of the project's development.

Monadic Non-determinism: Some 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`.

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

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

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

Implementation

This section details the slicing theories developed and implementations of type slicing, cast slicing, and the search procedure.

3.1. Type Slicing Theory

I developed a novel method, *type slicing*, to aid programmers in understanding *how* a bidirectional type system works. First, I define typing slices. Then, three slicing criteria, each associating typing derivations with different explanatory slices.

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

The second and third criterion were *very challenging* to formalise, requiring non-obvious mathematical machinery: *context typing slices* (section 3.1.2), *checking contexts* (section 3.1.5), and *type-indexed slices* (section 3.1.3). Only the basic definitions are given here, the full theory is found in chapter C.

Scope of the theory: The formal theory is developed over the Hazel *core calculus* (chapter B): base types, functions, the dynamic type, holes, and casts. In particular, the theory does *not* treat case expressions, sum types, or product types. The implementation supports these (section 3.3), but the metatheoretic claims in this chapter are made for the core calculus *only*; several fail in richer calculi [8].

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:

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

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

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

We can then define a *precision* partial order [79] on term slices: $\varsigma_1 \sqsubseteq \varsigma_2$ meaning ς_1 is less or equally precise than ς_2 . That is, ς_1 matches ς_2 structurally except that some sub-terms may be gaps. For example:

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

Lattice Structure: For any *complete term* t (having no gaps), the slices of t form a *bounded lattice structure* [78]. That is, every pair ς_1, ς_2 has a *join* $\varsigma_1 \sqcup \varsigma_2$ and *meet* $\varsigma_1 \sqcap \varsigma_2$. In general, not all slices have joins: $1 \not\sqcup 2$, but do have meets as $\square \sqsubseteq \varsigma$ for all ς .

Typing Assumption Slices

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

Hence, their slices are partial functions to *type slices*. Such that, a slice maps no more variables to no more precise types. This, and meets and joins, can be defined pointwise:

Definition 1 (Typing Assumption Slice Precision). *For typing assumption slices γ_1, γ_2 . Where $\text{dom}(f)$ is the set of variables for which a partial function f is defined:*

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

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

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

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

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 componentwise to term typing slices with all the same properties.

Typing: *Expression slices can be type checked under the type assumption slices by replacing gaps $\square$ by: holes of any metavariable $\llbracket \cdot \rrbracket^u$ in expressions, fresh variables in patterns, and the dynamic type in types. Notated by $\llbracket \cdot \rrbracket$.*

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

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:

$$(\lambda x. \underline{\llbracket \cdot \rrbracket^u}) : \text{Bool} \rightarrow \text{Int}$$

Contexts and Their Slices

We represent these surrounding contexts by a *term context* $\mathcal{C}$, which marks *exactly one* sub-term as $\bigcirc$. Where $\mathcal{C}\{t\}$ substitutes term t for the mark $\bigcirc$ in $\mathcal{C}$, only allowed if the marked position expects a term of the same class as t (pattern **Pat**, type **Typ**, or expression **Exp**). Notate the classes by $\mathcal{C} : \mathbf{X} \rightarrow \mathbf{Y}$. Context composition can be defined as substituting contexts into contexts if their classes match. It is *crucial* that composition is defined *syntactically*, and not pointwise, allowing syntactic deconstruction of contexts (used in AFun rule in appendix fig. C.2).

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

Definition 4 (Context Precision). *If $c : \mathbf{X} \rightarrow \mathbf{Y}$ and $c' : \mathbf{X} \rightarrow \mathbf{Y}$ are context slices, then $c' \sqsubseteq c$ if and only if, for all terms t of class $\mathbf{X}$, that $c'\{t\} \sqsubseteq c\{t\}$.*

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

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

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

Typing Assumption Contexts and Their Slices

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

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

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

Context Typing Slices

Finally, an *expression context typing slice*, p is a context slice with each sub-context recursively tagged with typing assumption context slices. Lattice relations are defined componentwise. Composition again must be carefully defined *syntactically* (appendix definition 37).

Type Checking: An analogous $\llbracket \cdot \rrbracket$ translation can be defined.

3.1.3. Type-Indexed Slices

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

$$\underline{(\lambda x. \llbracket \cdot \rrbracket^u)} : \text{Bool} \rightarrow \text{Int}$$

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

$$\underline{(\lambda x. \llbracket \cdot \rrbracket^u)} : \text{Bool} \rightarrow \text{Int}$$

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

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

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

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

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

$$\overline{p} = p$$

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

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

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

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

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

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

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

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

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

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

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

Application, notated $|>$, can be defined similarly, converting indexed context slices into valid indexed expression slices. The opposite direction is more difficult and can be found in appendix (section C.3.4).

3.1.4. Criterion 1: Synthesis Slices

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

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

Definition 7 (Synthesis Slices). *For a synthesising expression, $\Gamma \vdash e \Rightarrow \tau$. A synthesis slice is an expression typing slice ς^γ of e^Γ which also synthesises τ , that is, $\llbracket \gamma \rrbracket \vdash \llbracket \varsigma \rrbracket \Rightarrow \tau$.*

Proposition 4 (Minimum Synthesis Slices). *A minimum synthesis slice of $\Gamma \vdash e \Rightarrow \tau$, under $\sqsubseteq$, exists (and, being a least element, is necessarily unique).*

This property is particular to the core calculus: in extensions of the calculus (with, for example, case expressions) there are derivations with multiple incomparable *minimal* synthesis slices, hence no minimum, and finding a minimal slice of minimum *size* is NP-hard [8].

These slices can be calculated via a typing judgement $\Gamma \vdash e \Rightarrow \tau \mid \mathcal{S}$, meaning $\mathcal{S}$ is the minimum type-indexed synthesis slice of e^Γ synthesising τ . The judgement rules mimic Hazel's typing rules, giving an algorithm to calculate minimum synthesis slices (see section C.5).

3.1.5. Criterion 2: Analysis Slices

A similar idea can be devised for type analysis, represented using *context slices*. After all, it is the terms immediately *around* the sub-term where the type checking is enforced. For example, when checking this annotated term:

$$(\lambda x . \underline{\llbracket \rrbracket}^u) : \mathbf{Bool} \rightarrow \mathbf{Int}$$

The *inner hole term* $\underline{\llbracket \rrbracket}^u$ (underlined) is required to be consistent with **Int** due to the annotation and lambda constructor present in its context. The analysis slice will be:

$$(\lambda x . \underline{\llbracket \rrbracket}^u) : \mathbf{Bool} \rightarrow \mathbf{Int}$$

In other words, if the context slice was type checked, then the inner hole would *still* be required to analyse against **Int**. However, the overall synthesised type of the whole context may differ, the above would synthesise $? \rightarrow \mathbf{Int}$ vs. the original **Bool** $\rightarrow$ **Int**.

Checking Context

We only want to consider the smallest context *scope* 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:

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

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

$$(\lambda x : \text{Int}. \llbracket \cdot \rrbracket^u)(\underline{1})$$

Definition 8 (Checking Context). *For term e checking against τ : $\Gamma \vdash e \Leftarrow \tau$. A checking context for e is a context typing $\mathcal{C}^{\mathcal{F}}$, an expression context $\mathcal{C}$ tagged with a typing assumption context $\mathcal{F}$ (section 3.1.2), such that:*

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

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

Any expression has infinitely many checking contexts. But importantly, for a sub-term occurrence within a particular program there are only finitely many checking contexts, and exactly one *minimally scoped* checking context matching the structure of the program (appendix proposition 26). All minimally scoped checking contexts can be constructed syntactically via rules (appendix section C.4). Analysis slices are slices of minimally scoped checking contexts.

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

Proposition 5 (Minimum Analysis Slices). *A minimum analysis slice of $\Gamma \vdash e \Leftarrow \tau$ in a checking context $\mathcal{C}^{\mathcal{F}}$, under $\sqsubseteq$, exists.*

As for synthesis slices, this holds for the core calculus only: in richer calculi minimal analysis slices need not be unique, and no minimum need exist [8].

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

$$\Gamma; \mathcal{C} \vdash e \Leftarrow \tau \mid \mathcal{S}$$

The rules build upon rules defining minimally scoped checking contexts, and are more complex, making use of *type-indexed* synthesis slices, see fig. 3.1. This requires an involved conversion of (indexed) expression slices to context slices (section C.3.4).

3.1.6. Criterion 3: Contribution Slices

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

$$(\lambda f : \text{Int} \rightarrow ?. f(1))(\underline{\lambda x : \text{Int}. x})$$

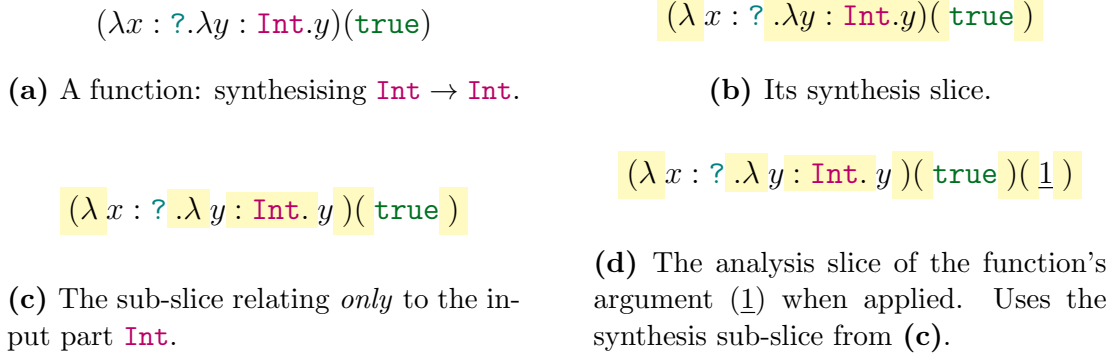


Figure 3.1: Demonstration: Analysis slice application uses synthesis slices

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

$$(\lambda f : \text{Int} \rightarrow ?. f(1)) (\lambda x : \text{Int}. x)$$

Notice that sub-terms which do not contribute must be dynamically annotated. Therefore, this criterion omits the dynamic code regions.

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

3.2. Cast Slicing Theory

Cast slicing propagates type slice information during evaluation, by tagging casts types with type slices. A primary reason in formalising type-indexed slices was to make slices decomposable during evaluation, retaining only sub-slices relevant to the part of the type involved in the decomposed cast. The first two criteria work together during elaboration, analysis slices inserted when casting on a elaborated expression who had it's type *analysed*, and the synthesised for synthesis slices. The rules are very involved, but build directly upon the Hazel calculus elaboration semantics; the rules are found in section C.9.

3.3. Type Slicing Implementation

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

3.3.1. Hazel Terms

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

3.3.2. Type Slice Data-Type

Expression slices as ASTs

Storing expression slices directly as ASTs is both *space and time inefficient*, even when accounting for the persistence [62, ch. 2] 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

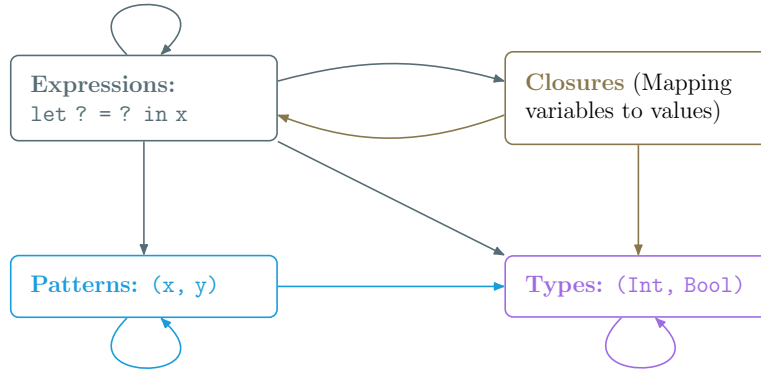


Figure 3.2: Mutually Recursive Hazel Terms.

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

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

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 tyslice_typ_term =
  | Unknown
  | Arrow(slice_t, slice_t) // Function type
  | ... // Type constructors
and tyslice_term = (tyslice_typ_term, code_slice)
and tyslice_t = IdTagged.t(slice_term)

```

Figure 3.4: Initial Type Slice Data-Type

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

Incremental Slices

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

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

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

Polymorphic Variants [37, ch. 7.4], notated `[ | ... ]` are used to more conveniently write functions on slices. This is possible due *row polymorphism* [63] [14, ch. 10.8] relating the variants by a *structural subtyping* relation [70]. We have that:¹

```
tyslice_empty_term :> tyslice_incr_term :> tyslice_term
```

¹ $x := y$ meaning x a subtype of y .

```

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

```

Figure 3.5: The type slice data-type

```

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

```

Figure 3.6: Apply Utility Function

Type constructors are either co-variant or contra-variant [66, ch. 2] with respect to the subtyping relation. For example, id tagging is covariant, so:

`IdTagged.t(typslice_incr_term) :> IdTagged.t(typslice_incr_term) = typslice_t`

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

`typslice_incr_term → typslice_incr_term :> typslice_empty_term → typslice_term`

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

Utility Functions: Functions on slices often only concern the underlying type, e.g. checking if a slice is a list type. Writing direct pattern matching code on `typ_term` and `typslice_term` is easier. An `apply` function can apply these direct functions to the term inside a slice. The bottom two branches can both be passed into `apply` function as they are sub-types of `typslice_term`. Many other utility functions are implemented, including mapping functions, wrapping functions, unpacking functions, matching functions.

Type Slice Joins

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

For basic type slices, when the same type information is available from multiple branches, highlighting only one branch is required, but both for contribution slices. See that the 1 in

fig. 3.7 is not highlighted (in green). However, we did still need some information from the left (the 0).

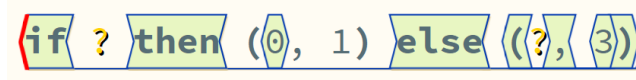


Figure 3.7: Type Slice Joins (note: Hazel syntax uses ? also for expression holes)

This gets complex, requiring tracking all the data relating to which branches should be highlighted, and accumulating the inconsistent parts of failed joins for use in minimised error slices (demonstrated in section 4.8.1). I define custom OCaml let bindings [13] to retain clarity while automatically combining branches in successful joins and combining inconsistencies in failed joins. For example, see fig. 3.8, where comments describe the logic abstracted by the custom bindings; note that all join-able type combinations use this same parallel binding logic to combine branches and inconsistencies.

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

```

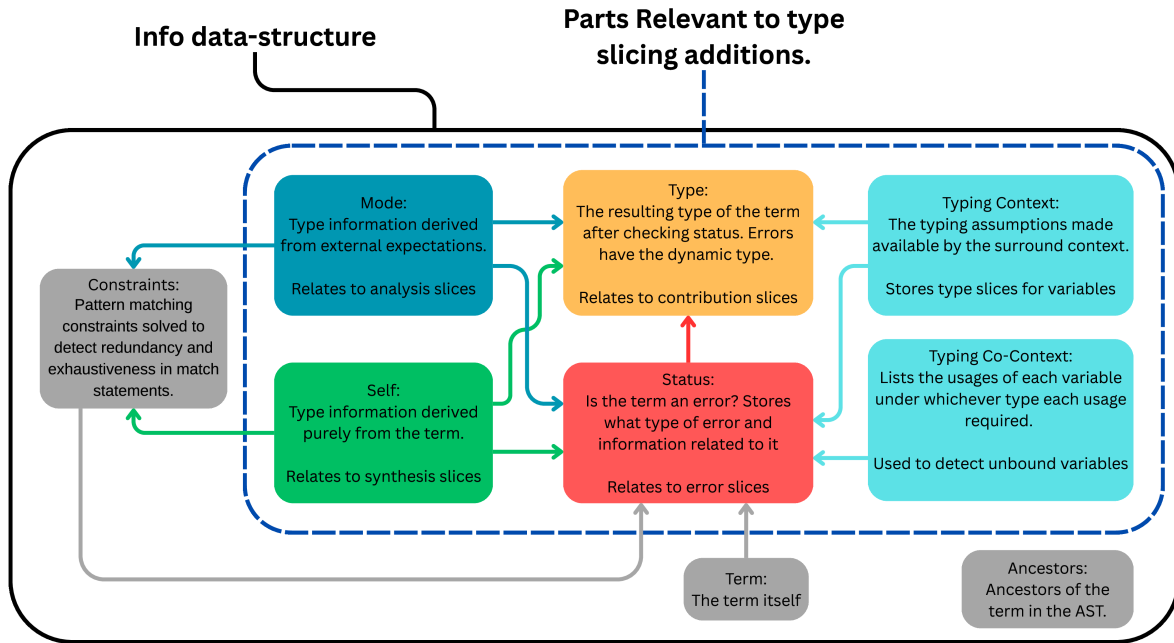
Figure 3.8: Joining function types

3.3.3. Static Type Checking

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

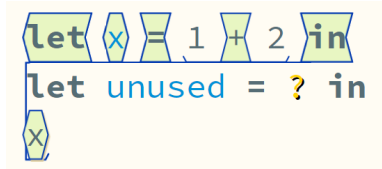
Self and Mode

Slicing logic relating to synthesis slices and analysis slices is factored into `Self.t` and `Mode.t` respectively, cleanly segregated from the type checking code. Doing this still required full understanding of the type checker implementation, ensuring the correct IDs are sliced.

Figure 3.9: `Info.t` data-structure

Typing (Co-)Context

The typing context and co-contexts are modified to use type slices. This deviates from the theoretical notion of an expression slice: the structural context in which the variable is used is untracked when passing through the context. Therefore, it requires using *unstructured* code slices. It is useful in practice, allowing slices calculated during binding to be retrieved upon usage, see fig. 3.10.

Figure 3.10: Type slice for variable `x`: includes its binding and slice.

Status and Type

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

3.3.4. Integration

To support nearly the full Hazel language, type slices needed to implement many functions, for example: type substitution, type normalisation, weak-head normalisation, tracking sum types, various structural matching functions etc. Additionally, almost every usage of types in the codebase had to be refactored to use type slices (which are not so easily pattern matched upon) while ensuring slices were correctly maintained.

3.3.5. User Interface & Examples

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

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

(a) `None` synthesises `IntOption` due to its type definition.

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

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

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

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

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

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

Figure 3.11: Type Slices

3.4. Cast Slicing Implementation

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

3.4.1. Elaboration

Cast insertion recursively traverses the unelaborated term, inserting casts to the term's statically determined type as stored in the `Info` data-structure. For example, a list literal recursively elaborates its terms and joins their slices, casting to the join. Ensuring all the type slice information is retained and/or reconstructed during elaboration was a meticulous and error-prone process.

3.4.2. Cast Transitions

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

However, as Hazel only checks consistency between casts between *ground types*, there are two rules where new² casts are *inserted* (ITGround, ITExpand). The new types are created via a *ground matching* relation taking the topmost compound constructor of types, e.g. ground functions fig. 3.12. Relevant portions of the appendix are fig. B.8, fig. B.10, fig. B.11.

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

²As opposed to being derived from decomposition.

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

Figure 3.12: Ground Matching Functions

3.4.3. Unboxing

When we know a final form’s (section 2.1.2) type, we may need to extract parts of the term according to the type during evaluation. For example, extracting the elements of a tuple. But due to casts and holes, this is not trivial [17].

