Decomposable Type Highlighting for Bidirectional Type and Cast Systems

MAX CARROLL, University of Cambridge, UK
ANIL MADHAVAPEDDY, University of Cambridge, UK
PATRICK FERRIS, University of Cambridge, UK

We explore how to provide programmers with an interactive interface for explaining the process by which static types and dynamic casts are derived, with the goal of improving the debugging of static and dynamic type errors. To this end, we define mathematical foundations for a decomposable highlighting system within a bidirectional system and show how these can be propagated through dynamic types in a cast system. Our prototype implementation in the gradually typed Hazel language includes a web-based user interface, through which we highlight the importance of type-level debugging.

1 INTRODUCTION

In static typing, blame for type errors is 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 are related to *multiple* locations [21] and furthermore, the errors only state the expected types without explanation for *why* they occur. In dynamic typing, errors do not typically specify any source code context that caused them, instead relying on the interpretation of (potentially complex and extensive) execution traces.

Vision: We seek to improve user understanding of static and dynamic type systems and type errors by providing a *formally* complete and decomposable highlighting system for bidirectional type systems (type slicing) and propagation of this information through dynamic cast systems (cast slicing). This would allow users to interactively explore *why* an expression has been typed by decomposing the highlighted segments by their influence on the expression's type, inspecting only the particular parts they do not understand. Further, in languages with dynamic type information, this highlighting information can be propagated through, for example, dynamic casts, providing source code context to explain why a cast was executed during evaluation.[1]

Progress: This paper lays out our approach to building mathematical foundations using Hazel [1], a research language that allows incomplete programs (with holes) with a focus on liveness, interaction design [3, 11, 12] and learning [8, 16]. Hazel is gradually typed, so these highlighting methods can be explored for explaining both static and dynamic types (and type errors). However, the foundations of this work apply more generally to many bidirectional type systems and cast systems. While this paper focuses on giving definitions and expected properties[2] of a formal foundation, a preliminary implementation is also deployed at https://hazel.org/build/witnesses-type-slicing/ for example usage.

2 BACKGROUND

First, we introduce the notions of bidirectional types, cast systems, gradual types, and the core Hazel calculus for reference.

[1]For either a compiler-generated or user-inserted cast.
[2]No proof mechanisation has been performed as of yet. But 'informal' proofs of most properties have been explored.

Authors' addresses: Max Carroll, mc2372@cam.ac.uk, mjvcarroll@gmail.com, University of Cambridge, Department of Computer Science and Technology, Cambridge, UK; Anil Madhavapeddy, avsm2@cam.ac.uk, University of Cambridge, Department of Computer Science and Technology, Cambridge, UK; Patrick Ferris, pf341@cam.ac.uk, University of Cambridge, Department of Computer Science and Technology, Cambridge, UK.

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

(a) Variables synthesise their type from the typing assumptions.

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

(b) Annotations synthesise the annotated type.

$$\frac{\Gamma \vdash e \Rightarrow \tau}{\Gamma \vdash e \Leftarrow \tau}$$

(c) Subsumption: A synthesising term checks against the same type.

2.1 Bidirectional Type Systems

A *bidirectional type system* [7] takes on a more algorithmic definition of typing judgements, being more intuitive to implement, while still allowing some amount of local type inference.

This is done in a similar way to annotating logic programs [20, pg. 123], by specifying the *mode* of the type parameter in a typing judgement, distinguishing when it is an *input* (type analysis/checking) and when it is an *output* (type synthesis). We express this with two judgements, read respectively as: *e synthesises an (output) type τ/analyses against an (input) type τ under typing context* Γ:

$$\Gamma \vdash e \Rightarrow \tau \qquad\qquad \Gamma \vdash e \Leftarrow \tau$$

Such languages should be *mode correct*[3] [5] and will have three obvious rules. That variables can synthesise their type, if it is accessible from the typing assumptions. Annotated terms synthesise their type from the annotation. Subsumption: a synthesising term must check against that same type.

2.2 (Dynamic) Cast Calculi

A cast calculus adds casts between types to an operational semantics. More specifically, we consider a dynamically typed system, with a distinguished dynamic type, notated $?$.

Cast expressions will be denoted $e\langle\tau_1 \Rightarrow \tau_2\rangle$ for expression e and types τ_1, τ_2, representing that e has type τ_1 and is cast to new type τ_2. Compound type casts can 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:

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

Or if f has the dynamic type, it should still be treated as a possible function:

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

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 to a corresponding value of the new type. The cast on the argument is reversed, in a similar vein to the contravariance of function argument types under sub-typing.

