Bidirectional Type Slicing

MAX CARROLL, ?University of Cambridge?, UK
ANIL MADHAVAPEDDY, University of Cambridge, UK
CYRUS OMAR, University of Michigan, USA

Development tools report *what* type an expression has, but not *why* it has that type. This paper develops a theory of *type slicing* that answers such questions: a programmer selects a term, queries any part of the type information associated with it, and receives a slice of the program—a well-formed partial program with irrelevant sub-terms folded away—that suffices to reproduce the queried type. We formulate type slicing for bidirectional type systems, where *synthesis slices* explain the type a term synthesises and *analysis slices* explain the type its surrounding context expects. The theory requires no cast dynamics as it applies to any bidirectional system equipped with a precision order on types and terms satisfying a downwards static graduality property. We develop the metatheory over a core calculus with holes, products, sums, and explicit polymorphism, based on the Hazelnut and marked lambda calculi, proving that every query has a minimal slice, and that refining a query monotonically shrinks its minimal slices. We then show how to calculate these slices both exactly and approximately. Finally, integrating type slicing with error marking theory extends these results to arbitrary ill-typed programs, so a single mechanism explains both types and type errors in *complete*, *incomplete*, and *erroneous* code. The metatheory is mechanised in Agda, and a linear-time approximation of type slicing is implemented for the Hazel programming environment.

1 Introduction

Programming often involves reasoning about static types. Development tools assist programmers in limited ways with this task, e.g. by reporting the type of an expression when hovering over it, or reporting the expected and actual type when there is a type error. However, these services only report *what* the type of an expression is, not *why* it is that type in the context of the surrounding program. Programmers must hunt through the program themselves to answer these questions, and the program often contains a large volume of code irrelevant to their query.

This paper develops the theory of *type slicing* from type-theoretic first principles. Type slices allow developers to ask "why questions" [23] about type information reported about arbitrary terms within a (possibly incomplete or erroneous) program. The answers are given in the form of a minimal slice of the original program (i.e. a program with unrelated terms folded away) that suffices to reproduce the queried type. Type slicing is inspired in principle by prior work on program slicing (§11), in particular, lattice-based *Galois slicing* [3, 34–36, 39]. However, slicing is here applied to a (non-Galois) static type checking setting, loosely more akin to type-error slicing [19, 47, 50], and the concurrent independently researched work on *type-directed slicing* [27] for Java.

We consider the problem of type slicing in the context of *bidirectional type systems* [17], which distinguish *type synthesis* (which type does this term have) from *type analysis* (which type is expected). Correspondingly, a *synthesis slice* (§4) of a selected term answers "which parts of the program caused it to have type τ?" and an *analysis slice* (§6) answers the complementary question, "which parts of the surrounding program enforced that this term have type τ?" Formalising analysis slices requires a novel *context typing* construction (§5), which types the one-hole context surrounding a focused sub-term. Queries can be *refined*, say from an entire function type to just its codomain, resulting in a smaller slice. This allows a programmer to incrementally focus on sub-parts of the type that are of interest, for example, determining that only a small region of a large record type involved in a type error is erroneous.

Authors' Contact Information: Max Carroll, ?University of Cambridge? Department of Computer Science and Technology, Cambridge, UK; Anil Madhavapeddy, University of Cambridge, Department of Computer Science and Technology, Cambridge, UK; Cyrus Omar, University of Michigan, Ann Arbor, USA.

Type Error Debugging. The type slicing machinery applies equally well to well-typed and ill-typed terms. In the case of ill-typed terms, it provides a mechanism that may aid in debugging type errors, subsuming the question of "why is this term a type error?" This is achieved by integrating type slicing with the theory of error marking (§7). A distinguishing feature of type slicing is that a slice is itself a valid incomplete program that preserves the original's shape; in Hazel [1], which allows incomplete programs, this is simply a valid program. Type slicing therefore *composes* with most other type debugging methods, as we can simply run them on the minimised programs.

For example, error *localisation* [32, 54] and *counter-factual* [13, 14, 25] type debugging techniques (§11.5) are equally applicable, and in fact simpler, on minimised programs, when their language assumptions (typically global inference) apply.

Further, editing the 'type' of an expression as suggested by such tools is not necessarily easy when the surrounding context expects a different type. The *analysis slice* points to the surrounding terms (typically annotations) that must be modified to make such an edit, likewise easing refactoring to a different type in general.

1.1 Contributions

The theory applies to *bidirectional type systems* with any form of *downwards static graduality* on expressions (§2), inspired by the gradual typing literature [44] extended to incomplete programs, i.e. programs with holes (and marked errors). No cast dynamics are required, so type slicing applies to any bidirectional system upon assigning a precision relation on types and terms (§3.2).

Concretely, this paper contributes: *(i)* a core calculus with holes, products, sums, and explicit polymorphism, based on the Hazelnut and marked lambda calculi [29, 56], with precision lattices satisfying graduality (§3); *(ii) synthesis slices* (§4), whose minimal slices exist for every query, refine monotonically, and decompose compositionally; *(iii) context typing* (§5), a novel judgement with totality, soundness, and gradual guarantees, enabling the dual *analysis slices* (§6); *(iv)* integration with error marking (§7), extending both slice forms to arbitrary ill-typed programs; *(v)* term-minimal slices supporting compositional calculation, NP-hardness of minimum-size slicing, and a linear-time approximation (§8–§10); and *(vi)* a comparison with prior forms of slicing and type debugging, and applications to set-theoretic and occurrence typing (§11).

Hazel [1] is an example full-scale language satisfying these conditions; we have extended its implementation with an efficient linear-time approximation of type slicing, introduced by example (§1.2). The prototype user interface is an initial proof of concept, and we make no specific usability claims; we expect the theory to support a variety of user interfaces and further applications for type debugging and explanation, and suggest several avenues for exploration (§12). The core calculus and metatheory are mechanised in Agda and available in the supplemental material.

1.2 Type Slicing in Hazel by Example

This section previews type slicing, formally developed later (§4, §6) over the core calculus (§3), through two worked examples in the Hazel implementation. The Hazel UI has a cursor inspector at the bottom of the screen which details which type the selected term *has*, and what it *expects*; these are the respective types to calculate a synthesis slice and an analysis slice for.

Hazel renders each slice by folding away every fragment the queried type does not require. The user can refine their slice by themselves folding away part of the type being sliced, which will correspondingly fold away additional fragments of the program.

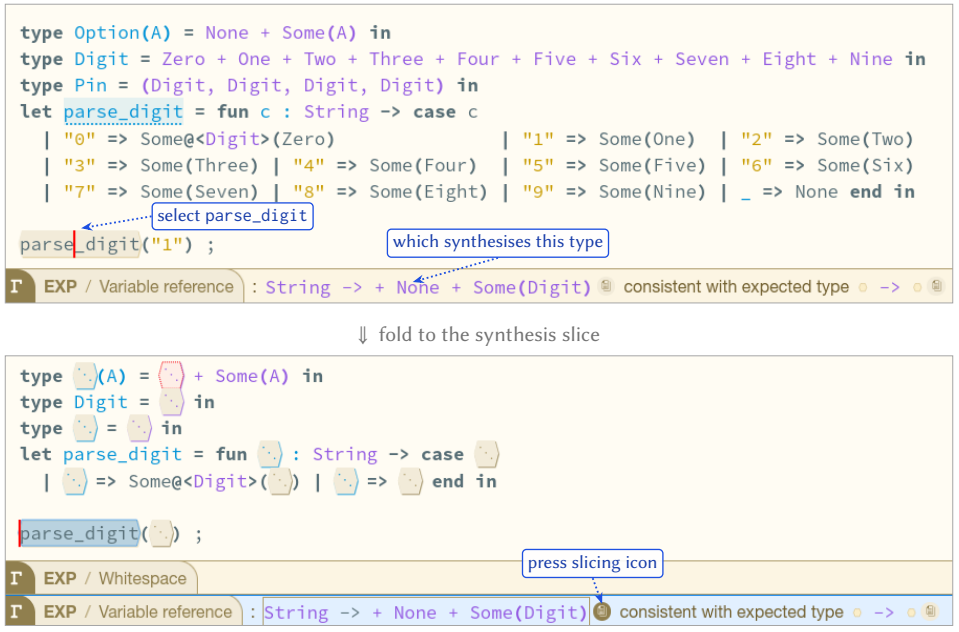
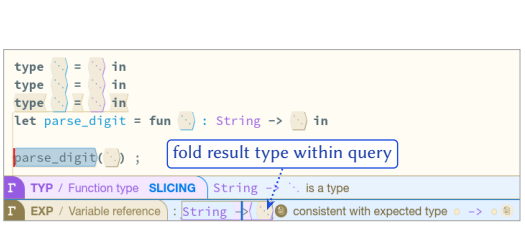