Slicing does not concern unboxing, but indeterminate evaluation (section 3.5) unveiled bugs within which had to be fixed. I raised and fixed these, with my PR eventually being merged into the main branch.

3.4.4. User Interface & Examples

Type slices within casts can be selected from the evaluation result and displayed. This requires 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.

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

≡ 1 + "one": (Int)

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

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

≡ [1: •, •: (Int): •, 3: •]

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

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

≡ float_of_int(•: (Int))

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

Figure 3.13: Cast Slicing Examples

3.5. Indeterminate Evaluation

Dynamic errors include evaluation traces, which can aid in debugging [42], yet static type errors lack such traces. Seidel et al. [31] offer an algorithm to search for these traces for static errors in OCaml by lazily, non-deterministically narrowing input holes to least specific values based on their usage context.

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

method, not limited to cast error searches. Specifics relating to cast errors and implementing search orderings, are covered in section 3.6.

3.5.1. Resolving Non-determinism

To model infinite non-determinism I create a monadic DSL with an explicitly tree/forest-based representation. The forest model allows for varying low-level search traversals. The module type of combinators is in `Nondeterminism.Search`; its underlying parametric type is `t('a)` with `'a` being the type of the solutions. Section 3.6.2 discusses the actual implementations of this interface, giving four searching procedures.

In addition to the functions from Section 2.1.4, I add standard `map` and `join` functions, and various other functions, e.g. `ifte` modelled after Mercury's if-then-else construct [5].

Abstracting Search Order: Forest Model

Typical stream-based models of non-determinism [72] only admit the possibility of depth-first search (DFS). Stream concatenation provides no way of remembering choice points and backtracking before finishing a computation.

Instead, we can use a model based on forests (lists of trees) [50]. Choice, similarly to streams, is performed by concatenating forests. Finally, to build tree structures, a `wrap` combinator wraps every tree up into a single tree with a new root, see fig. 3.14. Effectively, this encodes a notion of cost for search paths.

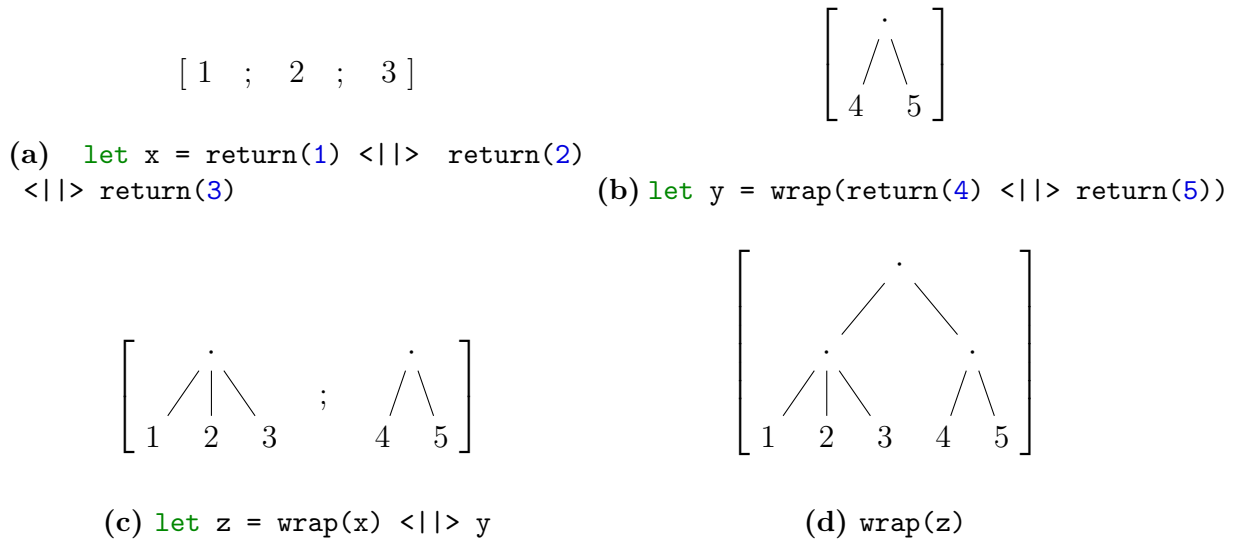


Figure 3.14: Forests Defined Using `wrap`

The DSL is extended with a `wrap : t('a) => t('a)` combinator. As `wrap` is abstract, the underlying implementation does not actually need to use a forest data structure. Therefore, DFS can still be efficiently implemented using regular streams. Finally, a `run` function produces a lazy list of solutions with ordering specified by the search method; these can be cycled through with some arrows in the UI.

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`, recursive functions can be written directly resulting in infinite choices without OCaml's strictness leading to infinite recursion.

3.5.2. A Non-Deterministic Evaluation Algorithm

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

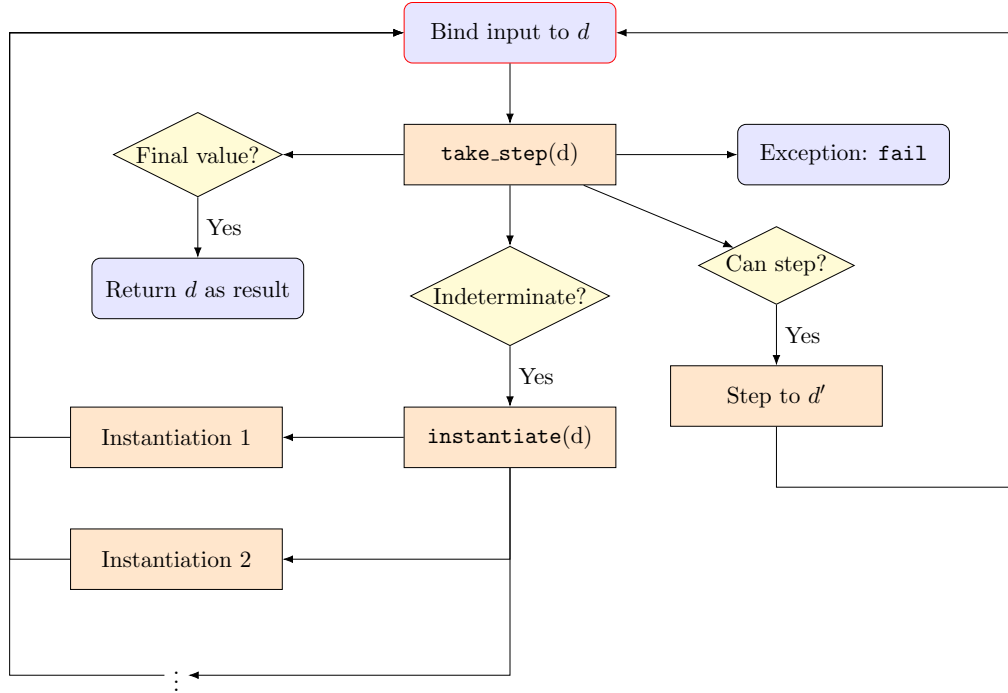


Figure 3.15: Block diagram of indeterminate evaluation to values

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

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

Evaluation steps are performed by a (deterministic) function `take_step`, classifying its input as either (concrete) values, indeterminate values, or steppable expressions. The step evaluator re-uses the Hazel stepper logic, which had existing bugs to be fixed: causing non-termination of fixed points³ and misclassifying concrete values as indeterminate values:

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

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

The generalised indeterminate evaluation takes a higher-order `logic` function which determines the return logic (e.g. returning only results with cast errors).

3.5.3. Hole Instantiation & Substitution

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

Choosing which Hole to Instantiate

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

When attempting to evaluate the indeterminate term some transitions rules require a sub-term to be concrete (e.g. a function during application). We chose to instantiate the hole that

³Due to incorrect management of closures; the main branch stepper is still broken as of May 9th.

```

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

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

```

Figure 3.16: Indeterminate Evaluation to Values

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**), which (rather obscurely⁴) separates this logic from the transition semantics. Therefore, hole choosing logic will automatically update to future changes in the transition semantics.

Synthesising Terms for Types

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

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

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

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

```

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)

```

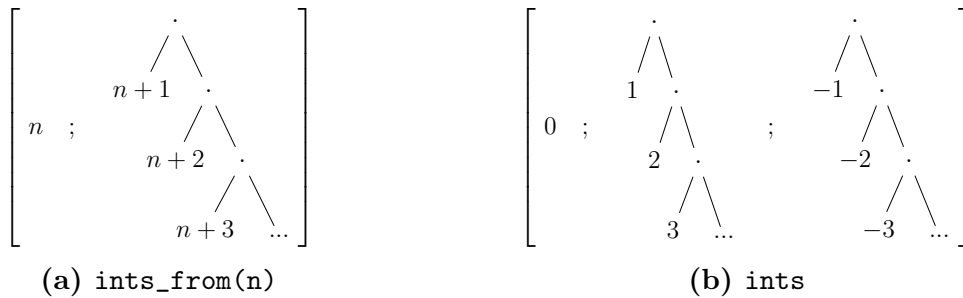


Figure 3.17: Enumerating Integers

⁴Lacking documentations and using advanced custom let operators [13]. Understanding took considerable time.

Strings: A string is either *empty* or is a string with a first character from a finite set. We can recursively wrap all strings, prefixed by each character. See fig. 3.18:

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

Figure 3.18: Enumerating Strings

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

Lists: A list is either the empty list, `[]` or a cons `?1 :: ?2`. To retain the correct dynamic type information, `?2` must be cast back to the list type.

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

Functions: Constant functions have least specific values `λ_. ?`. 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 not yet caught due to partial annotations, these will be lost. This is rare, and could be mitigated by generating the identity function where possible.

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

Substituting Holes

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

Hole substitution was described as part of the Hazel calculus section 2.1.2. But, unexpectedly, the main Hazel branch does not yet implement it. A full implementation of metavariables and delayed closures is complex. Therefore, as hole closures are not required for hole instantiation,⁵ I use existing term IDs identify holes, ensuring these are maintained and propagated correctly, so that duplicated holes retain the same ID.⁶

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

3.5.4. Determining the Types for Hole Substitutions

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

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

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

Cast Laziness: However, this is *not* always the case. For efficiency reasons, Hazel treats casts over compound data lazily, e.g. casts around tuples will only be 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 4.8.3 evaluates the consequence of this choice.

⁵Instantiations do not contain variable references.

⁶All of elaboration and dynamics required these checks.

```

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

≡ (⬤, ⬤)

```

Figure 3.19: Duplicated Holes

```

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

```

Figure 3.20: Dynamic/Type-case Match Statement

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

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

Extended Match Expression Instantiation (Pattern Instantiation)

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

3.6. Search Procedure

Now that a framework for indeterminate evaluation has been specified, the DSL implementations and cast error search logic can be addressed. The error search procedure can be switched on via a toggle in the settings.

3.6.1. Detecting Relevant Cast Errors

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

Multiple Cast Failures: Terms may have multiple cast failures, some of which discovered during static type checking and inserted via elaboration. But these failures will not stop evaluation until necessary. Therefore, we should consider only the casts which are directly *causing* a term to get stuck, this is implemented similarly to choosing which hole to instantiate (section 3.5.3).

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

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

3.6.2. Searching Methods

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

Depth First Search (DFS): Modelling DFS by streams is a typical method [72]: implementing choice and conjunction via appending, and `wrap` being identity function (no internal tree structure needed).

Breadth First Search (BFS): Breadth first search represents forests by sequences of sequences [60], with the n th inner sequences representing every solution in the n th level of the trees in the forest. Then, `choice` concatenates each level, and `wrap` conses the empty list, pushing every solution one level down.

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

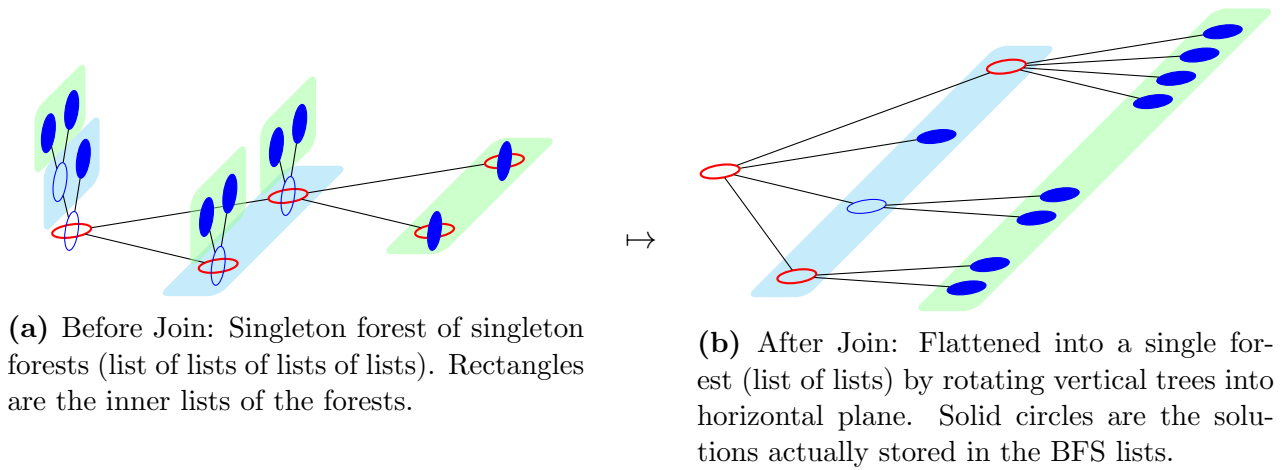


Figure 3.21: BFS Join

This join and mapping can be defined lazily; the standard definition for `bind` from the paper [50] does not work easily in OCaml due to its strictness.

Bounded Depth First Search (BDFS): BFS is fair, avoiding non-termination (for finite branching factor), but it has *exponential* space complexity in the depth of the level being explored [30]. In comparison, DFS only requires space linear in the depth explored.

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

Spivey [50] represent this by functions `Int => ['a, Int]`, calculating solutions within an integer depth bound. Solutions are tagged with their remaining depth budgets, which allows keeping only the fringe solutions (zero budget) upon each iteration, so the same solution will not appear twice.

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

Interleaved Streams (IDFS): Streams, but with choice and conjunction via interleaving also ensures fairness. This works even for trees with infinite branching factor. However, in-

terleaving has a linear space complexity, resulting in exponential space complexity as with BFS.

3.7. Implementation Availability

The implementation is available in the Hazel source repository [2]. The work described in this report is on the `witnesses-type-slicing` branch, deployed at <https://hazel.org/build/witnesses-type-slicing/>. The evaluation branch used to run the command-line experiments is `Evaluation`; it factors out a `Js_of_ocaml` [3] dependency from the Hazel core library so that evaluation code can run natively, at the cost of disabling the web interface on that branch.

The companion Part III technical report [8] uses the later Hazel branch `type-slicing-v2`, deployed at <https://hazel.org/build/type-slicing-v2/>. That branch should be treated as separate from the prototype evaluated here: it reflects the later type-slicing work, whereas this report's implementation also includes cast slicing and dynamic witness search.

Evaluation

This chapter evaluates the implemented type slicing, cast slicing, and witness search features. Supplementary results are found in chapter E.

4.1. Success

As demonstrated below, the type witness search procedure and slicing features cover the central static and dynamic error-explanation cases considered in this report. Extended pattern instantiation (section 3.5.4) was partially implemented, but is not required for the main evaluation results.

The implementation supports most Hazel features used in the evaluation, except that type slicing of type functions and labelled tuples¹ remains only partially functional.

4.2. Evaluation Criteria

The evaluation concerns three features: *type slicing*, *cast slicing*, and a *static type error witness search procedure*. Each has a distinct explanatory role:

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

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

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

4.3. Methodology

I evaluate the features and their various implementations (where applicable) along *four* axes. Quantitative measures were evaluated over a corpus of ill-typed and dynamically-typed Hazel programs (section 4.5):

Quantitative Analysis

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

The time and space usage of the search procedure implementations were micro-benchmarked for each ill-typed program in the corpus. Up to 100 runs were taken per program with estimated time, major and minor heap allocations were estimated using an ordinary linear regression (OLS) via the *bechamel* library [9].

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.²

¹Labelled tuples were only merged late in the development cycle.

²Not necessarily *always* true, but a reasonable assumption.

Qualitative Analysis

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

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

4.4. Hypotheses

I made several hypotheses and investigated each. The evidence and implications being discussed in the following sections.

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 in Errors: Casts are deconstructed 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 [38] 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 [53] and answer-set programs [40]. 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 were produced, containing both dynamic (unannotated) programs and annotated programs (containing statically caught errors). We have made this corpus available on GitHub [10].

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 [26], 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 [11]. 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.

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.

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 4.1: Benchmarks: Performance ratios to DFS over common programs

4.5.2. Statistics

The program corpus contains **698** small programs, **294** were annotations, of which **203** were applicable to performing the search procedure on.

4.6. Performance Analysis

4.6.1. Slicing

The type and cast slicing mechanisms do not increase the time complexity of the type checker nor evaluator. Hence, they are still as performant as the original, capable of interactive use for medium sized programs.

4.6.2. Search Procedure

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

As the search procedure may be non-terminating, these results are found given a 30s time limit. Micro-benchmarking the programs which do not time-out, the time and space used searching for each witness can be estimated.

The performance ratios between each implementation as compared to DFS on only the programs which *both terminate* are given in fig. 4.1. As expected, BFS and IDFS use more memory in total than BDFS and DFS, while DFS is the fastest.

4.7. Effectiveness Analysis

4.7.1. Slicing

Type slice sizes (amount of highlighting) were calculated over the entire corpus. While *cast slice* sizes were calculated over the resulting elaborated expressions.

Figure 4.2 shows that type are generally small. In particular, the proportion of the context³ highlighted is very low, generally less than 5% for dynamic code and 10% for annotated code. Therefore, they concisely explain the expressions' type on average.

Additionally, for errors, there are 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.

As hypothesised, combined slices within cast errors are smaller than combined static error slices on average (2x, fig. E.3). Therefore, casts can more precisely point to which part of an expressions type caused them.

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

³Calculated 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					
	unit	ok	Combined Error Slices			Minimised Error Slices		
			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
<i>(Unannotated)</i>								
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
<i>(Annotated)</i>								

* only 2 annotated programs had inconsistent branches

Figure 4.2: Effectiveness: Type Slices



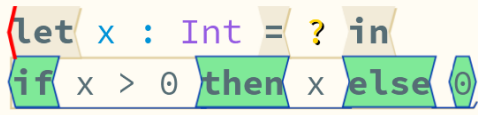
Figure 4.3: Search Procedure Coverage

directly (fig. 4.3). IDFS and BFS perform relatively poorly likely due to excessive memory usage (fig. E.2).

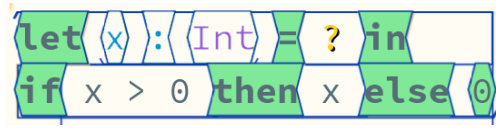
However, not all static errors have a dynamic witness, e.g. errors within dead code. I manually classified each failed program for BDFS to check if a witness does exist, but was not found, or no witness exists. This gives only 2% of cases where BDFS failed to actually find an *existing* witness; DFS and IDFS also meet the goal of failing in less than 25% of cases. Section 4.8.3 goes into further detail on categorising the programs which time out, and how this could be avoided.

Witness & Trace Size

As predicted by the small-scope hypothesis, most programs admitted *small* witnesses (avg. sizes of 1-2, i.e. mostly base cases).