2.3 Gradual Type Systems

A *gradual type system* [17, 18] 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 (pseudo-OCaml syntax) type checks:

```
let x : ? = 10 in   /* 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"`. Typically, the language is split into two parts:

[3]Ensuring that they can be easily implemented algorithmically. That is, never require the 'guessing' of inputs.

- 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.[4] The example above would reduce to a *cast error*:

```
10⟨Int⇒?⇏String⟩ ^ "str"
```

This is possible by introduction of a *consistency* equivalence relation notated $\tau_1 \sim \tau_2$ used in place of type equality in the typing rules. Consistency is a weakening of equality: all types are consistent with the dynamic type ?, and compound types are consistent if their sub-parts are:

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

Finally, 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 by *elaboration*, $\Gamma \vdash e \rightsquigarrow d : \tau$ read as: *external expression* e *is elaborated to internal expression* d *with type* τ *under typing context* Γ. For example, to insert casts around function applications:

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

Where $\blacktriangleright_{\to}$ explicitly pattern matches function types, including ? where $? \blacktriangleright_{\to} ? \to ?$. We place a cast on the function[5] d_1 to $\tau_2 \to \tau$ and on the argument d_2 to the function's expected argument type τ_2 to perform runtime type checking of arguments. Intuitively, casts must be inserted whenever type consistency is used, but deciding which casts to insert is non-trivial [6].

The runtime semantics of the internal expression is that of the *dynamic cast system* discussed above (2.2). A cast is succeeds if and only if the types are *consistent*.

2.4 The Hazel Calculus

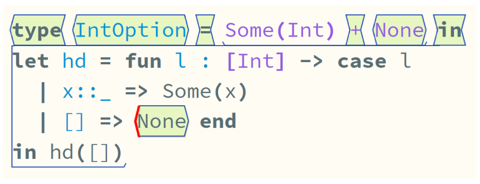
Hazel is built upon a bidirectional and gradually typed core lambda calculus [13]. It additionally includes *holes*, which can both be typed and treated as an indeterminate *final form* (value), allowing evaluation to proceed around them seamlessly. Errors can be treated as holes (i.e. with dynamic/unknown type) to allow for continued evaluation.

The core calculus [13] is a gradually and bidirectionally typed lambda calculus. Therefore it has a gradual and locally inferred bidirectional *external language* elaborated to an explicitly typed *internal language* including cast expressions. Holes will also be notated by ? and naturally synthesise the dynamic type.[6]

3 PRELIMINARY IMPLEMENTATION

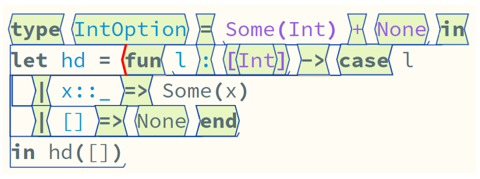
Next, we briefly demonstrate, by example in Hazel, the ideas of type and cast slicing, before diving into the mathematical foundations. Figure 2 shows the type slices of four sub-expressions at the cursor (in red). Typing of a sub-expression is done under typing assumptions; the UI also highlights the type slice of a variable or type definition if the assumption is required in order to type check.[7]

[4]i.e. the proposed *dynamic type system* above.
[5]This cast is required, as if $\tau_1 = ?$ then we need a cast to realise that it is even a function. Otherwise $\tau_1 = \tau_2 \to \tau$ and the cast is redundant.
[6]Notation here differs from the original Hazel paper [13], in order to be consistent with Hazel UI. Hole meta-variables and non-empty holes are of little interest to this paper, so are omitted.
[7]For example, the `IntOption` type definition, or the hd binding, in fig. 2



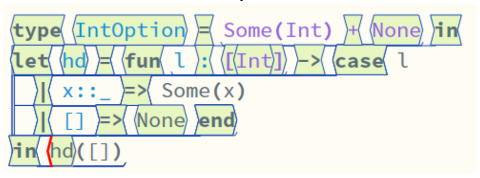
```
type IntOption = Some(Int) + None in
let hd = fun l : [Int] -> case l
  | x::_ => Some(x)
  | [] => None end
in hd([])
```

(a) None synthesises IntOption due to it being a value of IntOption. The assumption that None has type IntOption results in the slice of its definition to also be highlighted, notice also the inclusion of the alias binding for IntOption



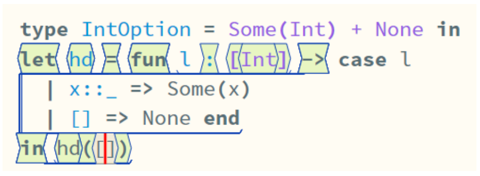
```
type IntOption = Some(Int) + None in
let hd = fun l : [Int] -> case l
  | x::_ => Some(x)
  | [] => None end
in hd([])
```

(b) The function synthesises [Int]⟶ IntOption due to its [Int] annotation and that the match branches synthesis IntOption. Both branches provide the same type information, so 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 hd synthesises [Int]⟶ IntOption by assumption similarly to (a). The slice of the definition of hd is also highlighted.

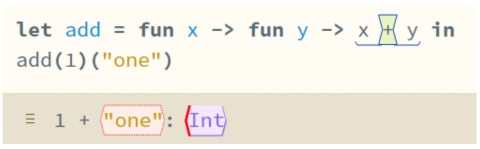