Fig. 1. Synthesis slice for `parse_digit`.

Figure 1 slices a small character parser. The synthesis slice reduces the function to the `String` domain annotation and the first branch determining the result option type, discarding all other branches.[1]

Refined queries may target *part* of a reported type: for example, asking only about the domain of a function type, by folding away the returned sum type (`String → □`). The resulting program shrinks the slice to exclude the entire function body (Fig. 2). This same slice is also the minimal *analysis* slice which *enforces* `String` on the argument.



(a) Refined synthesis query omitting the return Option type.

(b) Analysis slice of the `String` expectation on the argument of `parse_digit`.

Fig. 2. Refined synthesis and analysis slices.

Figure 3 considers a much more complex error where a sequencing parser combinator has been defined incorrectly, but used correctly. As is usual, the error has been incorrectly localised to the

[1]Folding all other branches into a single fold projector is not currently supported by the Hazel UI, but is shown here for illustrative purposes.

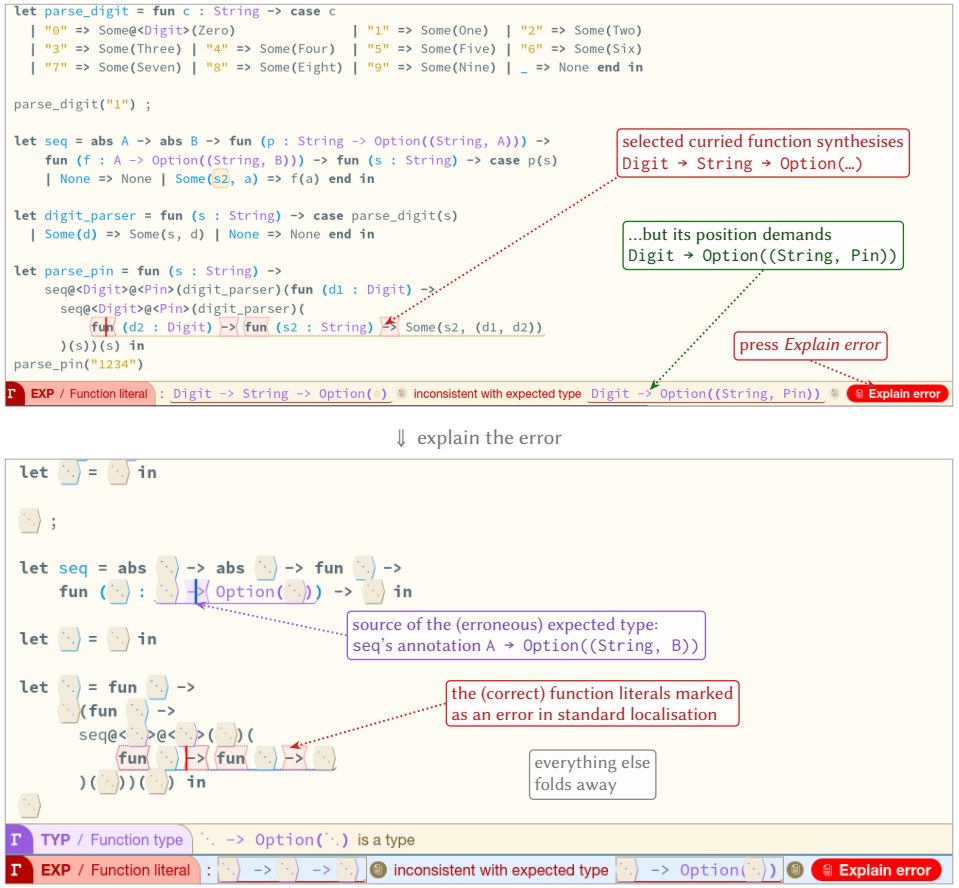


Fig. 3. Explain-error slice for the parser combinator.

usage site, trusting the (incorrect) annotation. Specifically, `seq`'s continuation is annotated as if it returns an `Option` immediately, rather than first accepting the remaining `String` threaded out of the previous parser. The use site provides the intended parser continuation.

Explain error queries the slices on only the outermost inconsistency in the types (the function vs the option in the return type); the resulting slice collapses the program to just the function shape of the continuation and the offending annotation.

Crucially, the slice did not assume the annotation is the ground truth. It shows the user both the expectation's *source* (the annotation) and the *usage* (the function), leaving the programmer to judge which is wrong. This is justified as programmers often add annotations incorrectly to existing code (49% of the time [30]). Should the user instead wish to trust the context unilaterally, they can simply only consider the synthesis slice (omitting the analysis slice entirely). We later make slicing of type errors precise, applying slicing to marked ill-typed programs (§7).

2 Language Scope

This section introduces the key characteristics of the languages to which type slicing can be applied.

2.1 Bidirectional Types

Type slicing is formalised for *bidirectional type systems* [17]. Such a type system is an algorithmic specification of typing judgements, naturally lending itself to direct typing algorithms which do not require *guessing* of types during type checking. These systems do require annotations, but annotation burden is lessened by locally inferring types [37].

This is achieved 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):

$$\Delta;\ \Gamma \vdash e \Rightarrow \tau$$

Read as: *e synthesises type* τ *under typing assumptions* Γ *and type variable assumptions* Δ. The type τ is an *output*.

$$\Delta;\ \Gamma \vdash e \Leftarrow \tau$$

Read as: *e analyses against type* τ *under typing assumptions* Δ, Γ. The type τ is an *input*. This is where the terminology of *synthesis / analysis* slices is derived from.

Such languages have two fundamental rules:

$$\textsc{SVar}\ \frac{x : \tau \in \Gamma}{\Delta;\ \Gamma \vdash x \Rightarrow \tau} \qquad \textsc{Sub}\ \frac{\Delta;\ \Gamma \vdash e \Rightarrow \tau}{\Delta;\ \Gamma \vdash e \Leftarrow \tau}$$

Variables synthesise their type from the typing assumptions. Subsumption bridges the two modes: any term that can synthesise a type also analyses against that same type.

2.2 Downwards Static Graduality

To calculate a type slice we will need a way to *gradually* omit sub-expressions from the program, and type check these less *'precise'* partial programs at less or equally 'precise' types. Formally, precision must form two partial orders (notated $\sqsubseteq$) over types and expressions, which can be lifted onto typing assumptions Γ.