(a) Ad-hoc Slice: Let expression omitted



(b) Ordinary Slice

Figure 4.4: An Ad-hoc Slice vs. Ordinary Slice

However, there was no linear correlation (Pearson correlation coefficient = 0) between witness sizes and trace sizes, even when normalised by the original deterministic evaluation trace lengths. This is likely because most errors are in the base cases, so few large witnesses are even found, with the noise from trace lengths to different programs' base cases dominating.

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 were devised.

4.8.1. Slicing

Type slicing theory (section 3.1) requires highlighted code to form valid expressions or contexts, though some highlighted parts, like unused or dynamic bindings, do not affect types and can be omitted. This motivated the use of unstructured (ad-hoc) slices (section 3.3.2).

In fig. 4.4, a bound integer x is only used in one branch. Since the other branch can already determine the type of the conditional, the x and `let` are excluded from the slice. Though the whole program is selected (see the red cursor) the let expression is omitted. A contribution slice would include everything, making it even more verbose.

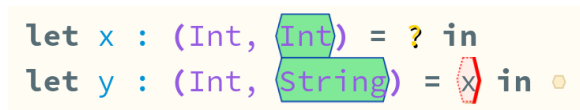
Error Slices

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

Some type parts may agree, hence another form of type joining was introduced to isolate only the inconsistent parts. For instance, in fig. 4.5, differences like (Int, Int) vs $(\text{Int}, \text{String})$ highlight only the mismatched sub-slices.



(a) Partially Inconsistent Branches



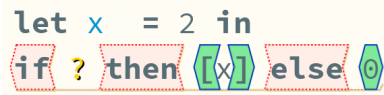
(b) Partially Inconsistent Expectations

Figure 4.5: Error Slices

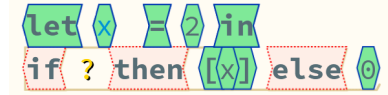
Compound type inconsistencies necessarily differ at the outermost constructor (e.g., `List` vs `Int`), being the *primary* cause of the error. Deeper inconsistencies are not those causing the type error. Therefore, we could extract only the slice on the outermost constructor. These minimised slices (fig. 4.6) are significantly (avg. 3x) smaller than full combined slices (fig. 4.2). This ratio may grow with program complexity, the corpus had few partially inconsistent cases.

4.8.2. Static-Dynamic Error Correspondence

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



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



(b) Ordinary Error Slice

Figure 4.6: Minimised Error Slices

However, for inconsistent branches, no direct cast failure occurs until a branch result is used in a static context. Still, since elaboration adds casts to each branch, these can be tracked with the error. But, we cannot distinguish if the error was caused by the branch or the use.

4.8.3. Categorising Programs Lacking Type Error Witnesses

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

- Witness Exists: BDFS failed to find an existing witness.
- Dead Code: The error lies in unreachable code:
 - Pattern Cast Failure: Error within a pattern matching branch, making the branch unreachable. These are detectable by the extended pattern directed instantiation algorithm (section 3.5.4).⁴
 - Unbound Constructor: Attempting to match an unbound constructor. Also detectable with the extended instantiation.
 - Wildcards: Erroneous code bound to the inaccessible wildcard pattern:


```
let _ = ... in ....
```
 - Non-Trivial: Less easily detectable. One example exhibited this, infinitely recursing for all inputs.
- Hazel Bugs: Unboxing bugs present in the main branch (excluded from the statistics).

Figure 4.7 shows this distribution and three (paraphrased) examples are given in the appendix fig. E.5. The full classification is in `failure-classification.txt`.

Non-Termination, Unfairness, and Search Order

DFS is fast but prone to non-termination, giving it lesser coverage than BDFS (fig. 4.3). If evaluation infinitely loops, so will DFS without exploring other instantiations. It is also unfair, it may never try some instantiations [51]. This affected 10% of cases.

To address this, BDFS, BFS, and IDFS were implemented, with BDFS performing the due to prioritising evaluation over instantiation. BFS and IDFS would do significantly better if they tried longer trace length more frequently, on average reaching only half the trace length (full data: fig. E.4). Wrapping evaluation in the evaluation algorithm would help this, but choosing how frequently to wrap evaluation is a delicate balancing act.

Dead Code & Nested Errors

Errors within dead code cannot have a witness as they are not dynamically reachable. A significant portion (39/47) of failed programs for BDFS had trivial dead code which could easily be detected and returned explicitly as proving no witness exists.

⁴The inconsistent patterns would be attempted, and subsequently reduced to expression cast failures.

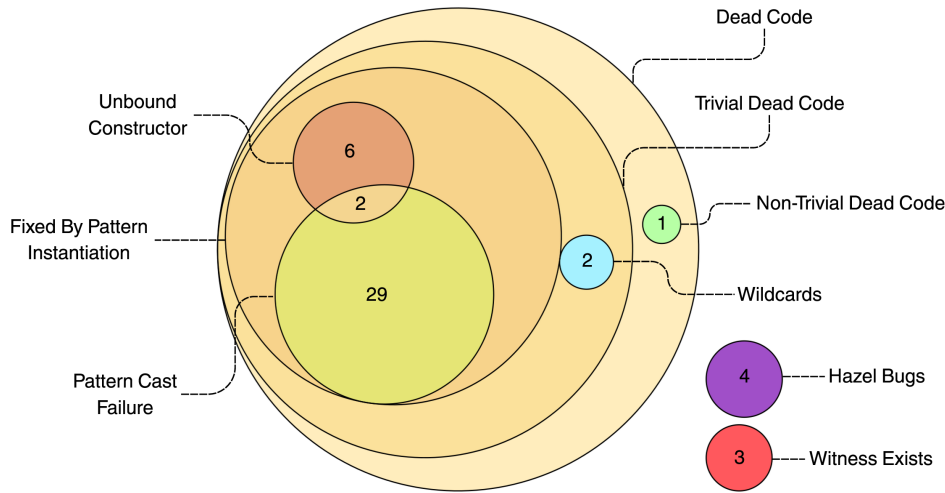


Figure 4.7: Distribution of Failed Program Classes

Additionally, code can become dead due to errors. 12 (32%) of the dead code classified had a nested error within a branch that is unreachable due to a pattern error on the branch pattern. Even if witnesses are found for these branch errors, the nested errors remain hidden.

Cast Laziness

Hazel treats casts lazily, deferring cast transitions on compound data until their elements are used (e.g., tuple elements). Hazel does not detect cast errors between these non-ground compound types (e.g., `[Int]` vs `[String]`) until elements are used. Therefore if such elements are unused in code, the error cannot be witnessed. These are uncommon, none appearing in the search corpus.

Addressing this would require eager cast semantics, as has been previously explored for dynamic and gradual type systems [64, 44] (often referred to as coercions). Additionally, they would catch dynamic errors earlier, for example, fig. 4.8 shows a cast inconsistency undetected (by lazy casts) at runtime until the tuple element is accessed.

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

≡ (1, "str"): (◻, ◻): (Int, Int)

Figure 4.8: Inconsistent Lazy Casts: No Cast Failure

Combinatorial Explosion

When multiple holes are involved in searching, the search space increases exponentially. This combinatorial explosion especially impacts IDFS and BFS, who prioritise instantiation over evaluation. This leads to high memory use and hinders evaluation from progressing far enough to detect errors, even when valid witnesses have been instantiated.

Further, some errors only arise on very specific inputs, e.g. (23, 31) and (31, 23) in fig. 4.9. Directing instantiation to maximise code coverage earlier could find such errors attempting fewer instantiations.

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

Figure 4.9: Witness requiring very specific instantiations

4.8.4. Improving Code Coverage

The search procedure struggles when compound witnesses require specific, interdependent parts, being much less likely to instantiate. These errors are also harder for programmers to detect or understand; understanding the error might require recognizing input interdependencies.

Intelligently directing hole instantiations to better cover the code would help. A pattern-directed instantiation as described in section 3.5.4 would discover 2 of the 3 missed witnesses by BDFS.

Still, BDFS retains a very high coverage over the search procedure when excluding dead code (97%).

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 [15] to debug the errors.

4.9.1. Interaction with Existing Hazel Type Error Localisation

Hazel has three error types addressed here: *inconsistent expectations* (analytic and synthetic types are inconsistent), *inconsistent branches* (branches or list element types are inconsistent), and *inexhaustive matches*.

Section 4.8.2 showed how witnesses can be associated to inconsistency errors. While indeterminate evaluation can give examples for pattern inexhaustivity, standard pattern matrix methods [47] are more efficient. The matrices would also be useful for directing pattern instantiation.

When errors arise from the programmer *misunderstanding* the program types, error localisation can be inaccurate (due to assuming different types to the programmer's expectations). The context inspector (fig. 4.10) clarifies what assumptions the system makes while slicing and witnesses can explain *why*.

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

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

Figure 4.10: Selected Static Error described by Hazel Context Inspector

When the programmer and system agree on the types, bidirectional typing generally localises the error(s) well [34, 15]. However, there is not always enough static information to even recognise errors, and many type annotations may need to be inserted to detect the error. The search procedure tests such code for such type errors automatically, and can be a quicker way to detect these errors than adding annotations.

4.9.2. Examples

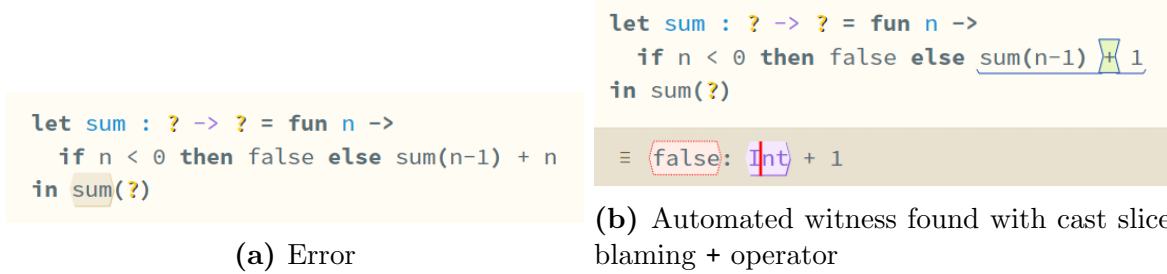


Figure 4.11: Dynamic Code: An error is not statically found. Cast slicing still works even without the type annotations, blaming addition forcing ints.

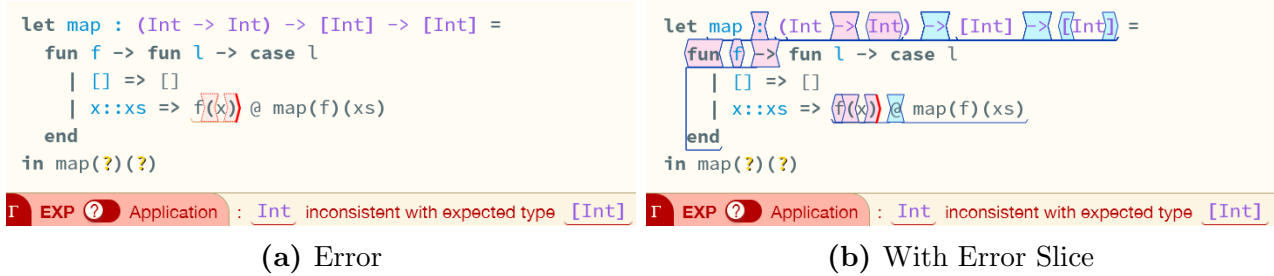


Figure 4.12: Inconsistent Expectations: $f(x)$ synthesises type `Int` but analyses against `[Int]`. The *minimised* error slice shows (in pink) why `Int` is synthesised (due to input `f` being annotated `Int -> Int` and applied) and (in blue) why a list is expected.

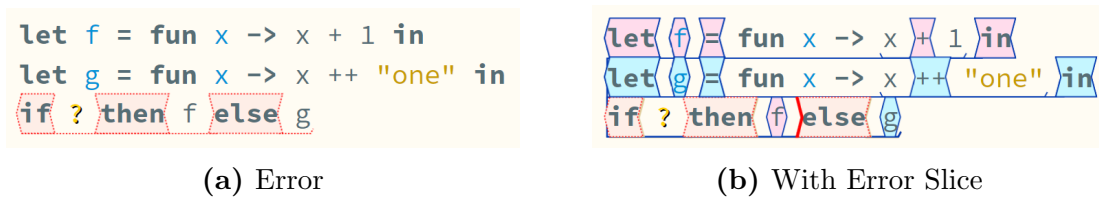


Figure 4.13: Inconsistent Branches: A more complex inconsistency involving non-local bindings. *Minimal* error slices highlight only the bindings, and conflicting addition and string concatenation operators.

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

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

(a) The user thinks that list cons :: concatenates lists.

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

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

(b) Error slice highlights the offending ::, but with considerable noise distracting from this. Also, if the user still expects :: to perform concatenation, this is useless.

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

```
≡ [] : Int:: [] : Int:: []
```

```
≡ [] : Int:: [[] : Int]
```

```
≡ [[] : Int, [] : Int]
```

(c) The user cycles through increasingly larger witnesses (until [[] , []]) to spot the consing pattern. End of trace concretely shows :: acting as cons.

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

```
≡ [[] : Int, [] : Int]
```

(d) Alternatively, the cast slice to Int concisely retrieves the relevant part of the original type slice.

Figure 4.14: Holistic Example involving all three features

```

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

```

(a) Fold takes curried functions, but uncurried add is used.

```

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

```

$\equiv (\langle \text{add} \rangle (0 : (\langle \langle \cdot, \cdot \rangle \rangle : \cdot \rightarrow \cdot)) (0))$

(c) Witness ([0]) found automatically: expects input 0 into add to be a tuple. Cast slice considers only the $(\text{Int}, \text{Int}) \not\sim \text{Int}$ inconsistency.

```

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

```

(b) Error slice is verbose, and merges multiple inconsistencies: $(\text{Int}, \text{Int}) \not\sim \text{Int}$ and $\text{Int} \not\sim \text{Int} \rightarrow \text{Int}$. The second inconsistency is minimised to $\text{Int} \not\sim ? \rightarrow ?$.

```

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

```

$\equiv (\langle \text{add} \rangle (0 : (\langle \langle \cdot, \cdot \rangle \rangle : \cdot \rightarrow \cdot)) (0))$

(d) Same witness has a second cast error: output of add(0) is expected to be a function. Cast slice considers only the minimised $\text{Int} \not\sim ? \rightarrow ?$ inconsistency.

Figure 4.15: A more subtle holistic example involving currying. Requires slice decomposition internally.

Conclusions

This report presented type slicing, cast slicing, and a dynamic witness search procedure for explaining static and dynamic type errors in Hazel. The implementation supports the central Hazel features considered in the evaluation, with only partial support for labelled tuples and type substitution.

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

An indeterminate evaluation framework (section 3.5) was built, greatly generalising the original search procedure [31] to allow multiple search orderings, and different classes of results, e.g. concrete values, or derived results like expression sizes.

5.1. Further Directions

5.1.1. UI Improvements, User Studies

The holistic evaluation suggests several usability and UI improvements. Once implemented, a user study could assess the real world effectiveness of type slicing and cast slicing in improving understanding of the type system, and locating errors.

5.1.2. Cast Slicing

Cast slicing propagates type slice information throughout evaluation. But, it provide no slicing on *execution properties* of the program. Extended slicing methods could, for example, provide a minimal program which evaluates to the *same* cast around the *same* value. This could build on traditional *dynamic slicing* methods [71, 41], and find use also in debugging *semantic* errors.

5.1.3. Property Testing

Indeterminate evaluation provides a way to generate inputs to function. So, it could be used for property testing, generating inputs to functions and testing expressions. A framework similar to SmallCheck [46] or QuickCheck [57] could be implemented. Being part of the evaluator, intermediate expressions and execution properties could also be tested.

5.1.4. Non-determinism, Connections to Logic Programming

Non-deterministic evaluation could be harnessed to implement non-deterministic constructs (e.g. a choice operator) for Hazel. Treating holes as unknowns and instantiating lazily is reminiscent of *free logic variables* in functional logic programming languages [43], e.g. in *Curry* [7]. Adding unification [69] would allow full logic programming in Hazel. A needed-narrowing evaluation strategy [56] would be a significantly more efficient than the current lazy non-deterministic instantiation.

5.1.5. Symbolic Execution

Use of symbolic Execution [24] would improve the proportion of witnesses found by the search procedure by improving code coverage (section 4.8.4). Indeterminate evaluation can be considered a form of simple symbolic execution, only considering constraints enforced by casts. Section 3.5.4 considered directing instantiation using patterns within match statements.

5.1.6. Let Polymorphism & Global Inference

Types in globally inferred languages are often more subtle, as there are fewer annotations asserting the programmers intent. This project would be particularly useful in these situations.

In Hazel: Global inference is difficult or impossible to combine with complex type systems. Global inference for high rank polymorphism, like Hazel’s System-F style polymorphism, is

undecidable [61]. However, a form of let-polymorphism via principal type schemes [74] is possible. Intersecting this with gradual typing has been explored by Garcia and Cimini [32] and Miyazaki et al. [22], the latter of which would integrate most smoothly with Hazel and indeterminate evaluation.

Constraint Slicing: The search procedure and cast slicing are relatively easily extended to a let-polymorphic Hazel in the style of Miyazaki et al. However, type slicing now has to consider non-local constraints alongside code which sourced them, differing significantly from bidirectional type slicing. Somewhat similar ideas, but limited only to errors, have been explored in *constraint-based type error slicing* by Haack and Wells [52].