```
type IntOption = Some(Int) + None in
let hd = fun l : [Int] -> case l
  | x::_ => Some(x)
  | [] => None end
in hd([])
```

(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].

Fig. 2. Type Slicing Examples

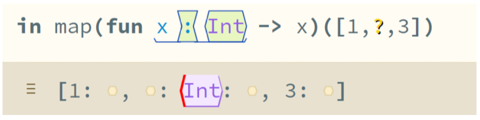
Casts between types are inserted around expression, or explicitly by the programmer. Type slices can be associated with these casted types, highlighting the source code that enforced their automated insertion by the compiler during elaboration, or the corresponding cast written in the code directly by the programmer. Figure 3 demonstrates two simple examples, the first of which shows how cast slicing could be use to perform error highlighting for dynamic errors.[8]



```
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.



```
in map(fun x : Int -> x)([1,?,3])

≡ [1: ○, ○: Int: ○, 3: ○]
```

(b) A hole (notated '?') cast to Int due to being the argument of a mapped function annotated with an Int input.

Fig. 3. Cast Slicing Examples

4 TYPE SLICING THEORY

This section details the underlying mathematical foundation, first defining some preliminary constructs, used to define two core slicing criteria: *Synthesis Slices*, and *Analysis Slices*, which minimally highlight the parts (slice) of a term directly causing the term to *synthesise* a type or of its context[9] enforcing it to *analyse against* some type.

[8]Both these examples select the type being casted to. But, the system works equally well selecting the castee type instead; however, the Hazel UI does not yet have a way to select the castee type.
[9]The code surrounding the term.

4.1 Expression Typing Slices

First, we introduce what *slices* are. The aim is to provide a formal representation of term highlighting.

4.1.1 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 expression highlights as shown on the left. This is encoded, on the right, by representing omitted sub-terms by *gap* terms, notated $\square$:

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

We can then define a *precision* partial order [10] 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* [4]. 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.

4.1.2 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. We represent typing assumptions by *partial functions* mapping variables to types. Hence, their slices are just partial functions to *type slices*. A slice must map a (possibly equal) subset of the variables to less or equally precise types. Precision, meets, and joins, can be defined pointwise:

Definition 4.1 (Typing Assumption Slice Precision). For typing assumption slices γ_1, γ_2. Where $\mathrm{dom}(f)$ is the set of variables for which a partial function f is *defined*:

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

Definition 4.2 (Typing Assumption Slice Joins and Meets). For typing slices γ_1, γ_2, and any variable x:

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

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

4.1.3 Expression Typing Slices. 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. These slices are the core construct for synthesis slices.

4.1.4 Typing. Expression slices can be *type checked* under the *type assumption slices* by replacing gaps $\square$ by: holes of any meta-variable ? in *expressions*, fresh variables in *patterns*, and the dynamic type in *types* (notated by $[\![-]\!]$). Other (non-gradual) systems require different interpretations, for example, a value of polymorphic type $\forall\alpha.\alpha$ could be used in place of gaps. Some form of polymorphism is required in order to determine at what point we have removed so much information that the sliced term has a *more general* (more polymorphic) type than before.

Definition 4.3 (Expression Typing Slice Type Checking). For expression typing slice ς^γ and type τ. $\gamma \vdash \varsigma \Rightarrow \tau$ iff $[\![\gamma]\!] \vdash [\![\varsigma]\!] \Rightarrow \tau$ and $\gamma \vdash \varsigma \Leftarrow \tau$ iff $[\![\gamma]\!] \vdash [\![\varsigma]\!] \Leftarrow \tau$.