Definition 2.1. A bidirectional calculus satisfies *downwards static graduality* if, for all $\Delta, \Gamma', \Gamma, e', e, \tau$ such that $\Gamma' \sqsubseteq \Gamma$ and $e' \sqsubseteq e$, both:

- If $\Delta;\ \Gamma \vdash e \Rightarrow \tau$ then there exists $\tau' \sqsubseteq \tau$ such that $\Delta;\ \Gamma' \vdash e' \Rightarrow \tau'$.
- For all $\tau' \sqsubseteq \tau$, if $\Delta;\ \Gamma \vdash e \Leftarrow \tau$ then $\Delta; \Gamma' \vdash e' \Leftarrow \tau'$.

For example, this precision order can correspond to the precision relation on types in a gradually typed system [43]. In this situation this graduality theorem corresponds to the downwards static case[2] of the graduality theorem in the refined criteria for gradual types [44], adapted to a bidirectional setting. The calculus we demonstrate type slicing on in this paper goes even further by allowing maximally incomplete programs, omitting any program sub-expression, based upon the Hazel calculus [28].

Although not necessary, it is also useful that precision forms a lattice with meets and joins to perform optimisations to type slicing calculation (§8).

3 The Core Calculus

In this section, we present the core calculus which type slicing builds upon. It is a bidirectional, gradually typed lambda calculus, with precision on types and terms, based on the Hazel calculus [28], extended with products, sums, and explicit System F polymorphism. We prove that this calculus satisfies graduality (§2.2), and we define and prove several lattice structures on precision.

[2]Hence the name.

3.1 Syntax & Relations

The syntax of the core calculus is given in Figure 4. 1 is the unit type, and $\square$ is a gap / hole serving a dual role: in the gradual typing interpretation it is the dynamic type, and in the slicing interpretation it represents unrequired type information (to explain a query). Similarly a $\square$ in expressions represents omitted sub-expressions, irrelevant to the explanation resulting from a query. In type checking, omitted expressions synthesise the dynamic (omitted) type.

$$\boxed{\text{Syntax}} \quad \text{Types } \tau \text{ and expressions } e$$

$$\tau ::= \square \mid 1 \mid \tau \to \tau \mid \tau \times \tau \mid \tau + \tau \mid \forall \alpha.\ \tau \mid \alpha$$

$$e ::= \square \mid () \mid x \mid \lambda x : \tau.\ e \mid \lambda x.\ e \mid e(e) \mid (e,\ e) \mid \pi_1\ e \mid \pi_2\ e$$
$$\mid\ \iota_1\ e \mid \iota_2\ e \mid (\textbf{case}\ e\ \textbf{of}\ x.\ e \mid y.\ e) \mid \Lambda\alpha.\ e \mid e\langle\tau\rangle \mid \textbf{let}\ x = e\ \textbf{in}\ e$$

Fig. 4. Syntax of the core calculus up to α-equivalence (on variables x).

Base types such as booleans are omitted from the core syntax, being encoded using sums in the usual way (e.g. $\mathit{Bool} \triangleq 1 + 1$); we use them freely in examples.

3.1.1 Consistency. Type consistency (Figure 5) is exactly as presented in standard gradual type systems (§2.2). That is, a weakening of equality, where every type is consistent with $\square$, and compound types are consistent when their components are. Consistency is reflexive and symmetric but *not* transitive.

$$\boxed{\tau_1 \sim \tau_2} \quad \tau_1 \text{ is consistent with } \tau_2$$

$$\sim\square_1\ \frac{}{\tau \sim \square} \qquad \sim\square_2\ \frac{}{\square \sim \tau} \qquad \sim 1\ \frac{}{1 \sim 1} \qquad \sim\to\ \frac{\tau_1 \sim \tau_1' \qquad \tau_2 \sim \tau_2'}{\tau_1 \to \tau_2 \sim \tau_1' \to \tau_2'}$$

$$\sim\times\ \frac{\tau_1 \sim \tau_1' \qquad \tau_2 \sim \tau_2'}{\tau_1 \times \tau_2 \sim \tau_1' \times \tau_2'} \qquad \sim + \frac{\tau_1 \sim \tau_1' \qquad \tau_2 \sim \tau_2'}{\tau_1 + \tau_2 \sim \tau_1' + \tau_2'} \qquad \sim\forall\ \frac{\tau \sim \tau'}{\forall\alpha.\ \tau \sim \forall\alpha.\ \tau'}$$

Fig. 5. Type consistency.

3.1.2 Precision. Precision (Figure 6) is a partial order that organises terms by *information content*: for types, $\tau' \sqsubseteq \tau$ means τ has at least as much type information as τ'. Here, $\square$ is the global minimum. Expression precision is defined analogously; for instance, the slices of $\lambda x : 1.\ x$ form the precision chain $\square \sqsubseteq \lambda x : \square.\ \square \sqsubseteq \lambda x : 1.\ \square \sqsubseteq \lambda x : 1.\ x$, each step filling in one more part of the term.

$$\boxed{\tau \sqsubseteq \tau'} \quad \tau \text{ is less precise than } \tau'$$

$$\sqsubseteq\square\ \frac{}{\square \sqsubseteq \tau} \qquad \sqsubseteq 1\ \frac{}{1 \sqsubseteq 1} \qquad \sqsubseteq\alpha\ \frac{}{\alpha \sqsubseteq \alpha} \qquad \sqsubseteq\to\ \frac{\tau_1 \sqsubseteq \tau_1' \qquad \tau_2 \sqsubseteq \tau_2'}{\tau_1 \to \tau_2 \sqsubseteq \tau_1' \to \tau_2'}$$

$$\sqsubseteq\times\ \frac{\tau_1 \sqsubseteq \tau_1' \qquad \tau_2 \sqsubseteq \tau_2'}{\tau_1 \times \tau_2 \sqsubseteq \tau_1' \times \tau_2'} \qquad \sqsubseteq + \frac{\tau_1 \sqsubseteq \tau_1' \qquad \tau_2 \sqsubseteq \tau_2'}{\tau_1 + \tau_2 \sqsubseteq \tau_1' + \tau_2'} \qquad \sqsubseteq\forall\ \frac{\tau \sqsubseteq \tau'}{\forall\alpha.\ \tau \sqsubseteq \forall\alpha.\ \tau'}$$

Fig. 6. Type precision.

Precision on Typing Assumptions. In this calculus, we represent typing assumptions as *finite* partial functions from variable names to types, with domain $\mathrm{dom}(\Gamma)$. Precision is defined by domain inclusion and pointwise precision on types in the (smaller) domain:

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

That is, a less precise type assumptions set either assumes fewer variables' types, or assumes less precise types on the present variables.

Remark 3.1. *Typical literature refers to these typing assumptions as 'typing contexts'. We avoid this terminology as we use the context to refer to the syntactic context around a subterm, i.e. in the same sense as used in contextual dynamics (Harper [20, Chapter 5]).*

3.2 Lattice Properties

3.2.1 Meets and Joins. The *meet* $\tau_1 \sqcap \tau_2$ is the greatest lower bound (GLB) under precision, retaining only sub-terms present in both terms. The *join* $\tau_1 \sqcup \tau_2$ is the least upper bound (LUB) under precision, omitting only subterms omitted in both components. For example:

$$(1 \to \square) \sqcap (\square \to 1) \;=\; \square \to \square, \qquad (1 \to \square) \sqcup (\square \to 1) \;=\; 1 \to 1.$$

Meets are *total*, but joins are defined only between *consistent* types.