Bibliography

- [1] *Hazel Project Website*. URL: <https://hazel.org/> (visited on 02/28/2025).
- [2] *Hazel Source Code*. Implementation branches relevant to this report: `witnesses-type-slicing` and `Evaluation`; the companion Part III report uses `type-slicing-v2`. URL: <https://github.com/hazeltgrove/hazel> (visited on 02/28/2025).
- [3] *Javascript of OCaml package*. URL: https://ocaml.org/p/js_of_ocaml/5.0.1/doc/Js_of_ocaml/Regexp/index.html (visited on 05/10/2025).
- [4] Oleg Kiselyov. *Delimited Control in OCaml*. URL: <https://okmij.org/ftp/continuations/implementations.html>.
- [5] *Mercury Reference Manual: Clauses*. URL: https://mercurylang.org/information/doc-latest/mercury_ref/Clauses.html#Overview-of-Mercury-semantics (visited on 04/12/2025).
- [6] *OCaml Effects Examples*. URL: <https://github.com/ocaml-multicore/effects-examples/tree/master> (visited on 03/26/2025).
- [7] *The Curry Programming Language*. URL: <https://curry-lang.org/> (visited on 04/16/2025).
- [8] Max Carroll. *Polymorphic type slicing*. Tech. rep. UCAM-CL-TR-YYY. Originally a Part III dissertation, University of Cambridge. Hazel implementation branch: `type-slicing-v2`. University of Cambridge, Computer Laboratory, 2026.
- [9] *Bechamel Micro Benchmarking Library*. 2025. URL: <https://github.com/mirage/bechamel>.
- [10] Patrick Ferris. *A Corpus of annotated and unannotated ill-typed Hazel Programs*. May 2025. URL: <https://github.com/patricoferris/hazel-corpus>.
- [11] Patrick Ferris. *OCaml to Hazel transpiler*. May 2025. URL: https://github.com/patricoferris/hazel_of_ocaml.
- [12] TIOBE Software. *TIOBE Programming Community Index*. 2025. URL: <https://www.tiobe.com/tiobe-index/> (visited on 02/27/2025).
- [13] The OCaml Development Team. *The OCaml Manual: release 5.3 – Binding Operators*. 2025. URL: <https://ocaml.org/manual/5.3/bindingops.html> (visited on 05/12/2025).
- [14] Benjamin C Pierce. *Advanced topics in types and programming languages*. MIT press, 2024.
- [15] Eric Zhao et al. “Total Type Error Localization and Recovery with Holes”. In: *Proceedings of the ACM on Programming Languages* 8.POPL (Jan. 2024), pp. 2041–2068. ISSN: 2475-1421. DOI: [10.1145/3632910](https://doi.org/10.1145/3632910). URL: <http://dx.doi.org/10.1145/3632910>.
- [16] David S. Warren. “Introduction to Prolog”. In: *Prolog: The Next 50 Years*. Ed. by David S. Warren et al. Cham: Springer Nature Switzerland, 2023, pp. 3–19. ISBN: 978-3-031-35254-6. DOI: [10.1007/978-3-031-35254-6_1](https://doi.org/10.1007/978-3-031-35254-6_1). URL: https://doi.org/10.1007/978-3-031-35254-6_1.
- [17] Yongwei Yuan et al. “Live Pattern Matching with Typed Holes”. In: *Proc. ACM Program. Lang.* 7.OOPSLA1 (Apr. 2023). DOI: [10.1145/3586048](https://doi.org/10.1145/3586048). URL: <https://doi.org/10.1145/3586048>.
- [18] Abdulaziz Alaboudi and Thomas D. LaToza. *An Exploratory Study of Debugging Episodes*. 2021. arXiv: [2105.02162](https://arxiv.org/abs/2105.02162) [cs.SE]. URL: <https://arxiv.org/abs/2105.02162>.
- [19] Hannah Potter and Cyrus Omar. “Hazel tutor: Guiding novices through type-driven development strategies”. In: *Human Aspects of Types and Reasoning Assistants (HATRA)* (2020).

- [20] Zack Coker et al. “A Qualitative Study on Framework Debugging”. In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2019, pp. 568–579. DOI: [10.1109/ICSME.2019.00091](https://doi.org/10.1109/ICSME.2019.00091).
- [21] Oleg Kiselyov. “Effects Without Monads: Non-determinism – Back to the Meta Language”. In: *Electronic Proceedings in Theoretical Computer Science* 294 (May 2019), pp. 15–40. ISSN: 2075-2180. DOI: [10.4204/eptcs.294.2](https://doi.org/10.4204/eptcs.294.2). URL: <http://dx.doi.org/10.4204/EPTCS.294.2>.
- [22] Yusuke Miyazaki, Taro Sekiyama, and Atsushi Igarashi. “Dynamic type inference for gradual Hindley–Milner typing”. In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: [10.1145/3290331](https://doi.org/10.1145/3290331). URL: <https://doi.org/10.1145/3290331>.
- [23] Cyrus Omar et al. “Live functional programming with typed holes”. In: *Proceedings of the ACM on Programming Languages* 3.POPL (Jan. 2019), pp. 1–32. ISSN: 2475-1421. DOI: [10.1145/3290327](https://doi.org/10.1145/3290327). URL: <http://dx.doi.org/10.1145/3290327>.
- [24] Roberto Baldoni et al. “A Survey of Symbolic Execution Techniques”. In: *ACM Comput. Surv.* 51.3 (May 2018). ISSN: 0360-0300. DOI: [10.1145/3182657](https://doi.org/10.1145/3182657). URL: <https://doi.org/10.1145/3182657>.
- [25] Moritz Beller et al. “On the dichotomy of debugging behavior among programmers”. In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE ’18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 572–583. ISBN: 9781450356381. DOI: [10.1145/3180155.3180175](https://doi.org/10.1145/3180155.3180175). URL: <https://doi.org/10.1145/3180155.3180175>.
- [26] Eric L Seidel and Ranjit Jhala. *A Collection of Novice Interactions with the OCaml Top-Level System*. June 2017. DOI: [10.5281/zenodo.806814](https://doi.org/10.5281/zenodo.806814). URL: <https://doi.org/10.5281/zenodo.806814>.
- [27] Baijun Wu and Sheng Chen. “How type errors were fixed and what students did?” In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017). DOI: [10.1145/3133929](https://doi.org/10.1145/3133929). URL: <https://doi.org/10.1145/3133929>.
- [28] Matteo Cimini and Jeremy G. Siek. “The gradualizer: a methodology and algorithm for generating gradual type systems”. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’16. ACM, Jan. 2016, pp. 443–455. DOI: [10.1145/2837614.2837632](https://doi.org/10.1145/2837614.2837632). URL: <http://dx.doi.org/10.1145/2837614.2837632>.
- [29] Robert Harper. *Practical Foundations for Programming Languages: Second Edition*. Cambridge University Press, Mar. 2016. ISBN: 9781316576892. DOI: [10.1017/cbo9781316576892](https://doi.org/10.1017/cbo9781316576892). URL: <http://dx.doi.org/10.1017/CBO9781316576892>.
- [30] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. pearson, 2016.
- [31] Eric L. Seidel, Ranjit Jhala, and Westley Weimer. “Dynamic witnesses for static type errors (or, ill-typed programs usually go wrong)”. In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. ICFP’16. ACM, Sept. 2016, pp. 228–242. DOI: [10.1145/2951913.2951915](https://doi.org/10.1145/2951913.2951915). URL: <http://dx.doi.org/10.1145/2951913.2951915>.
- [32] Ronald Garcia and Matteo Cimini. “Principal Type Schemes for Gradual Programs”. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’15. Mumbai, India: Association for Computing Machinery, 2015, pp. 303–315. ISBN: 9781450333009. DOI: [10.1145/2676726.2676992](https://doi.org/10.1145/2676726.2676992). URL: <https://doi.org/10.1145/2676726.2676992>.

- [33] Jeremy G. Siek et al. “Refined Criteria for Gradual Typing”. In: *Summit on Advances in Programming Languages*. 2015. URL: <https://api.semanticscholar.org/CorpusID:15383644>.
- [34] Jana Dunfield and Neelakantan R. Krishnaswami. “Complete and easy bidirectional type-checking for higher-rank polymorphism”. In: *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*. ICFP’13. ACM, Sept. 2013. DOI: [10.1145/2500365.2500582](https://doi.org/10.1145/2500365.2500582). URL: <http://dx.doi.org/10.1145/2500365.2500582>.
- [35] Ohad Kammar, Sam Lindley, and Nicolas Oury. “Handlers in action”. In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP’13. Boston, Massachusetts, USA: Association for Computing Machinery, 2013, pp. 145–158. ISBN: 9781450323260. DOI: [10.1145/2500365.2500590](https://doi.org/10.1145/2500365.2500590). URL: <https://doi.org/10.1145/2500365.2500590>.
- [36] Lucas Layman et al. “Debugging Revisited: Toward Understanding the Debugging Needs of Contemporary Software Developers”. In: *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. 2013, pp. 383–392. DOI: [10.1109/ESEM.2013.43](https://doi.org/10.1109/ESEM.2013.43).
- [37] Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. *Real World OCaml: Functional programming for the masses*. O’Reilly Media, Inc., 2013.
- [38] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2012. ISBN: 0262017156.
- [39] Oleg Kiselyov. “Typed Tagless Final Interpreters”. In: *Generic and Indexed Programming: International Spring School, SSGIP 2010, Oxford, UK, March 22–26, 2010, Revised Lectures*. Ed. by Jeremy Gibbons. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 130–174. ISBN: 978-3-642-32202-0. DOI: [10.1007/978-3-642-32202-0_3](https://doi.org/10.1007/978-3-642-32202-0_3). URL: https://doi.org/10.1007/978-3-642-32202-0_3.
- [40] Johannes Oetsch et al. “On the small-scope hypothesis for testing answer-set programs”. In: *Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning*. KR’12. Rome, Italy: AAAI Press, 2012, pp. 43–53. ISBN: 9781577355601.
- [41] Roly Perera et al. “Functional programs that explain their work”. In: *ACM SIGPLAN Notices* 47.9 (Sept. 2012), pp. 365–376. ISSN: 1558-1160. DOI: [10.1145/2398856.2364579](https://doi.org/10.1145/2398856.2364579). URL: <http://dx.doi.org/10.1145/2398856.2364579>.
- [42] Bas Cornelissen, Andy Zaidman, and Arie van Deursen. “A Controlled Experiment for Program Comprehension through Trace Visualization”. In: *IEEE Transactions on Software Engineering* 37.3 (2011), pp. 341–355. DOI: [10.1109/TSE.2010.47](https://doi.org/10.1109/TSE.2010.47).
- [43] Sergio Antoy and Michael Hanus. “Functional logic programming”. In: *Commun. ACM* 53.4 (Apr. 2010), pp. 74–85. ISSN: 0001-0782. DOI: [10.1145/1721654.1721675](https://doi.org/10.1145/1721654.1721675). URL: <https://doi.org/10.1145/1721654.1721675>.
- [44] David Herman, Aaron Tomb, and Cormac Flanagan. “Space-efficient gradual typing”. In: *High.-order Symb. Comput.* 23.2 (June 2010), pp. 167–189.
- [45] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. “Contextual modal type theory”. In: *ACM Transactions on Computational Logic* 9.3 (June 2008), pp. 1–49. ISSN: 1557-945X. DOI: [10.1145/1352582.1352591](https://doi.org/10.1145/1352582.1352591). URL: <http://dx.doi.org/10.1145/1352582.1352591>.

- [46] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. “Smallcheck and lazy small-check: automatic exhaustive testing for small values”. In: *Proceedings of the First ACM SIGPLAN Symposium on Haskell*. Haskell '08. Victoria, BC, Canada: Association for Computing Machinery, 2008, pp. 37–48. ISBN: 9781605580647. DOI: [10.1145/1411286.1411292](https://doi.org/10.1145/1411286.1411292). URL: <https://doi.org/10.1145/1411286.1411292>.
- [47] LUC MARANGET. “Warnings for pattern matching”. In: *Journal of Functional Programming* 17.3 (2007), pp. 387–421. DOI: [10.1017/S0956796807006223](https://doi.org/10.1017/S0956796807006223).
- [48] Linda Dailey Paulson. “Developers shift to dynamic programming languages”. In: *Computer* 40.2 (2007), pp. 12–15. DOI: [10.1109/MC.2007.53](https://doi.org/10.1109/MC.2007.53).
- [49] Jeremy G. Siek and Walid Taha. “Gradual Typing for Functional Languages”. In: *Scheme and Functional Programming Workshop*. 2006, pp. 81–92.
- [50] Michael Spivey. “Algebras for Combinatorial Search”. In: July 2006. DOI: [10.14236/ewic/MSFP2006.11](https://doi.org/10.14236/ewic/MSFP2006.11).
- [51] Oleg Kiselyov et al. “Backtracking, interleaving, and terminating monad transformers: (functional pearl)”. In: *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*. ICFP '05. Tallinn, Estonia: Association for Computing Machinery, 2005, pp. 192–203. ISBN: 1595930647. DOI: [10.1145/1086365.1086390](https://doi.org/10.1145/1086365.1086390). URL: <https://doi.org/10.1145/1086365.1086390>.
- [52] Christian Haack and J.B. Wells. “Type error slicing in implicitly typed higher-order languages”. In: *Science of Computer Programming* 50.1–3 (Mar. 2004), pp. 189–224. ISSN: 0167-6423. DOI: [10.1016/j.scico.2004.01.004](https://doi.org/10.1016/j.scico.2004.01.004). URL: <http://dx.doi.org/10.1016/j.scico.2004.01.004>.
- [53] Alexandr Andoni et al. “Evaluating the ”Small Scope Hypothesis””. In: (Oct. 2002).
- [54] Benjamin C. Pierce. *Types and Programming Languages*. 1st. The MIT Press, 2002. ISBN: 0262162091.
- [55] F. Tip and T. B. Dinesh. “A slicing-based approach for locating type errors”. In: *ACM Transactions on Software Engineering and Methodology* 10.1 (Jan. 2001), pp. 5–55. ISSN: 1557-7392. DOI: [10.1145/366378.366379](https://doi.org/10.1145/366378.366379). URL: <http://dx.doi.org/10.1145/366378.366379>.
- [56] Sergio Antoy, Rachid Echahed, and Michael Hanus. “A needed narrowing strategy”. In: *J. ACM* 47.4 (July 2000), pp. 776–822. ISSN: 0004-5411. DOI: [10.1145/347476.347484](https://doi.org/10.1145/347476.347484). URL: <https://doi.org/10.1145/347476.347484>.
- [57] Koen Claessen and John Hughes. “QuickCheck: a lightweight tool for random testing of Haskell programs”. In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. ICFP '00. New York, NY, USA: Association for Computing Machinery, 2000, pp. 268–279. ISBN: 1581132026. DOI: [10.1145/351240.351266](https://doi.org/10.1145/351240.351266). URL: <https://doi.org/10.1145/351240.351266>.
- [58] Robert Harper and Christopher Stone. “A Type-Theoretic Interpretation of Standard ML”. In: *Proof, Language, and Interaction*. The MIT Press, May 2000, pp. 341–388. ISBN: 9780262281676. DOI: [10.7551/mitpress/5641.003.0019](https://doi.org/10.7551/mitpress/5641.003.0019). URL: <http://dx.doi.org/10.7551/mitpress/5641.003.0019>.
- [59] Benjamin C. Pierce and David N. Turner. “Local type inference”. In: *ACM Transactions on Programming Languages and Systems* 22.1 (Jan. 2000), pp. 1–44. ISSN: 1558-4593. DOI: [10.1145/345099.345100](https://doi.org/10.1145/345099.345100). URL: <http://dx.doi.org/10.1145/345099.345100>.

- [60] Michael Spivey. “Combinators for breadth-first search”. In: *J. Funct. Program.* 10.4 (July 2000), pp. 397–408. ISSN: 0956-7968. DOI: [10.1017/S0956796800003749](https://doi.org/10.1017/S0956796800003749). URL: <https://doi.org/10.1017/S0956796800003749>.
- [61] J.B. Wells. “Typability and type checking in System F are equivalent and undecidable”. In: *Annals of Pure and Applied Logic* 98.1 (1999), pp. 111–156. ISSN: 0168-0072. DOI: [10.1016/S0168-0072\(98\)00047-5](https://doi.org/10.1016/S0168-0072(98)00047-5). URL: <https://www.sciencedirect.com/science/article/pii/S0168007298000475>.
- [62] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, Apr. 1998. ISBN: 9780511530104. DOI: [10.1017/cbo9780511530104](https://doi.org/10.1017/cbo9780511530104). URL: <http://dx.doi.org/10.1017/CBO9780511530104>.
- [63] Benedict R Gaster and Mark P Jones. *A polymorphic type system for extensible records and variants*. Tech. rep. Technical Report NOTTCS-TR-96-3, Department of Computer Science, University of Nottingham, 1996.
- [64] Fritz Henglein. “Dynamic typing: syntax and proof theory”. In: *Sci. Comput. Program.* 22.3 (June 1994), pp. 197–230. ISSN: 0167-6423. DOI: [10.1016/0167-6423\(94\)00004-2](https://doi.org/10.1016/0167-6423(94)00004-2). URL: [https://doi.org/10.1016/0167-6423\(94\)00004-2](https://doi.org/10.1016/0167-6423(94)00004-2).
- [65] Robert Harper and John C. Mitchell. “On the type structure of standard ML”. In: *ACM Transactions on Programming Languages and Systems* 15.2 (Apr. 1993), pp. 211–252. ISSN: 1558-4593. DOI: [10.1145/169701.169696](https://doi.org/10.1145/169701.169696). URL: <http://dx.doi.org/10.1145/169701.169696>.
- [66] Benjamin C Pierce. *Basic category theory for computer scientists*. MIT press, 1991.
- [67] Olivier Danvy and Andrzej Filinski. “Abstracting control”. In: *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*. LFP ’90. Nice, France: Association for Computing Machinery, 1990, pp. 151–160. ISBN: 089791368X. DOI: [10.1145/91556.91622](https://doi.org/10.1145/91556.91622). URL: <https://doi.org/10.1145/91556.91622>.
- [68] M. Abadi et al. “Dynamic typing in a statically-typed language”. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’89. Austin, Texas, USA: Association for Computing Machinery, 1989, pp. 213–227. ISBN: 0897912942. DOI: [10.1145/75277.75296](https://doi.org/10.1145/75277.75296). URL: <https://doi.org/10.1145/75277.75296>.
- [69] Kevin Knight. “Unification: A multidisciplinary survey”. In: *ACM Computing Surveys (CSUR)* 21.1 (1989), pp. 93–124.
- [70] Luca Cardelli. “Structural subtyping and the notion of power type”. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1988, pp. 70–79.
- [71] Bogdan Korel and Janusz Laski. “Dynamic program slicing”. In: *Information Processing Letters* 29.3 (Oct. 1988), pp. 155–163. ISSN: 0020-0190. DOI: [10.1016/0020-0190\(88\)90054-3](https://doi.org/10.1016/0020-0190(88)90054-3). URL: [http://dx.doi.org/10.1016/0020-0190\(88\)90054-3](http://dx.doi.org/10.1016/0020-0190(88)90054-3).
- [72] Philip Wadler. “How to replace failure by a list of successes a method for exception handling, backtracking, and pattern matching in lazy functional languages”. In: *Functional Programming Languages and Computer Architecture*. Ed. by Jean-Pierre Jouanaud. Berlin, Heidelberg: Springer Berlin Heidelberg, 1985, pp. 113–128. ISBN: 978-3-540-39677-2.
- [73] Maurice Bruynooghe. *Adding redundancy to obtain more reliable and more readable prolog programs*. Tech. rep. Department of Computer Science, KU Leuven, Leuven, Belgium, 1982.

- [74] Luis Damas and Robin Milner. “Principal type-schemes for functional programs”. In: *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’82. Albuquerque, New Mexico: Association for Computing Machinery, 1982, pp. 207–212. ISBN: 0897910656.
DOI: [10.1145/582153.582176](https://doi.org/10.1145/582153.582176). URL: <https://doi.org/10.1145/582153.582176>.
- [75] Mark Weiser. “Program slicing”. In: *Proceedings of the 5th International Conference on Software Engineering*. ICSE ’81. San Diego, California, USA: IEEE Press, 1981, pp. 439–449. ISBN: 0897911466.
- [76] David HD Warren. “Applied logic: its use and implementation as a programming tool”. PhD thesis. The University of Edinburgh, 1978.
- [77] Robert W. Floyd. “Nondeterministic Algorithms”. In: *J. ACM* 14.4 (Oct. 1967), pp. 636–644. ISSN: 0004-5411.
DOI: [10.1145/321420.321422](https://doi.org/10.1145/321420.321422). URL: <https://doi.org/10.1145/321420.321422>.
- [78] Garrett Birkhoff. *Lattice theory*. Vol. 25. American Mathematical Soc., 1940.
- [79] Holbrook Mann MacNeille. “Partially ordered sets”. In: *Transactions of the American Mathematical Society* 42.3 (1937), pp. 416–460.