4.2 Context Typing Slices

An expression's analysing type is enforced by its *surrounding context*. We must note a clash in terminology between *contexts* of a term (the part of a program surrounding a sub-term, such as those as used in contextual dynamics) opposing the *typing contexts* used to refer to typing assumptions made during type checking in the standard literature. For clarity, we refer to the typing assumptions as directly as is, and never refer to them as 'contexts'.

For example, the type of the underlined expression below is enforced by the surrounding highlighted context (the annotation):

$$\underline{(\lambda x.?)} : \texttt{Bool} \longrightarrow \texttt{Int}$$

4.2.1 *Contexts and Their Slices.*

We represent these surrounding contexts by a *term context* C, which marks *exactly one* sub-term as $\bigcirc$. Where $C\{t\}$ substitutes term t for the mark $\bigcirc$ in C[10] and composition is defined as substituting contexts into contexts, notated infix by $\circ$.

Contexts extend to context slices analogously to term slices and are notated as c. 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 $\bigcirc \not\sqsubseteq \bigcirc(1)$. This can be concisely defined pointwise.

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: $\bigcirc \not\sqcap \bigcirc(\square)$.

4.2.2 *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.

4.2.3 *Context Typing Slices.*

An *expression context typing slice*, d, is a context slice with each sub-context recursively tagged by typing assumption context slices. Then, retrieve the underlying context with $\mathrm{ctx}(d)$ and typing assumption context with $\mathrm{typ}(d)$ [11]. As before, lattice relations are defined componentwise. Gaps can be interpreted by holes during type checking analogously to expression typing slices.

4.3 Indexed Type Slices

Decomposing slices of expressions with compound types (i.e. functions) according to their component types is the core idea of this method. For example, the following context slice on the left explains why the underlined term analyses $\texttt{Bool} \longrightarrow \texttt{Int}$, and the right is a sub-slice explaining only the argument type:

$$\underline{(\lambda x.?)} : \texttt{Bool} \longrightarrow \texttt{Int} \qquad \underline{(\lambda x.?)} : \texttt{Bool} \longrightarrow \texttt{Int}$$

This has many uses which are core to both the user experience and the internal implementation:

- The programmer could use this to query sub-slices for only the sub-parts of the type of an expression that they do not already understand, hence omitting unneeded information and reducing the amount of highlighting. For example, if they knew why the argument of a function was an integer, but not why the return type was.

[10] Only allowed if the marked position expects a term of the same class as t (pattern `Pat`, type `Typ`, or expression `Exp`).
[11] By composing all individual contexts up from the mark $\bigcirc$

- Casts are decomposed throughout evaluation: a cast between function types will be separated into a cast between the two argument types and a cast between the two return types. Being able to decompose slices allows only the relevant source code to be retained.
- Internally, calculating slices is easier when sub-slices are available, during function application. See fig. 4.

The main property that indexed-slices should maintain is *reconstructability*: that slices can be reconstructed from their sub-parts by joining the sub-slices. 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. We only consider *context slices* here, but expressions slices are type-indexed analogously. Context slices are syntactically defined to correspond with the structure of types:

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

But they must be *reconstructable*. That is, the *full slice* of $\mathcal{S}$, notated $\overline{\mathcal{S}}$ must exist by joining the sub-slices within their contexts:

$$\overline{d} = d \qquad\qquad \overline{d_1 * \mathcal{S}_1 \to d_2 * \mathcal{S}_2} = d_1 \circ \overline{\mathcal{S}_1} \sqcup d_2 \circ \overline{\mathcal{S}_2}$$

4.4 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` $\to$ `Bool` type, and the variable x : `Int` and argument are irrelevant:

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

Definition 4.4 (Synthesis Slices). For a synthesising expression, $\Gamma \vdash e \Rightarrow \tau$. A synthesis slice is an expression typing slice ς^{γ} of e^{Γ} which also synthesises τ, that is, $[\![\gamma]\!] \vdash [\![\varsigma]\!] \Rightarrow \tau$.