3.2.2 The Slice Lattice. For a fixed type τ, the *slice lattice* of τ is the *lower set* under precision:

$$\lfloor \tau \rfloor \;=\; \{ v \mid v \sqsubseteq \tau \}$$

$\lfloor e \rfloor$, $\lfloor \mathcal{C} \rfloor$, and $\lfloor \Gamma \rfloor$ are defined analogously for expressions, contexts, and assumptions. Each slice lattice has top $\top = \tau$ and bottom $\bot = \square$. Then precision, meets, and joins simply lift unchanged into the slice lattice. Importantly all terms in the slice lattice are consistent, so joins are always defined:

Lemma 3.2 (Slices are Mutually Consistent). *If $v_1 \sqsubseteq \tau$ and $v_2 \sqsubseteq \tau$, then $v_1 \sim v_2$.*

Hence, all terms in the slice lattice are consistent, making it a well-defined *bounded lattice*. The meet and join are also distributive. Further, slice lattices are trivially finite:

Theorem 3.3. *Each slice lattice $\lfloor \tau \rfloor$, $\lfloor e \rfloor$, $\lfloor \mathcal{C} \rfloor$, and $\lfloor \Gamma \rfloor$ is finite.*

Theorem 3.4. *Each slice lattice $\lfloor \tau \rfloor$, $\lfloor e \rfloor$, $\lfloor \mathcal{C} \rfloor$, and $\lfloor \Gamma \rfloor$ is a bounded distributive lattice.*

Moreover, being finite and distributive, each slice lattice is bi-Heyting [15]. We will later introduce and use co-Heyting subtraction to represent 'disjoint' types (§8).

Notation: Slice Lattices as Highlighted Terms. An element of a slice lattice can be notated as a highlight of the original (top) term with omitted regions (the $\square$s) left unhighlighted. For instance, the slice $\mathit{Int} \to \square$ in $\lfloor \mathit{Int} \to \mathit{Bool} \rfloor$ can be represented as $\mathit{Int}\to\ \mathit{Bool}$.

3.3 Typing Rules

The bidirectional typing judgements are mostly standard; the complete synthesis and analysis rule sets appear in the supplementary material, and Figure 7 presents some of the major rules. $\Delta \vdash \tau\ \mathsf{wf}$ denotes when a type τ is *well-formed* under available type variables Δ.

We use joins to replicate type matching in the gradual typing literature.[3] $\tau \equiv \tau_1 \to \tau_2$ extracts the domain and codomain of τ, or the gap type $\square$ if $\tau \equiv \square$. Case expressions can synthesise a type when their branches synthesise *consistent* types. There is a dedicated rule for unannotated let bindings, which differ from function applications in a bidirectional setting due to type information flowing from the binding into the body. The typing rules satisfy graduality (§2.2).

[3] Typically notated $\tau \blacktriangleright \tau_1 \to \tau_2$.

$$\Rightarrow\mathsf{Var}\ \frac{x : \tau \in \Gamma}{\Delta; \Gamma \vdash x \Rightarrow \tau} \qquad \Rightarrow\lambda\mathsf{Ann}\ \frac{\Delta \vdash \tau_1\ \mathsf{wf} \quad \Delta; \Gamma, x : \tau_1 \vdash e \Rightarrow \tau_2}{\Delta; \Gamma \vdash \lambda x : \tau_1.\, e \Rightarrow \tau_1 \to \tau_2}$$

$$\Rightarrow\mathsf{let}\ \frac{\Delta; \Gamma \vdash e_1 \Rightarrow \tau_1 \quad \Delta; \Gamma, x : \tau_1 \vdash e_2 \Rightarrow \tau_2}{\Delta; \Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \Rightarrow \tau_2} \qquad \Rightarrow\Lambda\ \frac{\Delta, \alpha; \Gamma \vdash e \Rightarrow \tau}{\Delta; \Gamma \vdash \Lambda\alpha.\, e \Rightarrow \forall\alpha.\, \tau}$$

$$\Rightarrow\mathsf{App}\ \frac{\Delta; \Gamma \vdash e_1 \Rightarrow \tau \quad \tau \sqcup (\square \to \square) \equiv \tau_1 \to \tau_2 \quad \Delta; \Gamma \vdash e_2 \Leftarrow \tau_1}{\Delta; \Gamma \vdash e_1(e_2) \Rightarrow \tau_2}$$

$$\Rightarrow\mathsf{TApp}\ \frac{\Delta \vdash \sigma\ \mathsf{wf} \quad \Delta; \Gamma \vdash e \Rightarrow \tau \quad \tau \sqcup \forall\alpha.\, \square \equiv \forall\alpha.\, \tau'}{\Delta; \Gamma \vdash e\langle\sigma\rangle \Rightarrow [\alpha \mapsto \sigma]\,\tau'}$$

$$\Rightarrow\mathsf{case}\ \frac{\begin{array}{c}\Delta; \Gamma \vdash e \Rightarrow \tau \quad \tau \sqcup (\square + \square) \equiv \tau_1 + \tau_2 \quad \tau_1' \sim \tau_2' \\ \Delta; \Gamma, x : \tau_1 \vdash e_1 \Rightarrow \tau_1' \qquad \Delta; \Gamma, y : \tau_2 \vdash e_2 \Rightarrow \tau_2'\end{array}}{\Delta; \Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\ x.\, e_1 \mid y.\, e_2 \Rightarrow \tau_1' \sqcup \tau_2'}$$

Fig. 7. Selected synthesis rules.

Theorem 3.5. *The synthesis judgement* $\_ \vdash \_ \Rightarrow \_$ *and analysis judgement* $\_ \vdash \_ \Leftarrow \_$ *satisfy downwards static graduality.*

4 Synthesis Slices

The type information of a subexpression in a bidirectional type system is derived either from *within* the term by synthesis, or from the surrounding context. This section defines *synthesis slices* which concern how to explain an expression's *synthesised* type, that is, the type information *internal* to the expression.

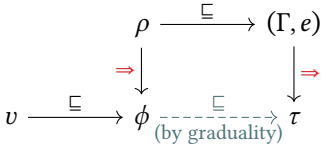
4.1 Synthesis Slices

Consider a 'program' $P = (\Gamma, e)$, that is, typing assumptions and an expression. When the program synthesises a type (in some type variable context Δ)[4], all of its type information is contained within the synthesis derivation $D :: \Delta; \Gamma \vdash e \Rightarrow \tau$.

Given the derivation D, we may *query* the type τ by taking a slice υ in $\lfloor\tau\rfloor$. A synthesis slice is a 'program slice' $\rho = (\gamma, \varsigma)$ in $\lfloor\Gamma, e\rfloor$ which synthesises a type at least as precise as the query υ; this corresponds to a highlighted version of the original program.

Intuitively, υ specifies the part of τ we wish to explain; for example, querying $\upsilon = \mathit{Int} \to \square$ for $\tau = \mathit{Int} \to \mathit{Bool}$ asks "which parts of the program account for the domain being *Int*?". The slice might not include *all* the program regions relevant to the query type, but will necessarily provide a consistent subset which suffices to totally *explain* (independently synthesise) the query.

Definition 4.1 (Synthesis slice). A *synthesis slice* of $D :: \Delta; \Gamma \vdash e \Rightarrow \tau$ on $\upsilon \in \lfloor\tau\rfloor$, written $\mathit{SynSlice}\ D \blacktriangleleft \upsilon$, is a pair $(\gamma, \varsigma) \in \lfloor\Gamma, e\rfloor$ such that $\Delta; \gamma \vdash \varsigma \Rightarrow \phi$ for some $\phi \sqsupseteq \upsilon$ (with $\phi \in \lfloor\tau\rfloor$ by graduality, Theorem 3.5).