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:

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

$$\tau ::= \tau \rightarrow \tau \mid b$$

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

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

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

Judgements & Inference Rules

A *judgement*, J , is an assertion about *expressions* in a language [29]. For example:

- Exp $e - e$ is an *expression*
- $n : \text{int} - n$ has type *int*
- $e \Downarrow v - e$ evaluates to *value* v

While an *inference rule* is a collection of judgements $J, J_1, \dots, J_n$:

$$\frac{J_1 \quad J_2 \quad \dots \quad J_n}{J}$$

Representing the *rule* that if the *premises*, $J_1, \dots, J_n$ 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:

$$J_1, \dots, J_n \vdash J$$

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 from variables to types for variables, notated $x_1 : \tau_1, \dots, x_n : \tau_n$. For example the SLTC² [54, ch. 9] has a typing rule for lambda expression and application as follows:

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

Meaning, $\lambda x. e$ has type $\tau_1 \rightarrow \tau_2$ 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 $\tau_1 \rightarrow \tau_2$ and its argument e_2 has type τ_1 .

Small Step Operational Semantics

A relation $e_1 \rightarrow e_2$ can be defined via judgement rules to determine the evaluation semantics of the language. It's multi-step (transitive closure) analogue is notated $e_1 \rightarrow^* e_2$, meaning $e_1 = e_2$ or there exists a sequence of steps $e_1 \rightarrow e'_1 \rightarrow \dots \rightarrow e_2$.

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

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 :

$$\frac{}{(\lambda x : \tau. e)v \rightarrow [v/x]e} \quad \frac{e_2 \rightarrow e'_2}{e_1(e_2) \rightarrow e_1(e'_2)}$$

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

²Simply typed lambda calculus.

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*. Note that the core calculus, and hence the slicing theory built upon it, contains no case expressions, sum types, or product types; the implementation supports these features, but they are not treated formally.

B.1. Syntax

$$\begin{aligned}\tau &::= b \mid \tau \rightarrow \tau \mid ? \\ e &::= c \mid x \mid \lambda x : \tau. e \mid \lambda x. e \mid e(e) \mid \llbracket \cdot \rrbracket^u \mid \llbracket e \rrbracket^u \mid e : \tau \\ d &::= c \mid x \mid \lambda x : \tau d \mid d(d) \mid \llbracket \cdot \rrbracket_\sigma^u \mid \llbracket d \rrbracket_\sigma^u \mid d \langle \tau \Rightarrow \tau \rangle \mid d \langle \tau \Rightarrow ? \not\Rightarrow \tau \rangle\end{aligned}$$

Figure B.1: Syntax: *types* τ , *external expressions* e , *internal expressions* d . With x ranging over variables, u over hole names, σ over $x \rightarrow d$ *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

$$\begin{aligned}\text{SConst} &\frac{}{\Gamma \vdash c \Rightarrow b} & \text{SVar} &\frac{x : \tau \in \Gamma}{\Gamma \vdash x \Rightarrow \tau} & \text{SFun} &\frac{\Gamma, x : \tau_1 \vdash e \Rightarrow \tau_2}{\Gamma \vdash \lambda x : \tau_1. e \Rightarrow \tau_1 \rightarrow \tau_2} \\ \text{SApp} &\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \quad \tau_1 \blacktriangleright \rightarrow \tau_2 \rightarrow \tau \quad \Gamma \vdash e_2 \Leftarrow \tau_2}{\Gamma \vdash e_1(e_2) \Rightarrow \tau} & \text{SEHole} &\frac{}{\Gamma \vdash \llbracket \cdot \rrbracket^u \Rightarrow ?} \\ \text{SNEHole} &\frac{\Gamma \vdash e \Rightarrow \tau}{\Gamma \vdash \llbracket e \rrbracket^u \Rightarrow ?} & \text{SAsc} &\frac{\Gamma \vdash e \Leftarrow \tau}{\Gamma \vdash e : \tau \Rightarrow \tau}\end{aligned}$$

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

$$\begin{aligned}\text{AFun} &\frac{\tau \blacktriangleright \rightarrow \tau_1 \rightarrow \tau_2 \quad \Gamma, x : \tau_1 \vdash e \Leftarrow \tau_2}{\Gamma \vdash \lambda x. e \Leftarrow \tau} & \text{ASubsume} &\frac{\Gamma \vdash e \Rightarrow \tau \quad \tau \sim \tau'}{\Gamma \vdash e \Leftarrow \tau'}\end{aligned}$$

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

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

$$\begin{aligned}\text{TCDyn1} &\frac{}{? \sim \tau} & \text{TCDyn2} &\frac{}{\tau \sim ?} & \text{TCRfl} &\frac{}{\tau \sim \tau} & \text{TCFun} &\frac{\tau_1 \sim \tau'_1 \quad \tau_2 \sim \tau'_2}{\tau_1 \rightarrow \tau_2 \sim \tau'_1 \rightarrow \tau'_2}\end{aligned}$$

Figure B.3: Type consistency

$\tau \blacktriangleright \rightarrow \tau_1 \rightarrow \tau_2$ τ has arrow type $\tau_1 \rightarrow \tau_2$

$$\text{MADyn} \frac{? \blacktriangleright \rightarrow ? \rightarrow ?}{\tau_1 \rightarrow \tau_2 \blacktriangleright \rightarrow \tau_1 \rightarrow \tau_2} \quad \text{MAFun} \frac{}{\tau_1 \rightarrow \tau_2 \blacktriangleright \rightarrow \tau_1 \rightarrow \tau_2}$$

Figure B.4: Type Matching

B.2.2. Elaboration

$\Gamma \vdash e \Rightarrow \tau \rightsquigarrow d \dashv \Delta$ e synthesises type τ and elaborates to d

$$\begin{array}{c} \text{ESConst} \frac{}{\Gamma \vdash c \Rightarrow b \rightsquigarrow c \dashv \emptyset} \quad \text{ESVar} \frac{x : \tau \in \Gamma}{\Gamma \vdash x \Rightarrow \tau \rightsquigarrow x \dashv \emptyset} \\[10pt] \text{ESFun} \frac{\Gamma, x : \tau_1 \vdash e \Rightarrow \tau_2 \rightsquigarrow d \dashv \Delta}{\Gamma \vdash \lambda x : \tau_1. e \Rightarrow \tau_1 \rightarrow \tau_2 \rightsquigarrow \lambda x : \tau_1. d \dashv \Delta} \\[10pt] \text{ESApp} \frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \quad \tau_1 \blacktriangleright \rightarrow \tau_2 \rightarrow \tau \quad \Gamma \vdash e_2 \Leftarrow \tau_2 \rightsquigarrow d_2 : \tau'_2 \dashv \Delta_2}{\Gamma \vdash e_1(e_2) \Rightarrow \tau \rightsquigarrow (d_1 \langle \tau'_1 \Rightarrow \tau_2 \rightarrow \tau \rangle) (d_2 \langle \tau'_2 \Rightarrow \tau_2 \rangle) \dashv \Delta_1 \cup \Delta_2} \\[10pt] \text{ESEHole} \frac{}{\Gamma \vdash \llbracket \cdot \rrbracket^u \Rightarrow ? \rightsquigarrow \llbracket \cdot \rrbracket_{\text{id}(\Gamma)}^u \dashv u :: \llbracket \cdot \rrbracket[\Gamma]} \\[10pt] \text{ESNEHole} \frac{\Gamma \vdash e \Rightarrow \tau \rightsquigarrow d \dashv \Delta}{\Gamma \vdash \llbracket e \rrbracket^u \Rightarrow ? \rightsquigarrow \llbracket d \rrbracket_{\text{id}(\Gamma)}^u \dashv \Delta, u :: \llbracket \cdot \rrbracket[\Gamma]} \\[10pt] \text{ESAsc} \frac{\Gamma \vdash e \Leftarrow \tau \rightsquigarrow d : \tau' \dashv \Delta}{\Gamma \vdash e : \tau \Rightarrow \tau \rightsquigarrow d \langle \tau' \Rightarrow \tau \rangle \dashv \Delta} \end{array}$$

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

$$\begin{array}{c} \text{EAFun} \frac{\tau \blacktriangleright \rightarrow \tau_1 \rightarrow \tau_2 \quad \Gamma, x : \tau_1 \vdash e \Leftarrow \tau_2 \rightsquigarrow d : \tau'_2 \dashv \Delta}{\Gamma \vdash \lambda x. e \Leftarrow \tau \rightsquigarrow \lambda x : \tau_1. d : \tau_1 \rightarrow \tau'_2 \dashv \Delta} \\[10pt] \text{EASubsume} \frac{e \neq \llbracket \cdot \rrbracket^u \quad \Gamma \vdash e \Rightarrow \tau' \rightsquigarrow d \dashv \Delta \quad \tau \sim \tau'}{\Gamma \vdash e \Leftarrow \tau \rightsquigarrow d : \tau' \dashv \Delta} \\[10pt] \text{EAEHole} \frac{}{\Gamma \vdash \llbracket \cdot \rrbracket^u \Leftarrow \tau \rightsquigarrow \llbracket \cdot \rrbracket_{\text{id}(\Gamma)}^u : \tau \dashv u :: \tau[\Gamma]} \\[10pt] \text{EANEHole} \frac{\Gamma \vdash e \Rightarrow \tau' \rightsquigarrow d \dashv \Delta}{\Gamma \vdash \llbracket e \rrbracket^u \Leftarrow \tau \rightsquigarrow \llbracket d \rrbracket_{\text{id}(\Gamma)}^u : \tau \dashv u :: \tau[\Gamma]} \end{array}$$

Figure B.5: Elaboration judgements

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

$\Delta; \Gamma \vdash \sigma : \Gamma'$ iff $\text{dom}(\sigma) = \text{dom}(\Gamma')$ 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

$$\begin{array}{c}
\text{TACons} \frac{}{\Delta; \Gamma \vdash c : b} \quad \text{TAVar} \frac{x : \tau \in \Gamma}{\Delta; \Gamma \vdash x : \tau} \quad \text{TAFun} \frac{\Delta; \Gamma, x : \tau_1 \vdash d : \tau_2}{\Delta; \Gamma \vdash \lambda x : \tau_1. d : \tau_1 \rightarrow \tau_2} \\
\\
\text{TAAp} \frac{\Delta; \Gamma \vdash d_1 : \tau_2 \rightarrow \tau \quad \Delta; \Gamma \vdash d_2 : \tau_2}{\Delta; \Gamma \vdash d_1(d_2) : \tau} \quad \text{TAEHole} \frac{u :: \tau[\Gamma'] \in \Delta \quad \Delta; \Gamma \vdash \sigma : \Gamma'}{\Delta; \Gamma \vdash \llbracket \sigma \rrbracket^u : \tau} \\
\\
\text{TANEHole} \frac{\Delta; \Gamma \vdash d : \tau' \quad u :: \tau[\Gamma'] \in \Delta \quad \Delta; \Gamma \vdash \sigma : \Gamma'}{\Delta; \Gamma \vdash \llbracket d \rrbracket^u_{\sigma} : \tau} \quad \text{TACast} \frac{\Delta; \Gamma \vdash d : \tau_1 \quad \tau_1 \sim \tau_2}{\Delta; \Gamma \vdash d \langle \tau_1 \Rightarrow \tau_2 \rangle : \tau_2} \\
\\
\text{TACastError} \frac{\Delta; \Gamma \vdash d : \tau_1 \quad \tau_1 \text{ ground} \quad \tau_2 \text{ ground} \quad \tau_1 \neq \tau_2}{\Delta; \Gamma \vdash d \langle \tau_1 \Rightarrow ? \Rightarrow \tau_2 \rangle : \tau_2}
\end{array}$$

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

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

$$\text{GBase} \frac{}{b \text{ ground}} \quad \text{GDynFun} \frac{}{? \rightarrow ? \text{ ground}}$$

Figure B.8: Ground types

B.3. Dynamics

B.3.1. Final Forms

$$\begin{array}{c}
\boxed{d \text{ final}} \quad d \text{ is final} \\
\text{FBoxedVal} \frac{d \text{ boxedval}}{d \text{ final}} \quad \text{FIndex} \frac{d \text{ indet}}{d \text{ final}} \\
\boxed{d \text{ val}} \quad d \text{ is a value} \\
\text{VConst} \frac{}{c \text{ val}} \quad \text{VFun} \frac{}{\lambda x : \tau. d \text{ val}} \\
\boxed{d \text{ boxedval}} \quad d \text{ is a boxed value} \\
\text{BVVal} \frac{d \text{ val}}{d \text{ boxedval}} \quad \text{BVFunCast} \frac{\tau \rightarrow \tau_2 \neq \tau_3 \rightarrow \tau_4 \quad d \text{ boxedval}}{d \langle \tau_1 \rightarrow \tau_2 \Rightarrow \tau_3 \rightarrow \tau_4 \rangle \text{ boxedval}} \\
\text{BVDynCast} \frac{d \text{ boxedval} \quad \tau \text{ ground}}{d \langle \tau \Rightarrow ? \rangle \text{ boxedval}} \\
\boxed{d \text{ indet}} \quad d \text{ is indeterminate} \\
\text{IEHole} \frac{}{\llbracket \cdot \rrbracket_\sigma^u \text{ indet}} \quad \text{INEHole} \frac{d \text{ final}}{\llbracket d \rrbracket_\sigma^u \text{ indet}} \quad \text{IAp} \frac{d_1 \neq d'_1 \langle \tau_1 \rightarrow \tau_2 \Rightarrow \tau_3 \rightarrow \tau_4 \rangle \quad d_1 \text{ indet} \quad d_2 \text{ final}}{d_1(d_2) \text{ indet}} \\
\text{ICastGD} \frac{d \text{ indet} \quad \tau \text{ ground}}{d \langle \tau \Rightarrow ? \rangle \text{ indet}} \quad \text{ICastDG} \frac{d \neq d' \langle \tau' \Rightarrow ? \rangle \quad d \text{ indet} \quad \tau \text{ ground}}{d \langle ? \Rightarrow \tau \rangle \text{ indet}} \\
\text{ICastFun} \frac{\tau_1 \rightarrow \tau_2 \neq \tau_3 \rightarrow \tau_4 \quad d \text{ indet}}{d \langle \tau_1 \rightarrow \tau_2 \Rightarrow \tau_3 \rightarrow \tau_4 \rangle \text{ indet}} \quad \text{ICastError} \frac{d \text{ final} \quad \tau_1 \text{ ground} \quad \tau_2 \text{ ground} \quad \tau_1 \neq \tau_2}{d \langle \tau_1 \Rightarrow ? \Rightarrow \tau_2 \rangle \text{ indet}}
\end{array}$$

Figure B.9: Final forms

B.3.2. Instructions

$$\begin{array}{c}
\boxed{d \longrightarrow d'} \quad d \text{ takes an instruction transition to } d' \\
\text{ITFun} \frac{}{(\lambda x : \tau. d_1)(d_2) \longrightarrow [d_2/x]d_1} \quad \text{ITCastId} \frac{}{d \langle \tau \Rightarrow \tau \rangle \longrightarrow d} \\
\text{ITAppCast} \frac{\tau_1 \rightarrow \tau_2 \neq \tau'_1 \rightarrow \tau'_2}{d_1 \langle \tau_1 \rightarrow \tau_2 \Rightarrow \tau'_1 \rightarrow \tau'_2 \rangle (d) \longrightarrow (d_1(d_2 \langle \tau'_1 \Rightarrow \tau_1 \rangle)) \langle \tau_2 \Rightarrow \tau'_2 \rangle} \\
\text{ITCast} \frac{\tau \text{ ground}}{d \langle \tau \Rightarrow ? \Rightarrow \tau \rangle \longrightarrow d} \quad \text{ITCastError} \frac{\tau_1 \neq \tau_2 \quad \tau_1 \text{ ground} \quad \tau_2 \text{ ground}}{d \langle \tau_1 \Rightarrow ? \Rightarrow \tau_2 \rangle \longrightarrow d \langle \tau_1 \Rightarrow ? \Rightarrow \tau_2 \rangle} \\
\text{ITGround} \frac{\tau \blacktriangleright_{\text{ground}} \tau'}{d \langle \tau \Rightarrow ? \rangle \longrightarrow d \langle \tau \Rightarrow \tau' \Rightarrow ? \rangle} \quad \text{ITExpand} \frac{\tau \blacktriangleright_{\text{ground}} \tau'}{d \langle ? \Rightarrow \tau \rangle \longrightarrow d \langle ? \Rightarrow \tau' \Rightarrow \tau \rangle}
\end{array}$$

Figure B.10: Instruction transitions (non-deterministic version)

B.3.3. Contextual Dynamics

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

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

Figure B.11: Ground type matching

Context syntax:

$$E ::= \circ \mid E(d) \mid d(E) \mid \langle E \rangle_{\sigma}^u \mid E \langle \tau \Rightarrow \tau \rangle \mid E \langle \tau \Rightarrow ? \not\Rightarrow \tau \rangle$$

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

$$\begin{array}{c} \text{EOuter} \frac{}{d = \circ[d]} \quad \text{EApp1} \frac{d_1 = E[d'_1]}{d_1(d_2) = E(d_2)[d'_1]} \quad \text{EApp2} \frac{d_2 = E[d'_2]}{d_1(d_2) = d_1(E)[d'_2]} \\ \\ \text{ECNEHole} \frac{d = E[d']}{\langle d \rangle_{\sigma}^u = \langle E \rangle_{\sigma}^u[d']} \quad \text{ECCast} \frac{d = E[d']}{d \langle \tau_1 \Rightarrow \tau_2 \rangle = E \langle \tau_1 \Rightarrow \tau_2 \rangle[d']} \\ \\ \text{ECCastError} \frac{d = E[d']}{d \langle \tau_1 \Rightarrow ? \not\Rightarrow \tau_2 \rangle = E \langle \tau_1 \Rightarrow ? \not\Rightarrow \tau_2 \rangle[d']} \end{array}$$

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

$$\text{Step} \frac{d_1 = E[d_2] \quad d_2 \longrightarrow d'_2 \quad d'_1 = E[d'_2]}{d_1 \mapsto d'_1}$$

Figure B.12: Contextual dynamics of the internal language (non-determinism version)

B.3.4. Hole Substitution

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

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

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

$$\begin{aligned}
\llbracket d/u \rrbracket \cdot & = \cdot \\
\llbracket d/u \rrbracket \sigma, d'/x & = \llbracket d/u \rrbracket \sigma, (\llbracket d/u \rrbracket d')/x
\end{aligned}$$

Figure B.13: Hole substitution

Slicing Theory

None of the proofs in this appendix are machine-checked.