Proposition 4.5. *A minimum synthesis slice of* $\Gamma \vdash e \Rightarrow \tau$, *under* $\sqsubseteq$, *exists.*

4.5 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 on the left, the *inner hole term* ? (underlined) must be consistent with `Int` due to the annotation and lambda constructor within its context, giving:

$$(\lambda x.\underline{?}) : \texttt{Bool} \to \texttt{Int} \qquad\qquad (\lambda\ x\ .\ \underline{?}\ ) : \texttt{Bool} \to \texttt{Int}$$

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

4.5.1 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. I refer to this as the *minimally scoped checking context*:

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

Definition 4.6 (Checking Context). If $\Gamma \vdash e \Leftarrow \tau$. Then, a checking context for e is a typing context d such that: $\text{ctx}(d) \neq \bigcirc$, and $\text{typ}(d)(\Gamma) \vdash \text{ctx}(d)\{e\} \Rightarrow \tau'$ for some τ' while still retaining the sub-derivation for $\Gamma \vdash e \Leftarrow \tau$.

Definition 4.7 (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.

Observant readers will notice that any expression has infinitely many checking contexts. But importantly, there are only finitely many checking contexts and *exactly one* minimally scoped checking context for a sub-expression which is itself (after substituting the sub-expression into its checking context) a sub-expression of a particular program.

Definition 4.8 (Analysis Slice). For $\Gamma \vdash e \Leftarrow \tau$ with a minimally scoped checking context d. An analysis slice is a context slice d' of d where $[\![d']\!]$ is also a checking context for e.

Proposition 4.9. *A minimum analysis slice of $\Gamma \vdash e \Leftarrow \tau$ in a checking context d, under $\sqsubseteq$, exists.*

Fig. 4 demonstrates an example of how this works for a more complex situation where function application enforces a type upon its argument.

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

(a) A function: synthesising $\texttt{Int} \to \texttt{Int}$.

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

(b) Its synthesis slice.

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

(c) The sub-slice relating *only* to the input part $\texttt{Int}$.

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

(d) The analysis slice of the function's argument ($\underline{1}$) when applied. Uses the synthesis sub-slice from (c).

Fig. 4. Demonstration: Analysis slice application uses synthesis slices

5 CAST SLICING THEORY

Cast slicing propagates type slice information during evaluation, by tagging casts types with type slices. The first two criteria work together during elaboration, inserted depending on if the cast component was derived from either type synthesis or analysis. The current rules are very involved, but build directly upon the Hazel calculus elaboration semantics. Future work will aim to simplify these rules.

Comparison with Blame: The idea of tagging information to casts is reminiscent of blame [19] in gradual typing. Blame determines whether a cast error is caused by the expression within a cast or the context around a cast, always blaming more dynamic code. Parallels between synthesis (of the expression) and analysis (of the context) slices can be seen, but these are actually orthogonal to blame, with type slicing concerning why casts are *inserted*. Future work could explore these connections and the integration of blame, with applications for type error debugging: we could point to exactly which portion of dynamic code (e.g. one particular static error inserted into a hole) is responsible for the dynamic error.

6 THE HAZEL IMPLEMENTATION

The implementation is under active development, can be found on the `witnesses-type-slicing` branch available at GitHub [2]. Hazel extends the core calculus with many advanced features including:[12] Lists, Tuples, Labelled Tuples (records) [15, ch. 11.7-8], Sum Types [15, ch. 11.10], Type

[12]As of July 2025

Aliases, Pattern Matching, Explicit System F Style Polymorphism [15, ch. 23], Iso-Recursive Types [15, ch. 22-23]. Type and cast slicing extends to these relatively simply, but, polymorphism and recursive types will require further (less trivial) extensions to the meta-theory.

7 FUTURE WORK

Future work aims to build upon these mathematical foundations in order to improve and explore the effectiveness of the human aspects of these highlighting systems.

Exploiting Decomposability in the UI:. The merits of this formal system stem from the ability to deconstruct slices by their type. Uses include refining the highlighting to explain exactly which code corresponds to a specific subpart of the expression's type. For example, a user may understand why an expression is a function, but not why it has a particular return type; conversely, they may only not understand why the expression is a function, but not care about the argument or return types themselves. We wish to further explore how to use this information to provide a more *intuitive and interactive UI* for use in the Hazel editor. This may include further refinements such as hiding, summarising, and jumping to the slices of variable definitions in an expression's type assumptions slice.

Polymorphism and Recursive Types: As previously mentioned, extending the meta-theory and implementation to parametric polymorphic systems and recursive types is planned.