Many such synthesis slices exist. In particular, the original program (Γ, e) is necessarily a synthesis slice; and for $\mathit{SynSlice}\ D \blacktriangleleft \square$, every slice of (Γ, e) is a valid synthesis slice. We refer to the $\upsilon \sqsubseteq \phi$ condition as *validity*. It is intentional that this is *not* an exact equality, as demonstrated by the counterexample below:

[4]Type variable contexts are not sliced as the only information such slices would encode is whether a type variable was used, which can be extracted purely syntactically from the resulting slice. Typing assumptions are different because we can *partially* require an assumption.

Counterexample 4.2 (Non-Existence of all Exact Synthesis Slices).
Consider $x : \mathbf{1} \to \mathbf{1} \vdash (x, x) \Rightarrow (1 \to 1) \times (1 \to 1)$, *and query* $v = (\mathbf{1} \to \square) \times (\square \to \mathbf{1})$.
Any strictly smaller slice of either $x : 1 \to 1$ *or* (x, x) *synthesises a type strictly less precise than* v. *Yet the original program synthesises* $(1 \to 1) \times (1 \to 1)$, *strictly more precise than* v.

4.2 Minimality

Synthesis slices provide *some* explanation of the query, but many are clearly useless, such as the original program itself, which corresponds to fully highlighting the program. We want to find minimal slices: those where further slicing any part of the program makes the slice invalid (synthesising a type insufficient to cover the query).

Definition 4.3 (Minimality). For an element a of a poset, a is minimal, $\mathsf{IsMinimal}(a)$, when every a' with $a' \sqsubseteq a$ is in fact equal to a.

Definition 4.4 (Minimal Synthesis Slices). A *MinSynSlice* $D \blacktriangleleft v$ is a *SynSlice* $D \blacktriangleleft v$ minimal in *SynSlice* $D \blacktriangleleft v$ under program-slice precision on $\lfloor \Gamma, e \rfloor$.

4.3 Existence of Minimal Slices

Crucially, any synthesis slice has a minimal slice below it (or is minimal):

Theorem 4.5 (Existence of minimal slices). *For every* s : *SynSlice* $D \blacktriangleleft v$, *there exists* m : *MinSynSlice* $D \blacktriangleleft v$ *with* $m \sqsubseteq s$.

Proof. The slice lattice $\lfloor \Gamma, e \rfloor$ is finite. Hence, $\{ s' : \textit{SynSlice } D \blacktriangleleft v \mid s' \sqsubset s \}$ is finite and computable by enumeration. If this set is empty, then s is minimal; otherwise, recurse on any of its elements. Each step strictly decreases precision, so finiteness ensures termination. □

Consequently, any derivation D has a minimal synthesis slice, as the original program is a trivial inhabitant of *SynSlice* $D \blacktriangleleft v$ for all v.

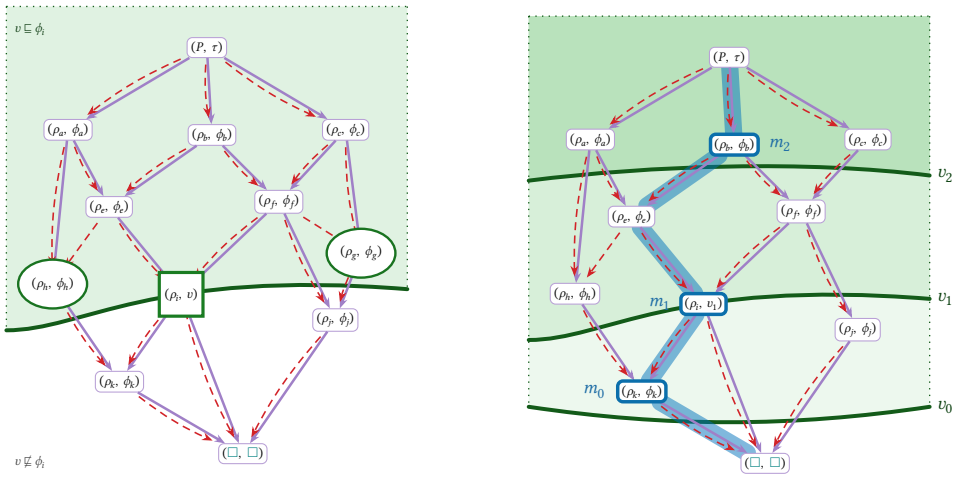
A Brute-Force Algorithm via Graduality. The proof above is not a practical algorithm. In practice, for any term, the list of maximal slices strictly less precise than a given term can be computed efficiently by omitting exactly *one* leaf from the AST.

The algorithm picks the first maximal strict slice that satisfies the query and recurses; if none of the slices satisfy the query, then a minimal slice has been found, by graduality (Theorem 3.5). This is $\mathcal{O}(n \times T)$ where n is the program size and T the type checking complexity; type checking is linear in the program size for this calculus, so finding a minimal slice is $\mathcal{O}(n^2)$.

See Figure 8a for an example lattice to descend through, stopping just before descending below the given query; all three minima are marked.

4.4 Refined Slices

In practice, the user may look at a large query v getting a large program slice ρ. Therefore they may want to refine the query to focus on a subpart, $v' \sqsubseteq v$, that they do not quite understand. It is always possible to pick a consistent program slice $\rho' \sqsubseteq \rho$ which is a refinement of the previous slice by *monotonicity* (which follows from graduality, Theorem 3.5):



(a) Synthesis slices above query v in the shaded region, with 3 minima consisting of 2 inexact minima (ellipses) and one minimum (square) synthesising exactly v.

(b) Monotonically descending minima m_2, m_1, m_0 for query refinements $v_2 \to v_1 \to v_0$, calculated by descending along the highlighted path.

Fig. 8. Two similar Hasse diagrams of program slices $\rho_i \in \lfloor P \rfloor$, ordered by strict program precision $\sqsubset$ (solid arrows), paired with corresponding synthesised types $\phi_i \in \lfloor \tau \rfloor$. Overlaid with precision relations on types (dashed arrows).

Theorem 4.6 (Monotonicity of minimal slices). *If* $v_1 \sqsubseteq v_2$ *in* $\lfloor \tau \rfloor$ *and* m_2 : *MinSynSlice* $D \blacktriangleleft v_2$, *then there exists* m_1 : *MinSynSlice* $D \blacktriangleleft v_1$ *with* $m_1 \sqsubseteq m_2$.

This is desirable because the user will not be shown *additional* information after a query refinement which would require further reasoning by the user to understand. Figure 8b illustrates a monotonic refinement path along 3 refined queries on the example lattice. If we did not follow a monotonic path during refinement we would get 'branch hopping':

Branch hopping. This is where highlighted regions switch between branches when refining the query, requiring the user to switch their reasoning between the contents in each branch. For example, refining $v = \mathbf{1} \to \mathbf{1}$ to $v' = \square \to \mathbf{1}$:

assumptions *assumptions*

$$\boxed{x : \mathbf{1} \to \mathbf{1}} \ \mathbf{if}\ \mathbf{true}\ \mathbf{then}\ x\ \mathbf{else}\ y \xrightarrow{v' \sqsubseteq v} \boxed{y : \mathbf{1} \to \mathbf{1}} \ \mathbf{if}\ \mathbf{true}\ \mathbf{then}\ x\ \mathbf{else}\ y$$

On the other hand, monotonicity does not necessarily hold in the upwards direction. Consider the ρ_j node in Figure 8b, which is minimal under query v_0, but no minimum for query v_1 is strictly more precise than ρ_j.

4.5 Decompositions

Instead of calculating slices by brute force (§4.3), it would be ideal if we could compositionally calculate minimal slices of a term from minimal subslices of its subterms. It is indeed possible to compose synthesis slices into larger synthesis slices, and we prove that all minima can be expressed by some composition; this subsection demonstrates two of the cases. We later consider how to recursively choose the compositions (§8).