C.1. Expression Typing Slices

C.1.1. Term Slices

Syntax

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

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

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

Type slices v :

$$v ::= \square_{typ} \mid ? \mid b \mid v \rightarrow v$$

Expression slices ς :

$$\varsigma ::= \square_{exp} \mid c \mid x \mid \lambda p : v. \varsigma \mid \lambda p. \varsigma \mid \varsigma(\varsigma) \mid \langle \rangle^u \mid \langle \varsigma \rangle^u \mid \varsigma : v$$

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

Precision Relation

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

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

Type slices v :

$$\frac{}{\square_{typ} \sqsubseteq v} \quad \frac{}{v \sqsubseteq v} \quad \frac{v'_1 \sqsubseteq v_1 \quad v'_2 \sqsubseteq v_2}{v'_1 \rightarrow v'_2 \sqsubseteq v_1 \rightarrow v_2}$$

Expression slices ς :

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

$$\frac{p' \sqsubseteq p \quad \varsigma' \sqsubseteq \varsigma}{\lambda p'. \varsigma' \sqsubseteq \lambda p. \varsigma} \quad \frac{\varsigma'_1 \sqsubseteq \varsigma_1 \quad \varsigma'_2 \sqsubseteq \varsigma_2}{\varsigma'_1(\varsigma'_2) \sqsubseteq \varsigma_1(\varsigma_2)} \quad \frac{\varsigma' \sqsubseteq \varsigma \quad v' \sqsubseteq v}{\varsigma' : v' \sqsubseteq \varsigma : v}$$

Proposition 6 (Precision is a partial order). *Term precision forms a partial order on term slices. That is:*

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

Proof. Rule induction on the derivations of $\sqsubseteq$. Any derivation is the gap axiom, reflexivity, or a congruence rule; reflexivity can always be unfolded into congruences with reflexivity at the leaves, so assume derivations are structural.

Reflexivity is an axiom.

Antisymmetry: if either derivation is the gap axiom, say $\varsigma_1 = \square$, then $\varsigma_2 \sqsubseteq \square$ forces $\varsigma_2 = \square$, since the only rules concluding precision below $\square$ are the gap axiom and reflexivity, both requiring $\varsigma_2 = \square$. Otherwise both derivations end in the congruence rule for a common constructor; representative case application: $\varsigma_1 = \varsigma_a(\varsigma_b)$, $\varsigma_2 = \varsigma_c(\varsigma_d)$ with $\varsigma_a \sqsubseteq \varsigma_c \sqsubseteq \varsigma_a$ and $\varsigma_b \sqsubseteq \varsigma_d \sqsubseteq \varsigma_b$, so the inductive hypothesis gives equality componentwise.

Transitivity: induction on the pair of derivations. If the first is the gap axiom then $\varsigma_1 = \square \sqsubseteq \varsigma_3$ by the gap axiom. Otherwise both are congruences for a common constructor and the inductive hypotheses compose componentwise. $\square$

Lattice Structure

Definition 13 (Term Slice Joins). *For all t :*

$$\square \sqcup t = t = t \sqcup \square \quad t \sqcup t = t$$

$$(v_1 \rightarrow v_2) \sqcup (v'_1 \rightarrow v'_2) = (v_1 \sqcup v'_1) \rightarrow (v_2 \sqcup v'_2)$$

$$(\lambda p : v. \varsigma) \sqcup (\lambda p' : v'. \varsigma') = \lambda(p \sqcup p') : (v \sqcup v'). (\varsigma \sqcup \varsigma') \quad (\lambda p. \varsigma) \sqcup (\lambda p'. \varsigma') = \lambda(p \sqcup p'). (\varsigma \sqcup \varsigma')$$

$$\varsigma_1(\varsigma_2) \sqcup \varsigma'_1(\varsigma'_2) = (\varsigma_1 \sqcup \varsigma'_1)(\varsigma_2 \sqcup \varsigma'_2) \quad (\varsigma : v) \sqcup (\varsigma' : v') = (\varsigma \sqcup \varsigma') : (v \sqcup v')$$

Definition 14 (Term Slice Meets). *For all t, t' :*

$$\square \sqcap t = \square = t \sqcap \square \quad t \sqcap t = t$$

$$(v_1 \rightarrow v_2) \sqcap (v'_1 \rightarrow v'_2) = (v_1 \sqcap v'_1) \rightarrow (v_2 \sqcap v'_2)$$

$$(\lambda p : v. \varsigma) \sqcap (\lambda p' : v'. \varsigma') = \lambda(p \sqcap p') : (v \sqcap v'). (\varsigma \sqcap \varsigma') \quad (\lambda p. \varsigma) \sqcap (\lambda p'. \varsigma') = \lambda(p \sqcap p'). (\varsigma \sqcap \varsigma')$$

$$\varsigma_1(\varsigma_2) \sqcap \varsigma'_1(\varsigma'_2) = (\varsigma_1 \sqcap \varsigma'_1)(\varsigma_2 \sqcap \varsigma'_2) \quad (\varsigma : v) \sqcap (\varsigma' : v') = (\varsigma \sqcap \varsigma') : (v \sqcap v')$$

Otherwise:

$$t \sqcap t' = \square$$

Proposition 7 (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 :

$$\varsigma_1 \sqsubseteq \varsigma_1 \sqcup \varsigma_2 \quad \varsigma_2 \sqsubseteq \varsigma_1 \sqcup \varsigma_2$$

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 :

$$\varsigma_1 \sqcap \varsigma_2 \sqsubseteq \varsigma_1 \quad \varsigma_1 \sqcap \varsigma_2 \sqsubseteq \varsigma_2$$

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:

$$\varsigma \sqsubseteq \varsigma_1 \sqcap \varsigma_2$$

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

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

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. First note that two slices of the same complete term t agree structurally wherever neither is a gap: by induction on t , at each position a slice of t is either a gap or carries the same constructor as t . Hence the join and meet equations are total on slices of a common t : the mismatched-constructor case $t \sqcap t' = \perp$ is never reached, and joins always exist, unlike for slices of differing terms.

Upper bound: induction on t . If either slice is $\perp$ the join is the other slice, an upper bound of both by the gap axiom and reflexivity. Otherwise, representative case $t = t_1(t_2)$: $\varsigma_1 = \varsigma_a(\varsigma_b)$ and $\varsigma_2 = \varsigma_c(\varsigma_d)$, and the congruence rule applies to the inductive hypotheses componentwise.

Least upper bound: let $\varsigma \sqsubseteq t$ bound ς_1, ς_2 from above; induction on t . If both slices are gaps then $\varsigma_1 \sqcup \varsigma_2 = \perp \sqsubseteq \varsigma$. If exactly one is a gap the join equals the other slice, which ς bounds by assumption. Otherwise neither is a gap, so ς cannot be a gap either (it bounds a non-gap slice from above), all three carry t 's constructor, and the inductive hypothesis applies componentwise.

Meets and the greatest lower bound property are dual, with the extra base case that $\perp$ is a lower bound of everything. The absorption and idempotent laws follow from joins and meets being least upper and greatest lower bounds in a partial order. Boundedness is the gap axiom ($\perp \sqsubseteq \varsigma$) together with $\varsigma \sqsubseteq t$, which holds by definition of ς being a slice of t . $\square$

C.1.2. Typing Assumption Slices

Typing Assumptions as Partial Functions

Definition 15 (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 $\Gamma(x) = \perp$. It's domain $\text{dom}(\Gamma)$ is largest the set of variables $S \subseteq \mathcal{X}$ for which Γ is defined: $\forall x \in S. \Gamma(x) \neq \perp$.*

Typing Assumption Slices

Definition 16 (Typing Assumption Slices). *A typing assumption slice γ is a partial function from the set of variables $\mathcal{X}$ to type slices v .*

Precision

Definition 17 (Typing Assumption Slice Precision). *For typing assumption slices γ_1, γ_2 . Where $\text{dom}(f)$ is the set of variables for which a partial function f is defined:*

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

Proposition 8 (Precision is a partial order). *Typing assumption precision forms a partial order on typing assumption slices.*

Proof. Reflexivity and transitivity are inherited pointwise from $\subseteq$ on domains and $\sqsubseteq$ on type slices. Antisymmetry: mutual inclusion gives $\text{dom}(\gamma_1) = \text{dom}(\gamma_2)$, and antisymmetry of type slice precision then applies at each variable of the common domain, so γ_1 and γ_2 agree wherever defined. $\square$

Lattice Structure

Definition 18 (Typing Assumption Slice Joins and Meets). *For typing slices γ_1, γ_2 , and any variable x :*

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

Proposition 9 (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 partial function $\emptyset$ and top element Γ .*

Proof. The domain of $\gamma_1 \sqcup \gamma_2$ is $\text{dom}(\gamma_1) \cup \text{dom}(\gamma_2)$, and of $\gamma_1 \sqcap \gamma_2$ is $\text{dom}(\gamma_1) \cap \text{dom}(\gamma_2)$; on the intersection both values are slices of the common $\Gamma(x)$, so their joins and meets exist by proposition 7. The bound and universal properties are checked per variable x . Where both slices are defined they follow from proposition 7 applied to slices of $\Gamma(x)$. Where exactly one is defined, the join takes its value, and any upper bound must be defined there with a value bounding it (by the domain inclusion in the definition of precision). Where neither is defined nothing is required. The bottom and top elements are immediate: $\emptyset$ has empty domain, and every slice of Γ is below Γ by definition. $\square$

C.1.3. Expression Typing Slices

Definition 19 (Expression Typing Slices). *An expression typing slice ρ is a pair ς^γ of expression slice ς and typing assumption slice γ .*

Precision

Definition 20 (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 10 (Precision is a Partial Order). *Expression typing precision forms a partial order on expression typing slices.*

Proof. This is the product order on pairs ς^γ : each law is checked in each component separately using the corresponding law for term slices and typing assumption slices. Representative case antisymmetry: $\varsigma_1^{\gamma_1} \sqsubseteq \varsigma_2^{\gamma_2} \sqsubseteq \varsigma_1^{\gamma_1}$ gives $\varsigma_1 = \varsigma_2$ and $\gamma_1 = \gamma_2$ componentwise, so the pairs are equal. $\square$

Lattice Structure

Definition 21 (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 11 (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 .*

Proof. The product of two bounded lattices is a bounded lattice, with joins, meets, and bounds taken componentwise. Representative case, the least upper bound property: an upper bound ς^γ of $\varsigma_1^{\gamma_1}$ and $\varsigma_2^{\gamma_2}$ bounds each component, so $\varsigma_1 \sqcup \varsigma_2 \sqsubseteq \varsigma$ and $\gamma_1 \sqcup \gamma_2 \sqsubseteq \gamma$ by the component lattices, which is exactly $(\varsigma_1 \sqcup \varsigma_2)^{\gamma_1 \sqcup \gamma_2} \sqsubseteq \varsigma^\gamma$. $\square$

Type Checking

Definition 22 (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:*

$$\frac{}{\llbracket \square_{typ} \rrbracket = ?} \qquad \frac{}{\llbracket \square_{pat} \rrbracket = -} \qquad \frac{u \text{ is fresh}}{\llbracket \square_{exp} \rrbracket = \langle \rangle^u}$$

$$\overline{[-]} = - \qquad \overline{[x]} = x$$
$$\frac{}{[\tau] = \tau} \qquad \frac{[v_1] = \tau_1 \quad [v_2] = \tau_2}{[v_1 \rightarrow v_2] = \tau_1 \rightarrow \tau_2}$$
$$\frac{}{\llbracket e \rrbracket = e} \quad \frac{\frac{\llbracket p \rrbracket = pt \quad \llbracket v \rrbracket = \tau \quad \llbracket \varsigma \rrbracket = e}{\llbracket \lambda p : v. \varsigma \rrbracket = \lambda pt : \tau. e}}{\llbracket p \rrbracket = pt \quad \llbracket \varsigma \rrbracket = e} \quad \frac{\frac{\llbracket \varsigma_1 \rrbracket = e_1 \quad \llbracket \varsigma_2 \rrbracket = e_2}{\llbracket \varsigma_1(\varsigma_2) \rrbracket = e_1(e_2)}}{\llbracket \varsigma \rrbracket = e \quad \llbracket v \rrbracket = \tau} \quad \frac{}{\llbracket \lambda p. \varsigma \rrbracket = \lambda pt. e} \quad \frac{}{\llbracket \varsigma : v \rrbracket = e : \tau}$$
$$\begin{aligned} \llbracket \gamma \rrbracket(x) &= \llbracket \gamma(x) \rrbracket && \text{if } \gamma(x) \neq \perp \\ \llbracket \gamma \rrbracket(x) &= \perp && \text{if } \gamma(x) = \perp \end{aligned}$$
$$\mathcal{P} ::= \bigcirc_{pat}$$
$$\mathcal{J} ::= \textcolor{blue}{O}_{typ} \mid \mathcal{J} \rightarrow \tau \mid \tau \rightarrow \mathcal{J}$$
$$\mathcal{C} ::= \bigcirc_{exp} \mid \lambda \mathcal{P} : \tau. e \mid \lambda_{pt} : \mathcal{T}. e \mid \lambda_{pt} : \tau. \mathcal{C} \mid \lambda \mathcal{P}. e \mid \lambda_{pt}. \mathcal{C} \mid \mathcal{C}(e) \mid e(\mathcal{C}) \mid e : \mathcal{T} \mid \mathcal{C} : \tau$$
$$\begin{array}{c}
\frac{\circlearrowleft_{pat}\{pt\} = pt}{\mathcal{T}\{\tau_1\} = \tau'_1} \\
\frac{(\mathcal{T} \rightarrow \tau_2)\{\tau_1\} = \tau'_1 \rightarrow \tau_2}{\mathcal{P}\{pt\} = pt'} \\
\frac{(\lambda\mathcal{P} : \tau. e)\{pt\} = \lambda pt' : \tau. e}{\mathcal{T}\{\tau\} = \tau'} \\
\frac{(\lambda pt : \mathcal{T}. e)\{\tau\} = \lambda pt : \tau'. e}{e = e'}
\end{array}
\quad
\begin{array}{c}
\frac{\circlearrowleft_{typ}\{\tau\} = \tau}{\mathcal{T}\{\tau_2\} = \tau'_2} \\
\frac{(\tau_1 \rightarrow \mathcal{T})\{\tau_2\} = \tau_1 \rightarrow \tau'_2}{\mathcal{P}\{pt\} = pt'} \\
\frac{(\lambda\mathcal{P}. e)\{pt\} = \lambda pt'. e}{\mathcal{T}\{\tau\} = \tau'} \\
\frac{(e : \mathcal{T})\{\tau\} = e : \tau'}{C\{e\} = e'} \\
\frac{C\{e\} = e'}{(\lambda pt. C)\{\tau\} = \lambda pt. e'} \\
\frac{C\{e_1\} = e'_1}{(C(e_2))\{e_1\} = e'_1(e_2)}
\end{array}$$

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

Definition 27 (Context Composition). *Defined analogously as context substitution, but substituting contexts syntactically, provided the input and output classes match. If $C_1 : \mathbf{X} \rightarrow \mathbf{Y}$ and $C_2 : \mathbf{Y} \rightarrow \mathbf{Z}$ then $C_2\{C_1\} = C_2 \circ C_1 : \mathbf{X} \rightarrow \mathbf{Z}$. Equivalence of contexts can be defined syntactically and coincides exactly with an extensional definition.*

Composition could alternatively be defined pointwise, and we get the same result here, but it is later important we keep this syntactic to allow decomposition around the marks inside contexts.

Proposition 12 (Context composition is associative). *For all $C_1 : \mathbf{X} \rightarrow \mathbf{Y}$, $C_2 : \mathbf{Y} \rightarrow \mathbf{Z}$, and $C_3 : \mathbf{Z} \rightarrow \mathbf{W}$ then:*

$$(C_3 \circ C_2) \circ C_1 = C_3 \circ (C_2 \circ C_1)$$

Proof. Induction on the structure of the outer context C_3 . Base case $C_3 = \bigcirc$: both sides are $C_2 \circ C_1$ by the mark substitution rule. Representative inductive case $C_3 = C'_3(e)$: both compositions substitute inside C'_3 , leaving e untouched, so the claim follows from the inductive hypothesis for C'_3 . All other cases are identical: composition only rewrites along the unique path from the root to the mark. $\square$

C.2.2. Context Slices

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

Precision

Definition 28 (Context Precision). *If $c : \mathbf{X} \rightarrow \mathbf{Y}$ and $c' : \mathbf{X} \rightarrow \mathbf{Y}$ are context slices, then $c' \sqsubseteq c$ if and only if, for all terms t of class $\mathbf{X}$, that $c'\{t\} \sqsubseteq c\{t\}$.*

Proposition 13 (Precision is a Partial Order). *Context precision forms a partial order on context slices.*

Proof. Reflexivity and transitivity are pointwise from term slice precision. Antisymmetry requires that contexts agreeing on all fillings are syntactically equal; this is the extensionality noted with the definition of composition: filling with two distinct terms of class $\mathbf{X}$ identifies the position of the mark, and away from the mark the fillings determine each syntactic component. Mutual precision then gives equality by antisymmetry of term slice precision at every filling. $\square$

Proposition 14 (Context Filling Preserves Precision). *For context slice $c : \mathbf{X} \rightarrow \mathbf{Y}$ and term slice ς of class $\mathbf{X}$. Then if we have slices $\varsigma' \sqsubseteq \varsigma$, $c' \sqsubseteq c$ then also $c'\{\varsigma'\} \sqsubseteq c\{\varsigma\}$.*

Proof. By transitivity it suffices to show $c'\{\varsigma\} \sqsubseteq c\{\varsigma\}$ and $c\{\varsigma'\} \sqsubseteq c\{\varsigma\}$. The first is the definition of context precision, extended from complete terms to slices: by induction on the derivation of $c' \sqsubseteq c$, the filled slice sits at the mark in both sides and is compared by reflexivity, while all other components are compared exactly as before. The second is by induction on the structure of c : at the mark, $\varsigma' \sqsubseteq \varsigma$ by assumption; at every other node the components of c are unchanged and compared by reflexivity, with the congruence rule applied along the path from the root to the mark. $\square$

Lattice Structure

Definition 29 (Context Slice Joins & Meets). *For context slices $c_1 : \mathbf{X} \rightarrow \mathbf{Y}$ and $c_2 : \mathbf{X} \rightarrow \mathbf{Y}$ and any term t of class $\mathbf{X}$:*

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

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

Definition 30 (Purely Structural Contexts). *Least specific slices of some context, containing only gaps $\square$ and the mark $\circ$. The purely structural context of $\mathcal{C}$ is the unique one that is a slice of $\mathcal{C}$*

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

$$\mathcal{T}_s ::= \circ_{typ} \mid \mathcal{T}_s \rightarrow \square \mid \square \rightarrow \mathcal{T}$$

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

Proposition 15 (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. As for term slices, two slices of the same context $\mathcal{C}$ agree structurally wherever neither is a gap, and both retain the mark in the same structural position (precision preserves the mark's position), so joins and meets are total on slices of a common $\mathcal{C}$ and keep the mark in place. The bound and universal properties follow pointwise on fillings from proposition 7, using the pointwise definition of joins and meets. The difference from term slices is only the bottom element: the mark cannot be sliced away, so the least slice is the purely structural context, gaps everywhere except the mark. $\square$

C.2.3. Typing Assumption Contexts & Context Slices

Definition 31 (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 32 (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 16 (Precision is a Partial Order). *Typing assumption context precision forms a partial order on typing assumption context slices.*

Proof. Reflexivity and transitivity are pointwise from precision on typing assumption slices. Antisymmetry holds for the extensional equality of functions used here: mutual precision gives $f'(\gamma) = f(\gamma)$ for every argument γ by antisymmetry on typing assumption slices. (For implementation these functions may instead be treated intensionally with finite domain, as noted at the end of this appendix.) $\square$