Decomposable Type Eliminators: The slice of a function application will include the application and part of the slice of the function, all compressed within a single type constructor, and therefore not decomposable (see sub-figure (b) in figure 4). In the future, we wish to improve this situation, potentially by creating a direct correspondence between derivations and slices, allowing indexing on both derivations *and* types. Then slices could be decomposable according to the typing rules, for which an interactive UI could highlight code for each typing rule and even show/explain the rules in a pop-up similarly to the Explain This framework in Hazel [16]. This would require an entirely new, or at least major extension, to the presented theories in this paper.

Other ideas to deal with this which are purely to do with the UI, not requiring new formalisation, include emphasising[13] the fundamental sub-expression that 'sourced' the type (inside of the type eliminator), For example, emphasising the inner lambda in fig. 4 (b).

Cast Slicing: Understanding how and why a cast was manipulated throughout evaluation requires inspecting potentially long and complex evaluation traces. UI to simplify evaluation traces to focus only on specific casts would be of use here. Additionally, there is scope to develop *dynamic* slicing methods which would highlight minimal programs that evaluate to (a possibly less precise value) involving the same cast. These could be more akin to dynamic slicing in imperative languages [9], or in functional languages [14]. As Hazel can evaluate incomplete programs, the user would even be able to *run* this minimal program, and work only with the simpler resulting trace.

User Study: While these methods appear to have intuitive use in understanding type systems and type error debugging, the real-world effectiveness should be explored by user studies. A study on the methods effectiveness for the learning aspect, involving new users (e.g. students) would be feasible.

REFERENCES

[1] 2025. *Hazel Project Website*. https://hazel.org/
[2] 2025. *Hazel Source Code*. https://github.com/hazelgrove/hazel

[13] i.e. in a darker shade

[3] Michael D. Adams, Eric Griffis, Thomas J. Porter, Sundara Vishnu Satish, Eric Zhao, and Cyrus Omar. 2025. Grove: A Bidirectionally Typed Collaborative Structure Editor Calculus. *Proc. ACM Program. Lang.* 9, POPL, Article 73 (Jan. 2025), 29 pages. https://doi.org/10.1145/3704909

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

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

[6] Matteo Cimini and Jeremy G. Siek. 2016. 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, 443–455. https://doi.org/10.1145/2837614.2837632

[7] Jana Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming (ICFP'13)*. ACM. https://doi.org/10.1145/2500365.2500582

[8] Matthew Keenan and Cyrus Omar. 2024. Learner-Centered Design Criteria for Classroom Proof Assistants. In *Proceedings of HATRA*.

[9] Bogdan Korel and Janusz Laski. 1988. Dynamic program slicing. *Inform. Process. Lett.* 29, 3 (Oct. 1988), 155–163. https://doi.org/10.1016/0020-0190(88)90054-3

[10] Holbrook Mann MacNeille. 1937. Partially ordered sets. *Trans. Amer. Math. Soc.* 42, 3 (1937), 416–460.

[11] David Moon, Andrew Blinn, and Cyrus Omar. 2023. Gradual Structure Editing with Obligations. In *2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 71–81. https://doi.org/10.1109/VL-HCC57772.2023.00016

[12] Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. 2021. Filling typed holes with live GUIs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 511–525. https://doi.org/10.1145/3453483.3454059

[13] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live functional programming with typed holes. *Proceedings of the ACM on Programming Languages* 3, POPL (Jan. 2019), 1–32. https://doi.org/10.1145/3290327

[14] Roly Perera, Umut A. Acar, James Cheney, and Paul Blain Levy. 2012. Functional programs that explain their work. *ACM SIGPLAN Notices* 47, 9 (Sept. 2012), 365–376. https://doi.org/10.1145/2398856.2364579

[15] Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.

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

[17] Jeremy G. Siek and Walid Taha. 2006. *Gradual Typing for Functional Languages*. 81–92.

[18] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *Summit on Advances in Programming Languages*.

[19] Philip Wadler and Robert Bruce Findler. 2009. *Well-Typed Programs Can't Be Blamed*. Springer Berlin Heidelberg, 1–16. https://doi.org/10.1007/978-3-642-00590-9_1

[20] David HD Warren. 1978. *Applied logic: its use and implementation as a programming tool*. Ph. D. Dissertation. The University of Edinburgh.

[21] Baijun Wu and Sheng Chen. 2017. How type errors were fixed and what students did? *Proc. ACM Program. Lang.* 1, OOPSLA, Article 105 (Oct. 2017), 27 pages. https://doi.org/10.1145/3133929