4.5.1 Products. Given slices $(\gamma_1, \varsigma_1) : \mathit{SynSlice}\ D_1 \blacktriangleleft v_1$ and $(\gamma_2, \varsigma_2) : \mathit{SynSlice}\ D_2 \blacktriangleleft v_2$ for the two components of a pair $D = \Rightarrow\texttt{Pair}\ D_1\ D_2$ (with synthesised types ϕ_1, ϕ_2), we form their product slice $(\gamma_1, \varsigma_1) \times (\gamma_2, \varsigma_2) : \mathit{SynSlice}\ D \blacktriangleleft v_1 \times v_2$ by joining the two assumption slices and pairing the term slices:

$$(\gamma_1, \varsigma_1) \times (\gamma_2, \varsigma_2) \triangleq (\gamma_1 \sqcup \gamma_2,\ (\varsigma_1, \varsigma_2))$$

The pair synthesises (possibly more precise)[5] types $\phi_1' \times \phi_2' \sqsupseteq \phi_1 \times \phi_2 \sqsupseteq v_1 \times v_2$. For this combinator, we get the following decomposition theorem:

Theorem 4.6.1 (*Product Decomposition*). *Let $m : \mathit{MinSynSlice}\ \Rightarrow\texttt{Pair}\ D_1\ D_2 \blacktriangleleft v_1 \times v_2$. Then there exist minimal $m_1 : \mathit{MinSynSlice}\ D_1 \blacktriangleleft v_1$ and $m_2 : \mathit{MinSynSlice}\ D_2 \blacktriangleleft v_2$ with $m = m_1 \times m_2$.*

However, we shall later see that the converse fails: the product of two minimal slices need not be minimal, since the joined assumptions can leave one component's term slice redundant. So one cannot simply recurse through the derivation, pairing sub-slices on v_1 and v_2 to obtain a minimal slice on $v_1 \times v_2$. We explore and resolve this problem later (§8.1).

4.5.2 Let Bindings. A more subtle case is that of let-bindings, which have different assumptions for their two premises:

$$\Rightarrow\texttt{Let}\ \frac{\Delta;\ \Gamma \vdash e_1 \Rightarrow \tau_1 \qquad \Delta;\ \Gamma, x : \tau_1 \vdash e_2 \Rightarrow \tau_2}{\Delta;\ \Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \Rightarrow \tau_2}$$

A slice of a let-expression can be constructed from a slice (γ_1, ς_1) of the definition (synthesising ϕ_1) and a slice (γ_2, ς_2) of the body, constrained with an assumption consistency condition: the body's used assumptions for the bound variable x must be *at most as precise as* the type synthesised by the slice of the definition, $\gamma_2(x) \sqsubseteq \phi_1$.

Then, upon joining the assumptions for the two slices, we must remove the assumption on x from the body (notated $\gamma_{\setminus x}$), as this is now provided by the definition:

$$(\gamma_1, \varsigma_1)\ \mathsf{def}\ (\gamma_2, \varsigma_2) \triangleq (\gamma_1 \sqcup (\gamma_2)_{\setminus x},\ \mathbf{let}\ x = \varsigma_1\ \mathbf{in}\ \varsigma_2)$$

Theorem 4.6.2 (*Binding Decomposition*). *Let $m : \mathit{MinSynSlice}\ \Rightarrow\texttt{Let}\ D_1\ D_2 \blacktriangleleft v$. Then there exists $\phi_1 \in \lfloor \tau_1 \rfloor$ and minimal slices $m_1 : \mathit{MinSynSlice}\ D_1 \blacktriangleleft \phi_1$ (synthesising ϕ_1') and $m_2 = (\gamma_2, \varsigma_2) : \mathit{MinSynSlice}\ D_2 \blacktriangleleft v$ satisfying $\gamma_2(x) \sqsubseteq \phi_1'$, with $m = m_1\ \mathsf{def}\ m_2$.*

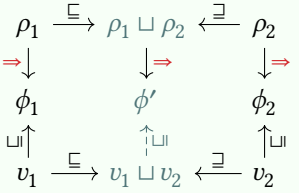
4.6 Contribution Slices

Finally, a minimal slice only provides an explanation; it does not show *all* information *contributing* in any way to a query type. The latter is a useful notion when performing refactors which change the type, where quickly identifying the regions of the program which rely on the old type is crucial.

Such a region can be uniquely expressed as the *join* of all minimal slices for the query. The contribution slice is efficient in the sense that it is a *least* upper bound of actual minimal slices. For this to work, synthesis slices must be closed under joins: the joined slice must also be *valid*:

[5] Due to $\gamma_1 \sqcup \gamma_2$ being larger than γ_1 and γ_2.

Theorem 4.7 (Join Closure of Synthesis Slices). *For $\rho_1 = (\gamma_1, \varsigma_1) : \mathit{SynSlice}\ D \blacktriangleleft \upsilon_1$ and $\rho_2 = (\gamma_2, \varsigma_2) : \mathit{SynSlice}\ D \blacktriangleleft \upsilon_2$, then $\rho_1 \sqcup \rho_2 : \mathit{SynSlice}\ D \blacktriangleleft \upsilon_1 \sqcup \upsilon_2$, with the joined slice taken pointwise: $(\gamma_1 \sqcup \gamma_2,\ \varsigma_1 \sqcup \varsigma_2)$.*



5 Context Typing

Context typing is a novel construction for bidirectional type systems. It is inspired by one-hole contexts (zippers) of algebraic data types [22], decomposing a data structure into a focused element and its surrounding context. We apply the same idea to *typing derivations*: given a derivation of an expression e and focusing on a sub-term e' in e, we focus on the subderivation of e' and pair it with a 'context typing derivation' of the surrounding context.

A context typing derivation therefore must record the extra assumptions made available at the focus by additional bound variables introduced in the captured context, and the mode of the focus (whether or not the sub-term synthesises a type). These are necessary for type checking the focus (i.e. e') directly from the typing derivation of the context.

5.1 Context Syntax and Slice Lattice

Formally we must first define one-hole contexts on terms themselves (Figure 9). The focus of the context is notated $\circ$ and is restricted to a single position in an expression. For any context, a term may be placed at the focus to produce a complete term. This is notated by $\mathcal{C}[e]$, 'plugging e into context $\mathcal{C}$'.

$\boxed{\text{Syntax}}$ One-hole expression contexts $\mathcal{C}$

$$\begin{aligned}\mathcal{C} ::= {} & \circ \mid \lambda x : \tau.\ \mathcal{C} \mid \lambda x.\ \mathcal{C} \mid \mathcal{C}(e) \mid e(\mathcal{C}) \mid \mathcal{C}\langle\tau\rangle \\ \mid {} & (\mathcal{C},\ e) \mid (e,\ \mathcal{C}) \mid \iota_1\ \mathcal{C} \mid \iota_2\ \mathcal{C} \mid \pi_1\ \mathcal{C} \mid \pi_2\ \mathcal{C} \\ \mid {} & (\textbf{case}\ \mathcal{C}\ \textbf{of}\ x.\ e \mid y.\ e) \\ \mid {} & (\textbf{case}\ e\ \textbf{of}\ x.\ \mathcal{C} \mid y.\ e) \mid (\textbf{case}\ e\ \textbf{of}\ x.\ e \mid y.\ \mathcal{C}) \\ \mid {} & \Lambda\alpha.\ \mathcal{C} \mid \textbf{let}\ x = \mathcal{C}\ \textbf{in}\ e \mid \textbf{let}\ x = e\ \textbf{in}\ \mathcal{C}\end{aligned}$$

$\boxed{\mathcal{C}[e]}$ Plug expression e into the hole $\circ$

$$\begin{aligned}\circ[e] &= e \\ (\lambda x : \tau.\ \mathcal{C})[e] &= \lambda x : \tau.\ \mathcal{C}[e] \\ (\mathcal{C}_1(e_2))[e] &= \mathcal{C}_1[e](e_2)\end{aligned}$$

(similarly for all other constructors)

$\boxed{\square_{\mathcal{C}}}$ Purely structural context: replace all sub-terms with $\square$, preserving structure

$$\square_{\circ} = \circ \qquad \square_{\lambda x:\tau.\ \mathcal{C}} = \lambda x : \square.\ \square_{\mathcal{C}} \qquad \square_{\mathcal{C}(e)} = \square_{\mathcal{C}}(\square) \quad \text{etc.}$$

Fig. 9. One-hole expression contexts. $\circ$ marks the hole; $\mathcal{C}[e]$ plugs it.