Proposition 17 (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 $f'(\gamma') \sqsubseteq f(\gamma)$.*

Proof. The typing assumption context slices arising in this theory are built from identities and extension or removal of assumptions (for example $\gamma \mapsto \gamma \setminus x : v$), all monotone, and composition preserves monotonicity. For monotone f' : $f'(\gamma') \sqsubseteq f'(\gamma) \sqsubseteq f(\gamma)$, the first step by monotonicity and the second by the pointwise definition of $f' \sqsubseteq f$. $\square$

Lattice Structure

Definition 33 (Typing Assumption Context Slice Joins & Meets). *For typing assumption context slices f_1 and f_2 and any typing assumption slice γ :*

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

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

Proposition 18 (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 $\mathcal{F}$.*

Proof. Joins and meets are computed argumentwise, so every bound and universal property reduces, at each argument γ , to the bounded lattice of typing assumption slices. Representative case: if f bounds f_1 and f_2 from above then at each γ , $f(\gamma)$ bounds $f_1(\gamma) \sqcup f_2(\gamma)$, which is $(f_1 \sqcup f_2)(\gamma)$; hence $f_1 \sqcup f_2 \sqsubseteq f$. $\square$

C.2.4. Context Typing Slices

Definition 34 (Expression Context Typing Slice). *An expression context typing slice p is a pair context slice with sub-contexts recursively tagged by typing assumption context slice f . If f is omitted, then assume it is the identity.*

Precision

Definition 35 (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 19 (Precision is a Partial Order). *Expression context typing slice precision forms a partial order on expression context typing slices.*

Proof. This is the product order of context slice precision and typing assumption context slice precision; each law holds componentwise as for expression typing slices. $\square$

Composition

Definition 36 (Expression Context Typing Slice Composition). *It is important that we define composition $c_1^{f_1} \circ c_2^{f_2}$ **syntactically**. That is, the same definition as context substitution but with each sub-context tagged with typing assumption contexts.*

Proposition 20 (Expression Context Slice Composition is Associative). *For all p_1, p_2 , and p_3 then:*

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

Proof. As for untagged contexts, by induction on the structure of the outer context. The tags do not disturb the argument: substitution composes the typing assumption context slices along the path to the mark by function composition, which is itself associative. $\square$

Definition 37 (Expression Context Typing Slice Substitution).

$$\frac{}{\textcolor{blue}{\bigcirc}^{\mathcal{F}}\{t^\gamma\} = t^{\mathcal{F}(\gamma)}} \quad \frac{\mathcal{C}_2^{\mathcal{F}}\{e_2^{\gamma_2}\} = e_2^{\gamma_2'}}{(e_1(\mathcal{C}^{\mathcal{F}_2}))^{\mathcal{F}_1}\{e_2^{\gamma_2}\} = (e_1(e_2'))^{\mathcal{F}_1}\{\gamma_2'\}} \quad \frac{\mathcal{C}_1^{\mathcal{F}}\{e^\gamma\} = e^{\gamma'}}{(\mathcal{C}^{\mathcal{F}_1} : \tau)^{\mathcal{F}_2}\{e^\gamma\} = (e' : \tau)^{\mathcal{F}_2}\{\gamma'\}}$$

etc. (similarly extended to all terms)

Lattice Structure

Definition 38 (Expression Context Typing Slice Joins and Meets). *For expression typing slices $c_1^{f_1}, c_2^{f_2}$:*

$$\begin{aligned} c_1^{f_1} \sqcup c_2^{f_2} &= (c_1 \sqcup c_2)^{f_1 \sqcup f_2} \\ c_1^{f_1} \sqcap c_2^{f_2} &= (c_1 \sqcap c_2)^{f_1 \sqcap f_2} \end{aligned}$$

Proposition 21 (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 $\mathcal{C}^{\mathcal{F}}$.*

Proof. The product of the context slice lattice and the typing assumption context slice lattice, taken componentwise exactly as for expression typing slices. $\square$

Interpreting Context Typing Slices by Contexts and Typing Assumption Contexts

A $\llbracket c \rrbracket$ function can be analogously defined as to expressions, replacing the gaps by $_$, $?$, $\llbracket _ \rrbracket^u$.

C.3. Type-Indexed Slices

C.3.1. Type-Indexed Context Typing Slices

Definition 39 (Type-Indexed Context Typing Slices). *Syntactically defined:*

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

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:

$$\begin{aligned} \overline{p} &= p \\ \overline{p_1 * \mathcal{S}_1 \rightarrow p_2 * \mathcal{S}_2} &= p_1 \circ \overline{\mathcal{S}_1} \sqcup p_2 \circ \overline{\mathcal{S}_2} \end{aligned}$$

Definition 40 (Type-Indexed Context Typing Slice Composition). *For type-indexed context typing slices $\mathcal{S}$ and $\mathcal{S}'$. If $\mathcal{S} = p$ and $\mathcal{S}' = p'$:*

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

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

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

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

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

Proposition 22 (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. Induction on the structure of $\mathcal{S}$, with a case analysis mirroring the definition of composition. The leaf-leaf case is definitional. In the arrow case $\mathcal{S} = p_1 * \mathcal{S}_1 \rightarrow p_2 * \mathcal{S}_2$, composition descends into $\mathcal{S}_1$ and $\mathcal{S}_2$, and the claim follows from the inductive hypotheses together with associativity of composition. The interesting case is a leaf composed with an arrow, $\mathcal{S} = p$ and $\mathcal{S}' = p'_1 * \mathcal{S}'_1 \rightarrow p'_2 * \mathcal{S}'_2$, where

$$\overline{\mathcal{S} \circ \mathcal{S}'} = (p \circ p'_1) \circ \overline{\mathcal{S}'_1} \sqcup (p \circ p'_2) \circ \overline{\mathcal{S}'_2}$$

must equal $p \circ (p'_1 \circ \overline{\mathcal{S}'_1} \sqcup p'_2 \circ \overline{\mathcal{S}'_2})$. This is left composition distributing over joins, which holds because joins are defined pointwise on fillings and filling a fixed slice p with a join is the join of the fillings: outside the mark the two fillings of p agree, so the join there is resolved by idempotence, and at the mark it is exactly the join of the arguments. $\square$

C.3.2. Type-Indexed Expression Typing Slices

(Overloading the $\mathcal{S}$ notation)**Definition 41** (Type-Indexed Expressions Typing Slices). *Syntactically defined:*

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

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:

$$\begin{aligned} \overline{\rho} &= \rho \\ \overline{p_1 * \mathcal{S}_1 \rightarrow p_2 * \mathcal{S}_2} &= p_1\{\overline{\mathcal{S}_1}\} \sqcup p_2\{\overline{\mathcal{S}_2}\} \end{aligned}$$

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

C.3.3. Function Matching

Definition 42 (Slice Function Matching).

$$\begin{aligned} p \blacktriangleright \rightarrow \circ * p \rightarrow \circ * p \quad \rho \blacktriangleright \rightarrow \circ * \rho \rightarrow \circ * \rho \\ p_1 * \mathcal{S}_1 \rightarrow p_2 * \mathcal{S}_2 \blacktriangleright \rightarrow p_1 * \mathcal{S}_1 \rightarrow p_2 * \mathcal{S}_2 \end{aligned}$$

C.3.4. Global Application

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

Definition 43 (Reverse Application). *For an expressions slice ρ . Reverse application $\mid >$ converts type-indexed context typing slices to type-indexed expressions typing slices:*

$$\begin{aligned} \rho \mid > p &= p(\rho) \\ \rho \mid > (p_1 * \mathcal{S}_1 \rightarrow p_2 * \mathcal{S}_2) &= p_1 * (\rho \mid > \mathcal{S}_1) \rightarrow p_2 * (\rho \mid > \mathcal{S}_2) \end{aligned}$$

Proposition 23 (Validity). *If $\mathcal{S}$ is a valid type indexed context typing slice. For all expression typing slices ρ , then $\rho \mid > \mathcal{S}$ is a valid type-indexed expression typing slice with the same structure as $\mathcal{S}$.*

Proof. Induction on the structure of $\mathcal{S}$. The leaf case is immediate: $p(\rho)$ is an expression typing slice. In the arrow case, the components $\rho \mid > \mathcal{S}_i$ are valid with the same structure as $\mathcal{S}_i$ by the inductive hypotheses; the full slice of the result exists because its two components, $p_1\{\rho \mid > \mathcal{S}_1\}$ and $p_2\{\rho \mid > \mathcal{S}_2\}$, are both slices of the one complete term $\mathcal{C}\{e\}$, where $\mathcal{C}$ is the complete context underlying $\mathcal{S}$ and e the complete expression underlying ρ , and slices of a common complete term always have joins (proposition 7). $\square$

Definition 44 (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:*

$$\begin{aligned} f \$ \rho &= f(\rho) \\ f \$ (p_1 * \mathcal{S}_1 \rightarrow p_2 * \mathcal{S}_2) &= \circ * (f \$ p_1 \circ \mathcal{S}_1) \rightarrow \circ * (f \$ p_2 \circ \mathcal{S}_2) \end{aligned}$$

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

Proposition 24 (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. Induction on the structure of $\mathcal{S}$. The leaf case applies f directly. In the arrow case, eager application pushes f to the leaves through the accumulated contexts $p_i \circ \mathcal{S}_i$, so each component is valid with the same structure by the inductive hypotheses, and the mark placed at each $*$ position has the correct class by construction. The eagerness matters: applying f only at the top would leave the inner components as expression slices, misaligning the marks that make the result a context slice at every index; this is the validity property the definition is engineered to preserve. $\square$

C.4. Checking Contexts

Definition 45 (Checking Context). *For term e checking against τ : $\Gamma \vdash e \Leftarrow \tau$. A checking context for e is a context typing $\mathcal{C}^{\mathcal{F}}$, an expression context $\mathcal{C}$ tagged with a typing assumption context $\mathcal{F}$, such that:*

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

Definition 46 (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 25 (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:*

$$\frac{\Gamma \vdash e \Leftarrow \tau}{\Gamma \vdash \circ : \tau \text{ checks } e \text{ against } \tau} \qquad \frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \quad \tau_1 \blacktriangleright \rightarrow \tau_2 \rightarrow \tau \quad \Gamma \vdash e_2 \Leftarrow \tau}{\Gamma \vdash e_1(\circ) \text{ checks } e_2 \text{ against } \tau_2}$$

$$\frac{\Gamma(x) = \tau_1 \quad \Gamma \setminus x : \tau_1 \vdash \mathcal{C}^{\mathcal{F}} \text{ checks } \lambda x. e \text{ against } \tau_1 \rightarrow \tau_2}{\Gamma \vdash \mathcal{C}^{\mathcal{F}} \circ (\lambda x. \circ)^{\Gamma \mapsto \Gamma \setminus x : \tau_1} \text{ checks } e \text{ against } \tau_2}$$

Proof. There are three parts: each rule constructs a checking context, each is minimally scoped, and no other minimally scoped checking contexts exist.

For the first part, verify the three conditions of definition 45 rule by rule. Representative case, the annotation rule: $(\circ : \tau)\{e\} = e : \tau$ synthesises τ by SAsc, whose premise is exactly the retained sub-derivation $\Gamma \vdash e \Leftarrow \tau$, and $\circ : \tau \neq \circ$. The application rule is analogous via SApp. For the lambda rule, the composed context wraps e in $\lambda x. \circ$ and continues with a checking context for $\lambda x. e$, with the typing assumption context removing the binder's assumption, matching AFun.

For the second part, the base cases are immediate: the only proper sub-context of $\circ : \tau$ or $e_1(\circ)$ is $\circ$ itself, excluded by definition. For the lambda rule, a proper sub-context either lies within $\lambda x. \circ$ (so is $\circ$, excluded), or properly contains it and hence corresponds to a proper sub-context of the outer checking context for $\lambda x. e$, which is not a checking context by the inductive hypothesis.

For the third part, inspect where analysis judgements appear as premises of the typing rules (fig. B.2): the annotated term in SAsc, the argument in SApp, and the body under an unannotated lambda in AFun; subsumption has only a synthesis premise. A minimally

scoped checking context ends at the nearest enclosing synthesis judgement, and walking from the analysed term up to that judgement passes only through AFun steps before terminating at an SAsc or SApp step, which is precisely the shape the three rules generate. $\square$

Proposition 26 (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. Existence: induction on the typing derivation, walking upward from the sub-derivation $\Gamma' \vdash e' \Leftarrow \tau'$. Each step upward is one of the three rule forms above (the only typing rules with analysis premises), and the walk terminates at the nearest enclosing synthesis judgement, which exists because the root judgement $\Gamma \vdash e \Rightarrow \tau$ synthesises. Composing the steps yields a minimally scoped checking context whose filling is by construction a sub-term of e (or e itself), and tracking the typing assumption contexts along the AFun steps gives $\mathcal{F}(\Gamma') = \Gamma$ when $\mathcal{C}\{e'\} = e$.

Uniqueness: minimally scoped checking contexts generated by distinct rules have distinct outermost structure, so at most one can match the structure of the occurrence of e' within e ; and the typing derivation fixes which rule applies at each step of the path from e' to its nearest synthesising ancestor. $\square$

C.5. Criterion 1: Synthesis Slices

Definition 47 (Synthesis Slices). *For a synthesising expression, $\text{synthesise } \tau$. A synthesis slice is an expression typing slice ς^Γ of e^Γ which also synthesises τ , that is, $\llbracket \gamma \rrbracket \vdash \llbracket \varsigma \rrbracket \Rightarrow \tau$.*

Proposition 27 (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 exists (and, being a least element, is necessarily unique).*

Proof. The judgement of fig. C.1 computes a candidate, and conjecture 1 asserts it is the minimum; independently, existence can be argued directly. The set of synthesis slices of e^Γ is non-empty (containing e^Γ itself) and finite, so it suffices that it is closed under meets, for then the meet of all synthesis slices is the minimum. Meet closure is checked against the typing rules. Representative case SVar with a fully static τ : a slice synthesising τ must retain both the variable occurrence and its assumption in full, so any two such slices agree on the retained material and their meet still synthesises τ . The essential point is that in the core calculus a synthesis derivation consumes each retained leaf through exactly one rule, with no choice of justification. Exactly this fails in richer calculi, where two sub-terms (for example the branches of an if expression) can each independently justify the synthesised type: there, meets of synthesis slices need not be synthesis slices and no minimum need exist [8]. $\square$

Definition 48 (Type-Indexed Slices of Types).

$$\widehat{b} = b^\emptyset$$

$$\widehat{\tau_1 \rightarrow \tau_2} = (\text{blue circle} \rightarrow \text{green square}) * \widehat{\tau_1} \rightarrow (\text{green square} \rightarrow \text{blue circle}) * \widehat{\tau_2}$$

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

$$\begin{array}{c}
\text{SConst} \frac{}{\Gamma \vdash c \Rightarrow b \mid c^\emptyset} \quad \text{SVar} \frac{x : \tau \in \Gamma}{\Gamma \vdash x \Rightarrow \tau \mid x^{x:\tau}} \quad \text{SVar?} \frac{x : ? \in \Gamma}{\Gamma \vdash x \Rightarrow \tau \mid \square^\emptyset} \\
\text{SFun} \frac{\Gamma, x : \tau_1 \vdash e \Rightarrow \tau_2 \mid \mathcal{S}_2 \quad \llbracket S_2 \rrbracket = \varsigma^\gamma \quad \gamma(x) = \tau'_1}{\Gamma \vdash \lambda x : \tau_1. e \Rightarrow \tau_1 \rightarrow \tau_2 \mid (\lambda \square : \square. \square)^{\gamma \mapsto \gamma \setminus x:\tau_1} * \widehat{\tau_1} \rightarrow (\lambda x : \tau'_1. \square)^{\gamma \mapsto \gamma \setminus x:\tau'_1} * \mathcal{S}_2} \\
\text{SFunUnused} \frac{\Gamma, x : \tau_1 \vdash e \Rightarrow \tau_2 \mid \mathcal{S}_2 \quad \llbracket S_2 \rrbracket = \varsigma^\gamma \quad \gamma(x) = \perp}{\Gamma \vdash \lambda x : \tau_1. e \Rightarrow \tau_1 \rightarrow \tau_2 \mid (\lambda \square : \square. \square)^{\gamma \mapsto \gamma \setminus x:\tau_1} * \widehat{\tau_1} \rightarrow (\lambda \square : \square. \square) * \mathcal{S}_2} \\
\text{SApp} \frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \mid \mathcal{S}}{\Gamma \vdash e_1(e_2) \Rightarrow \tau \mid (\square(\square)) \circ \mathcal{S}} \quad \text{SEHole} \frac{}{\Gamma \vdash \llbracket \square \rrbracket^u \Rightarrow ? \mid \square^\emptyset} \\
\text{SNEHole} \frac{\Gamma \vdash e \Rightarrow \tau}{\Gamma \vdash \llbracket e \rrbracket^u \Rightarrow ? \mid \square^\emptyset} \quad \text{SAsc} \frac{\Gamma \vdash e \Leftarrow \tau}{\Gamma \vdash e : \tau \Rightarrow \tau \mid (\square : \square) \circ \widehat{\tau}}
\end{array}$$

Figure C.1: Minimum synthesis slice calculation

Conjecture 1 (Correctness). *If $\Gamma \vdash e \Rightarrow \tau$ then:*

- $\Gamma \vdash e \Rightarrow \tau \mid \rho$ 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 $\rho \sqsubseteq \rho'$.

C.6. Criterion 2: Analysis Slices

Definition 49 (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 $\Gamma \vdash e \Leftarrow \tau$.

Proposition 28 (Minimum Analysis Slices). *The minimum analysis slice analysing e in a checking context $\mathcal{C}^\mathcal{F}$ is an analysis slice p such that any other analysis slice p' is at least as specific, $p \sqsubseteq p'$. This minimum always exists.*

Existence follows as for minimum synthesis slices: analysis slices of a checking context are finitely many, non-empty ($\mathcal{C}^\mathcal{F}$ itself), and closed under meets over the core calculus. The judgement below computes the minimum; that it does so is conjecture 2. As with synthesis slices, existence is particular to the core calculus [8].

Issues with checking equality of functions might arise in implementing this. But, typing assumption contexts can be made intensional by assuming finite domain. It is probably also possible to remove function equality checks entirely.