Precision extends to contexts, $\mathcal{C}' \sqsubseteq \mathcal{C}$, additionally requiring the focus to be in the same structural position. This is required so that plugging a context preserves precision:

Lemma 5.1 (Plug Preserves Precision). *If $\mathcal{C}' \sqsubseteq \mathcal{C}$ and $e' \sqsubseteq e$, then $\mathcal{C}'[e'] \sqsubseteq \mathcal{C}[e]$.*

This lemma allows slicing to independently slice the *external* type information from the context and the *internal* type information of the focused term, whilst ensuring the whole program slice (plugging the two together) remains a valid slice of the original program.

5.2 The Context Typing Judgement

Then, we notate context typing by a judgement $\Delta; \Gamma \vdash \mathcal{C}$ **at** $d \triangleright \Delta'; \Gamma' [m]$, read as: "under assumptions Δ, Γ, context $\mathcal{C}$ in a derivation of outer mode d has its focus typed under $\Delta'; \Gamma'$ in focus mode m".

The *outer derivation mode* d records whether the whole expression $\mathcal{C}[e]$ is in a synthesis derivation producing type τ ($d = \Rightarrow \tau$) or an analysis derivation against τ ($d = \Leftarrow \tau$). The *focus mode* m records the type the focus e must synthesise ($m = \Rightarrow \tau'$), or the type it is checked against ($m = \Leftarrow \tau'$). Finally, Γ' records the *focus assumptions*. Each rule in these derivations is subject to *additional premises* matching the corresponding term typing rules.

Example. Consider the derivation of $\Delta; \Gamma \vdash e_1(e_2) \Rightarrow \tau_2$ via $\Rightarrow$App, where $\Delta; \Gamma \vdash e_1 \Rightarrow \tau$, $\tau \sqcup (\square \to \square) \equiv \tau_1 \to \tau_2$, and $\Delta; \Gamma \vdash e_2 \Leftarrow \tau_1$.

- **Function position** (e_1 is the focus, $\mathcal{C} = \circ(e_2)$): The corresponding context typing is: $\Delta; \Gamma \vdash \circ(e_2)$ **at** $\Rightarrow \tau_2 \triangleright \Delta; \Gamma [\Rightarrow \tau]$. The focus must synthesise type τ as per the focus mode, with the extra premises being $\tau \sqcup (\square \to \square) \equiv \tau_1 \to \tau_2$ and $\Delta; \Gamma \vdash e_2 \Leftarrow \tau_1$.
- **Argument position** (e_2 is the focus, $\mathcal{C} = e_1(\circ)$): The corresponding context typing is: $\Delta; \Gamma \vdash e_1(\circ)$ **at** $\Rightarrow \tau_2 \triangleright \Delta; \Gamma [\Leftarrow \tau_1]$. The focus analyses against τ_1 as per the focus mode, with the extra premises being: $\Delta; \Gamma \vdash e_1 \Rightarrow \tau$ and $\tau \sqcup (\square \to \square) \equiv \tau_1 \to \tau_2$.

Specifying these rules is systematic, one for each typing rule and focus location. We give a few representative rules in Figure 10. The only additional rules are for identity contexts s$\circ$ and a$\circ$ (the base cases) where the focus mode must match the outer derivation mode.

$\boxed{\Delta; \Gamma \vdash \mathcal{C} \textbf{ at } d \triangleright \Delta'; \Gamma' [m]}$ Context $\mathcal{C}$ at outer derivation mode d types its focus under $\Delta'; \Gamma'$ in focus mode m

$\boxed{\text{Base cases}}$ identity contexts (focus mode matches outer derivation mode):

$$\text{s}\circ \frac{}{\Delta; \Gamma \vdash \circ \textbf{ at } \Rightarrow \tau \triangleright \Delta; \Gamma [\Rightarrow \tau]} \qquad \text{a}\circ \frac{}{\Delta; \Gamma \vdash \circ \textbf{ at } \Leftarrow \tau \triangleright \Delta; \Gamma [\Leftarrow \tau]}$$

$\boxed{\text{Synthesis mode: } d = \Rightarrow \tau}$ the overall expression $\mathcal{C}[e]$ synthesises:

$$\text{s}\circ_1 \frac{\Delta; \Gamma \vdash \mathcal{C} \textbf{ at } \Rightarrow \tau \triangleright \Delta'; \Gamma' [m] \quad \tau \sqcup (\square \to \square) \equiv \tau_1 \to \tau_2 \quad \Delta; \Gamma \vdash e_2 \Leftarrow \tau_1}{\Delta; \Gamma \vdash \mathcal{C}(e_2) \textbf{ at } \Rightarrow \tau_2 \triangleright \Delta'; \Gamma' [m]}$$

$$\text{s}\circ_2 \frac{\Delta; \Gamma \vdash e_1 \Rightarrow \tau \quad \tau \sqcup (\square \to \square) \equiv \tau_1 \to \tau_2 \quad \Delta; \Gamma \vdash \mathcal{C} \textbf{ at } \Leftarrow \tau_1 \triangleright \Delta'; \Gamma' [m]}{\Delta; \Gamma \vdash e_1(\mathcal{C}) \textbf{ at } \Rightarrow \tau_2 \triangleright \Delta'; \Gamma' [m]}$$

$\boxed{\text{Subsumption: } \Rightarrow \tau' \rightsquigarrow \Leftarrow \tau}$ the unique bridge from synthesis to analysis derivation mode:

$$\text{aSub} \frac{\Delta; \Gamma \vdash \mathcal{C} \textbf{ at } \Rightarrow \tau' \triangleright \Delta'; \Gamma' [m] \quad \tau \sim \tau'}{\Delta; \Gamma \vdash \mathcal{C} \textbf{ at } \Leftarrow \tau \triangleright \Delta'; \Gamma' [m]}$$

Fig. 10. Selected context typing rules.

Correctness of context typing is formalised via a *totality* theorem: every well-typed expression can be decomposed into context and expression at some focus, with a context typing matching a (consistent) typing derivation of the focused term. Conversely, a *soundness* theorem states that a valid context typing and expression satisfying the focus typing mode can be composed into a well-typed plugged expression.

Theorem 5.2 (Plug Decomposition – Totality). *For any context $\mathcal{C}$, expression e, and outer derivation mode d:*

- *If* $\Delta; \Gamma \vdash \mathcal{C}[e] \Rightarrow \tau$, *then there exist* Δ', Γ', *and* m *such that* $\Delta; \Gamma \vdash \mathcal{C}$ **at** $\Rightarrow \tau \triangleright \Delta'; \Gamma' [m]$ *and* e *satisfies focus mode* m *under* $\Delta'; \Gamma'$.
- *If* $\Delta; \Gamma \vdash \mathcal{C}[e] \Leftarrow \tau$, *then there exist* Δ', Γ', *and* m *such that* $\Delta; \Gamma \vdash \mathcal{C}$ **at** $\Leftarrow \tau \triangleright \Delta'; \Gamma' [m]$ *and* e *satisfies focus mode* m *under* $\Delta'; \Gamma'$.