Conjecture 2 (Correctness). *If $\mathcal{C}$ is a checking context for e and $\Gamma \vdash e \Leftarrow \tau$. Then $\Gamma; \mathcal{C} \vdash e \Leftarrow \tau \mid \mathcal{S}$ and $\mathcal{S}$ is the minimum analysis slice of e .*

C.7. Criterion 3: Contribution Slices

Definition 50 (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:

$\boxed{\Gamma; \mathcal{C} \vdash e \Leftarrow \tau \mid \mathcal{S}}$ e which analyses against type τ in minimally scoped checking context $\mathcal{C}$ and typing context Γ has minimum type-indexed analysis slice $\mathcal{S}$.

$$\begin{array}{c}
\text{AAnnot} \frac{\Gamma \vdash (\bigcirc : \tau)^{\mathcal{F}} \text{ checks } e \text{ against } \tau}{\Gamma; (\bigcirc : \tau)^{\mathcal{F}} \vdash e \Leftarrow \tau \mid (\varsigma^{\gamma} \mapsto (\bigcirc : \varsigma)^{\gamma}) \$ \hat{\tau}} \\
\text{AApp} \frac{\Gamma \vdash (e_1(\bigcirc))^{\mathcal{F}} \text{ checks } e_2 \text{ against } \tau_2 \quad \mathcal{F}(\Gamma) \vdash e_1 \Rightarrow \tau_1 \mid \mathcal{S}_1 \quad \mathcal{S}_1 \blacktriangleright \rightarrow p_2 * \mathcal{S}_2 \rightarrow p * \mathcal{S}}{\Gamma; (e_1(\bigcirc))^{\mathcal{F}} \vdash e_2 \Leftarrow \tau_2 \mid (\varsigma^{\gamma} \mapsto (\varsigma(\bigcirc))^{\gamma' \mapsto \gamma' \sqcup \gamma}) \$ p_2 \circ \mathcal{S}_2} \\
\text{AFun} \frac{\Gamma(x) = \tau_1 \quad \mathcal{F}_1(\Gamma); \mathcal{C}^{\mathcal{F}_2} \vdash \lambda x. e \Leftarrow \tau_1 \rightarrow \tau_2 \mid \mathcal{S} \quad \mathcal{S} \blacktriangleright \rightarrow p_1 * \mathcal{S}_1 \rightarrow p_2 * \mathcal{S}_2}{\Gamma; \mathcal{C}^{\mathcal{F}_2} \circ (\lambda x. \bigcirc)^{\mathcal{F}_1} \vdash e \Leftarrow \tau_2 \mid p_2 \circ \mathcal{S}_2 \circ (\lambda \square. \bigcirc)^{\gamma \mapsto \gamma \setminus x:\tau_1}}
\end{array}$$

As composition has been defined **syntactically**, then $\mathcal{C}^{\mathcal{F}_2} \circ (\lambda x. \bigcirc)^{\mathcal{F}_1}$, is a valid way to pattern match upon contexts, hence contexts for AFun can be deconstructed uniformly in such a way.

Figure C.2: Minimum analysis slice calculation

- 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 $\Gamma \vdash e \Rightarrow \tau \mid_{\mathcal{C}} \mathcal{S}$ and $\Gamma; \mathcal{C} \vdash e \Leftarrow \tau \mid_{\mathcal{C}} \mathcal{S}_e \mid \mathcal{S}_c$, where $\mathcal{S}_e$ is the expression slice part and $\mathcal{S}_c$ is the contextual part:

$$\frac{\Gamma \vdash e \Rightarrow \tau \mid_{\mathcal{C}} \mathcal{S}_s \quad \tau \sim \tau' \quad \Gamma; \mathcal{C} \vdash e \Leftarrow \tau' \mid \mathcal{S}_c}{\Gamma; \mathcal{C} \vdash e \Leftarrow \tau' \mid_{\mathcal{C}} \text{match}(\mathcal{S}_c, \mathcal{S}_s) \mid \mathcal{S}_c}$$

Where match takes the matches the *right* slice to have the same structure as the *left slice*, expanding with funmatching if required:

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

$$\text{match}(p, - \rightarrow -) = \square^{\emptyset}$$

$$\text{match}(- \rightarrow -, p) = \bigcirc * p \rightarrow \bigcirc * p$$

$$\text{match}(c_1 * \mathcal{S}_1 \rightarrow c_2 * \mathcal{S}_2, c'_1 * \mathcal{S}'_1 \rightarrow c'_2 * \mathcal{S}'_2) = c_1 * \text{match}(\mathcal{S}_1, \mathcal{S}'_1) \rightarrow c_2 * \text{match}(\mathcal{S}_2, \mathcal{S}'_2)$$

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

C.8. Cast Syntax

The below demonstrates inserting synthesis and analysis slices, but contribution slices could equally be used instead. Extend the cast syntax to casts between types and synthesis and or analysis slices. The pair of slices explain why the cast was enforced, and they will always match structurally with the annotated type.

To avoid conflict, analysis (contextual) slices are now notated $\mathcal{A}$. Generally speaking a cast will be *from* synthesised types *to* analysed expectations.

$$\langle \tau_1 \mid \mathcal{S}_1 \mid \mathcal{A}_1 \Rightarrow \tau_2 \mid \mathcal{A}_2 \mid \mathcal{S}_2 \rangle$$

If an analysis slice or synthesis slice are omitted, assume they are $\bigcirc^\emptyset$ or $\square^\emptyset$ respectively.

C.9. Elaboration

Nicely complements existing elaboration rules (fig. B.5).

Note that elaboration adds a new situation analysing left application parts. The ESApp rule is split explicitly into the cases when $\tau_1 = ?$, as the cast on d_1 need only be inserted in this case. We are effectively adding in a new analysis slice for e_1 in $\bigcirc(e_2)$ when e_1 synthesises $?$. The fact it is a function is then derived from the outer context:

Definition 51 (Dynamic Synthesising Function Analysis Slice).

$$\frac{\Gamma \vdash e_1 \Rightarrow ?}{\Gamma \vdash \bigcirc(e_2) \text{ checks } e_1 \text{ against } \tau_2 \rightarrow \tau}$$

$$\frac{AFunDyn \quad \Gamma \vdash (\bigcirc : \tau)^{\mathcal{F}} \text{ checks } e \text{ against } \tau}{\Gamma; (\bigcirc(e_2))^{\mathcal{F}} \vdash e_1 \Leftarrow \tau_2 \rightarrow \tau \mid \bigcirc * (\bigcirc(\square))^{\mathcal{F}} \rightarrow \bigcirc * (\bigcirc(\square))^{\mathcal{F}}}$$

$\boxed{\Gamma \vdash e \Rightarrow \tau \mid \mathcal{S} \rightsquigarrow d \dashv \Delta}$ e synthesises type τ with synthesis slice $\mathcal{S}$ and elaborates to d and hole context Δ .

$$\begin{array}{c}
\text{ESConst} \frac{}{\Gamma \vdash c \Rightarrow b \mid c^\emptyset \rightsquigarrow c \dashv \emptyset} \quad \text{ESVar} \frac{x : \tau \in \Gamma}{\Gamma \vdash x \Rightarrow \tau \mid x^{x:\tau} \rightsquigarrow x \dashv \emptyset} \\
\text{ESFunUnused} \frac{\Gamma, x : \tau_1 \vdash e \Rightarrow \tau_2 \mid \mathcal{S}_2 \rightsquigarrow d \dashv \Delta \quad \llbracket \mathcal{S}_2 \rrbracket = \varsigma^\gamma \quad \gamma(x) = \perp}{\Gamma \vdash \lambda x : \tau_1. e \Rightarrow \tau_1 \rightarrow \tau_2 \mid (\lambda \square : \circ. \square)^{\gamma \mapsto \gamma \setminus x : \tau_1} * \widehat{\tau_1} \rightarrow (\lambda \square : \square. \circ) * \mathcal{S}_2 \rightsquigarrow \lambda x : \tau_1. d \dashv \Delta} \\
\text{ESFun} \frac{\Gamma, x : \tau_1 \vdash e \Rightarrow \tau_2 \mid \mathcal{S}_2 \rightsquigarrow d \dashv \Delta \quad \llbracket \mathcal{S}_2 \rrbracket = \varsigma^\gamma \quad \gamma(x) = \tau'_1}{\Gamma \vdash \lambda x : \tau_1. e \Rightarrow \tau_1 \rightarrow \tau_2 \mid (\lambda \square : \circ. \square)^{\gamma \mapsto \gamma \setminus x : \tau_1} * \widehat{\tau_1} \rightarrow (\lambda x : \tau'_1. \circ)^{\gamma \mapsto \gamma \setminus x : \tau'_1} * \mathcal{S}_2 \rightsquigarrow \lambda x : \tau_1. d \dashv \Delta} \\
\text{ESApp} \frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \mid \mathcal{S}_1 \quad \tau_1 \blacktriangleright \tau_2 \rightarrow \tau \quad \mathcal{S}_1 \blacktriangleright p_2 * \mathcal{S}_2 \rightarrow p * \mathcal{S} \quad \Gamma; \circ(e_2) \vdash e_1 \Leftarrow \tau_2 \rightarrow \tau \mid \mathcal{A}_1 \rightsquigarrow d_1 : \tau'_1 \mid \mathcal{S}'_1 \mid \mathcal{A}'_1 \dashv \Delta_1 \quad \Gamma; e_1(\circ) \vdash e_2 \Leftarrow \tau_2 \mid \mathcal{A}_2 \rightsquigarrow d_2 : \tau'_2 \mid \mathcal{S}'_2 \mid \mathcal{A}'_2 \dashv \Delta_2}{\Gamma \vdash e_1(e_2) \Rightarrow \tau \rightsquigarrow (d_1 \langle \tau'_1 \mid \mathcal{S}'_1 \mid \mathcal{A}'_1 \Rightarrow \tau_2 \rightarrow \tau \mid \mathcal{A}_1 \mid p_2 * \mathcal{S}_2 \rightarrow p * \mathcal{S} \rangle) (d_2 \langle \tau'_2 \mid \mathcal{S}'_2 \mid \mathcal{A}'_2 \Rightarrow \tau_2 \mid \mathcal{A}_2 \rangle) \dashv \Delta_1 \cup \Delta_2} \\
\text{ESEHole} \frac{}{\Gamma \vdash \mathbb{O}^u \Rightarrow ? \rightsquigarrow \square^\emptyset \dashv \mathbb{O}_{\text{id}(\Gamma)}^u u :: \mathbb{O}[\Gamma]} \\
\text{ESNEHole} \frac{\Gamma \vdash e \Rightarrow \tau \mid \mathcal{S} \rightsquigarrow d \dashv \Delta}{\Gamma \vdash \langle e \rangle^u \Rightarrow ? \mid \square^\emptyset \rightsquigarrow \langle d \rangle_{\text{id}(\Gamma)}^u \dashv \Delta, u :: \mathbb{O}[\Gamma]} \\
\text{ESAsc} \frac{\Gamma; \circ : \tau \vdash e \Leftarrow \tau \mid \mathcal{A} \rightsquigarrow d : \tau' \mid \mathcal{S}' \mid \mathcal{A}' \dashv \Delta}{\Gamma \vdash e : \tau \Rightarrow \tau \mid (\square : \circ) \circ \widehat{\tau} \rightsquigarrow d \langle \tau' \mid \mathcal{S}' \mid \mathcal{A}' \Rightarrow \tau \mid \mathcal{A} \rangle \dashv \Delta}
\end{array}$$

$\boxed{\Gamma; \mathcal{C}^\mathcal{F} \vdash e \Leftarrow \tau \mid d \rightsquigarrow \tau' : \mathcal{S}' \mid \mathcal{A}' \mid \Delta \dashv}$ e in minimally scoped checking context $\mathcal{C}^\mathcal{F}$ analyses against type τ with analysis slice $\mathcal{A}$ and elaborates to d of consistent type τ' which has synthesis slice part $\mathcal{S}'$ and analysis slice part $\mathcal{A}'$ (structurally matching the synthetic part).

$$\begin{array}{c}
\text{EAFun} \frac{\Gamma; \mathcal{C} \vdash \lambda x. e \Leftarrow \tau \mid \mathcal{A} \quad \tau \blacktriangleright \tau_1 \rightarrow \tau_2 \quad \mathcal{A} \blacktriangleright p_1 * \mathcal{A}_1 \rightarrow p_2 * \mathcal{A}_2 \quad \Gamma, x : \tau_1; \mathcal{C} \circ (\lambda x. \circ)^{\Gamma \mapsto \Gamma \setminus x : \tau_1} \vdash e \Leftarrow \tau_2 \mid \mathcal{A}_2 \rightsquigarrow d : \tau'_2 \mid \mathcal{S}'_2 \mid \mathcal{A}'_2 \dashv \Delta \quad \llbracket \mathcal{S}'_2 \rrbracket = \varsigma^\gamma \quad \gamma(x) = \tau'_1 \quad \mathcal{A}'_2 = \mathcal{A}_2'' \circ (\lambda pt. \circ)^\mathcal{F}}{\Gamma; \mathcal{C} \vdash \lambda x. e \Leftarrow \tau \mid \mathcal{A} \rightsquigarrow \lambda x : \tau_1. d : \tau_1 \rightarrow \tau'_2 \mid \circ * \square^\emptyset \rightarrow (\lambda x. \circ)^{\gamma \mapsto \gamma \setminus x : \tau'_1} * \mathcal{S}'_2 \mid p_1 * \mathcal{A}_1 \rightarrow \circ * \mathcal{A}_2'' \dashv \Delta} \\
\text{EAFunUnused} \frac{\Gamma; \mathcal{C} \vdash \lambda x. e \Leftarrow \tau \mid \mathcal{A} \quad \tau \blacktriangleright \tau_1 \rightarrow \tau_2 \quad \mathcal{A} \blacktriangleright p_1 * \mathcal{A}_1 \rightarrow p_2 * \mathcal{A}_2 \quad \Gamma, x : \tau_1; \mathcal{C} \circ (\lambda x. \circ)^{\Gamma \mapsto \Gamma \setminus x : \tau_1} \vdash e \Leftarrow \tau_2 \mid \mathcal{A}_2 \rightsquigarrow d : \tau'_2 \mid \mathcal{S}'_2 \mid \mathcal{A}'_2 \dashv \Delta \quad \llbracket \mathcal{S}'_2 \rrbracket = \varsigma^\gamma \quad \gamma(x) = \perp \quad \mathcal{A}'_2 = \mathcal{A}_2'' \circ (\lambda pt. \circ)^\mathcal{F}}{\Gamma; \mathcal{C} \vdash \lambda x. e \Leftarrow \tau \mid \mathcal{A} \rightsquigarrow \lambda x : \tau_1. d : \tau_1 \rightarrow \tau'_2 \mid \circ * \square^\emptyset \rightarrow (\lambda \square. \circ) * \mathcal{S}'_2 \mid p_1 * \mathcal{A}_1 \rightarrow \circ * \mathcal{A}_2'' \dashv \Delta} \\
\text{EASubsume} \frac{e \neq \mathbb{O}^u \quad e \neq \langle e' \rangle^u \quad \Gamma; \mathcal{C} \vdash e \Leftarrow \tau \mid \mathcal{A} \quad \Gamma \vdash e \Rightarrow \tau' \mid \mathcal{S}' \rightsquigarrow d \dashv \Delta \quad \tau \sim \tau'}{\Gamma; \mathcal{C} \vdash e \Leftarrow \tau \mid \mathcal{A} \rightsquigarrow d : \tau' \mid \mathcal{S}' \mid \text{match}(\mathcal{S}', \mathcal{A}) \dashv \Delta} \\
\text{EAEHole} \frac{\Gamma; \mathcal{C} \vdash \mathbb{O}^u \Leftarrow \tau \mid \mathcal{A}}{\Gamma; \mathcal{C} \vdash \mathbb{O}^u \Leftarrow \tau \mid \mathcal{A} \rightsquigarrow \mathbb{O}_{\text{id}(\Gamma)}^u : \tau \mid \text{match}(\mathcal{A}, \square^\emptyset) \mid \mathcal{A} \dashv u :: \tau[\Gamma]} \\
\text{EANEHole} \frac{\Gamma \vdash e \Rightarrow \tau' \mid \mathcal{S}' \rightsquigarrow d \dashv \Delta \quad \Gamma; \mathcal{C} \vdash \langle e \rangle^u \Leftarrow \tau \mid \mathcal{A}}{\Gamma; \mathcal{C} \vdash \langle e \rangle^u \Leftarrow \tau \mid \mathcal{A} \rightsquigarrow \langle d \rangle_{\text{id}(\Gamma)}^u : \tau \mid \text{match}(\mathcal{A}, \square^\emptyset) \mid \mathcal{A} \dashv u :: \tau[\Gamma]}
\end{array}$$

Figure C.3: Elaboration judgements

The inserted cast on d_1 in ESApp goes from the actual synthesised type for d_1 (which could be $?$) to its corresponding function type. Both the analysis part requiring it to be a *function* (any function) and the parts specifying its component types if synthesised are retained. Analysis elaboration produces both synthesis and analysis slices as EAFun must retain the argument context slice to explain unannotated functions, while the synthesis part is determining unannotated functions' output type.

C.10. Dynamics

Final forms, contextual dynamic, and hole substitution remain the same as the Hazel calculus. But the cast transitions need to work with cast slices. The only difference is that instruction transitions require the use of slice ground matching.

$S \blacktriangleright_{\text{ground}} S'$ S matches ground slices S'

$$\frac{\tau_1 \rightarrow \tau_2 \neq ? \rightarrow ?}{\blacktriangleright_{\text{ground}} ? \rightarrow ? \mid p_1 * \square^\emptyset \rightarrow p_2 * \square^\emptyset \mid p'_1 * \text{struct}(\llbracket \mathcal{A}_1 \rrbracket) \rightarrow p'_2 * \text{struct}(\llbracket \mathcal{A}_2 \rrbracket)}$$

Where $\text{struct}(p)$ takes the *purely structural* context slice of p .

Figure C.4: Ground type matching for slices

Future Work: When type-indexed synthesis slices are converted to type-indexed analysis slices, they lose their incremental structure. A way to retain this would be beneficial. This could be done by pairing synthesis slices with a global two-hole context.

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 D.1: More Specific Matches

To account for this, we can take ideas from pattern matrix techniques for producing exhaustivity warnings, [47]. 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.

Supplementary Results and Corpus Data

Statistics on the corpus are found in fig. E.1.

	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 E.1: Hazel Program Corpus

The averages for each search method over their successful programs for each implementation are given in fig. E.2. Note that each method succeeds on *differing* sets of programs.

	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 E.2: Benchmarks: Search Implementations

Cast are between slices ‘from’ a type ‘to’ a type. Their average sizes given in fig. E.3.

Averages		Subdivisions			
	unit	Ok		Errors	
		from	to	from	to
Cast Slice	size	5.5	1.2	5.9	1.5
Std. dev.		8.1	3.7	7.1	2.0
Proportion	%	1	0.2	1	0.2
Std. dev.		1	0.5	1	0.4
<i>(Unannotated)</i>					
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
<i>(Annotated)</i>					

Figure E.3: Effectiveness: Cast Slices

Trace length and instantiation size data in fig. E.4.

	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 E.4: Witness & Trace Sizes

Some examples of programs in the translated corpus are given in fig. E.5. All three were examples where BDFS did not terminate, and are accompanied with their failure classification.

```

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 E.5: (Paraphrased) Failure Examples

On Representing Non-Determinism

High Level Representation

Other high level representations, besides monads, include:

Logic Programming DSL: Languages like Prolog [16] and Curry [7] 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 [21] created a tagless final style [39] domain specific language within OCaml. This approach fully abstracts the search procedure from the non-deterministic algorithm constructs.

Delimited Continuations: Delimited continuations [4, 67] 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 [6, 35], while handlers can flexibly define the search procedure and accounting logic, e.g. storing solutions in a list. As with delimited continuations, to try multiple solutions, the continuations must be cloned.

Reasoning Against

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

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

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

Optimised DSL: Introducing a formal DSL including optimisations, such as the proposed Tagless-Final DSL [39, 21], is very complex. However, this would allow more flexibility in writing non-deterministic evaluation, with some optimisations made automatically by the DSL.

¹An important dependency of Hazel.