Here "e satisfies focus mode m under $\Delta'; \Gamma'$" means: if $m = \Rightarrow \tau'$ then $\Delta'; \Gamma' \vdash e \Rightarrow \tau'$; if $m = \Leftarrow \tau'$ then $\Delta'; \Gamma' \vdash e \Leftarrow \tau'$.

Theorem 5.3 (Plug Composition – Soundness). *For any context $\mathcal{C}$, expression e, outer mode d, and focus mode m: if $\Delta; \Gamma \vdash \mathcal{C}$ **at** $d \triangleright \Delta'; \Gamma' [m]$ and e satisfies focus mode m under $\Delta'; \Gamma'$, then $\mathcal{C}[e]$ satisfies outer derivation mode d under $\Delta; \Gamma$.*

In order to apply the same ideas as synthesis slices, we also prove a substantial static gradual guarantee (§3.3) for context typings, defining precision on modes $m' \sqsubseteq m$ by lifting: $\Rightarrow \tau' \sqsubseteq \Rightarrow \tau$ iff $\tau' \sqsubseteq \tau$, and likewise for $\Leftarrow$. The focus mode's class itself (analysis or synthesis) is propagated unchanged through every context typing rule.

Theorem 5.4 (Static Gradual Guarantee – Context, Synthesis Position). *If $\Gamma' \sqsubseteq \Gamma$, $\mathcal{C}' \sqsubseteq \mathcal{C}$, and $\Delta; \Gamma \vdash \mathcal{C}$ **at** $\Rightarrow \tau_p \triangleright \Delta_f; \Gamma_f [m]$, then there exist τ'_p, Γ'_f, and m' with $\tau'_p \sqsubseteq \tau_p$, $\Gamma'_f \sqsubseteq \Gamma_f$, $m' \sqsubseteq m$, and $\Delta; \Gamma' \vdash \mathcal{C}'$ **at** $\Rightarrow \tau'_p \triangleright \Delta_f; \Gamma'_f [m']$.*

Theorem 5.5 (Static Gradual Guarantee – Context, Analysis Position). *If $\Gamma' \sqsubseteq \Gamma$, $\mathcal{C}' \sqsubseteq \mathcal{C}$, $\tau'_p \sqsubseteq \tau_p$, and $\Delta; \Gamma \vdash \mathcal{C}$ **at** $\Leftarrow \tau_p \triangleright \Delta_f; \Gamma_f [m]$, then there exist Γ'_f and m' with $\Gamma'_f \sqsubseteq \Gamma_f$, $m' \sqsubseteq m$, and $\Delta; \Gamma' \vdash \mathcal{C}'$ **at** $\Leftarrow \tau'_p \triangleright \Delta_f; \Gamma'_f [m']$.*

6 Analysis Slices

With this in mind, analysis slices are simple analogues to synthesis slices, except being conducted on *analysis inner mode* context typing derivations. And, as the overall programs that we consider must synthesise a type, the usual analysis slices concern derivations only in *synthesis* outer derivation mode.[6]

Intuitively, slices of the context explain why a sub-term is *expected* by the surrounding context to synthesise or analyse against a given type. We typically only care about the cases where the inner mode is in *analysis*; a (well-typed) surrounding context does not actually influence *why* a term synthesises a type.[7]

6.1 Definition

Structurally, analysis slices are identical to synthesis slices (§4.1), but with expression slices ς replaced by context slices $\kappa \in \lfloor \mathcal{C} \rfloor$.

[6] An extension to analysis outer modes, which is useful for efficient calculation, is discussed later.

[7] A context may define a variable which is used in a synthesising term, non-locally affecting the subterm's type. However, assumption slices are treated entirely *separately* in both modes. That is, we do not ask the analysis slice to explain how exactly the bindings provide the minimal assumptions for the focus, only how it explains the *type* at the focus.

Definition 6.1. An *analysis slice* of $Cls : \Delta; \Gamma \vdash \mathcal{C}$ **at** $\Rightarrow \tau_p \triangleright \Delta'; \Gamma' \ [\Leftarrow \tau]$ on $v \in \lfloor \tau \rfloor$, written $AnaSlice\ Cls \blacktriangleleft v$, is a pair $(\kappa, \gamma) \in \lfloor \mathcal{C}, \Gamma \rfloor$ such that $\Delta; \gamma \vdash \kappa$ **at** $\Rightarrow \psi \triangleright \Delta''; \Gamma'' \ [\Leftarrow \phi]$ for some inner-type witness $\phi \sqsupseteq v$ (with $\phi \in \lfloor \tau \rfloor$ and $\psi \in \lfloor \tau_p \rfloor$ by graduality, Theorem 5.4).

6.2 Minimality

$MinAnaSlice\ Cls \blacktriangleleft v$ is defined analogously to Definition 4.3, minimising over the context slice (κ, γ). And as before, existence of minimal analysis slices and monotonicity follow from finiteness and graduality.

Theorem 6.2 (Existence of minimal analysis slices). *For every $s : AnaSlice\ Cls \blacktriangleleft v$, there exists $m : MinAnaSlice\ Cls \blacktriangleleft v$ with $m \sqsubseteq s$.*

Theorem 6.3 (Monotonicity of minimal analysis slices). *If $v_1 \sqsubseteq v_2$ in $\lfloor \tau \rfloor$ and $m_2 : MinAnaSlice\ Cls \blacktriangleleft v_2$, then there exists $m_1 : MinAnaSlice\ Cls \blacktriangleleft v_1$ with $m_1 \sqsubseteq m_2$.*

6.3 Minimally Scoped Analysis Slices

As mentioned earlier, a synthesising term derives all of its type information internally.[8] Therefore, the type is actually only enforced by the context up until the innermost subderivation with a synthesis outer derivation mode.

For example, for an annotated term nested inside another annotation, only the inner annotation actually enforces a type on the inner term. Therefore, an algorithm need only check this region (the outer context can be fully omitted).

7 Error Marking & Type Error Debugging

This section formally applies type slicing to static error squiggles as in Fig. 3 (§1.2). It demonstrates that we can not only use this calculus to "explain types" in well-typed programs, but also effectively "explain type errors" in ill-typed programs.

This builds upon the theory of type error marking for bidirectional systems with holes of Zhao et al. [56], which we briefly recap next and extend to the type slicing core calculus (§3).

7.1 Error Marking

A marking algorithm is a pair of bidirectional judgements,

$$\Delta; \Gamma \vdash e \rightsquigarrow \check{e} \Rightarrow \tau \qquad \text{and} \qquad \Delta; \Gamma \vdash e \rightsquigarrow \check{e} \Leftarrow \tau,$$

reading "under $\Delta; \Gamma$, the source expression e *marks to* the marked expression $\check{e}$, synthesising (resp. being analysed against) type τ". The marked output $\check{e}$ has the same structure as e but wraps each point of local type failure in a *mark*. The marks, representing highlighting an entire erroneous term e, are tabulated in Table 1 alongside the corresponding typing rule failures.

The core calculus studied gives rise to four classes of errors: *type inconsistencies* when subsumption cannot be applied, *shape errors* when rules expect a sub-term to join (match) a specific type shape, *mode limitations* where syntax which cannot synthesise its type is expected to provide type information; and variable *scoping errors*, which do not pertain to types and so cannot be debugged by type slicing.

Importantly, marked terms can be type-checked by treating the marks as 'non-empty' holes, synthesising $\square$ (and recursively marking and type-checking the inner term).

[8] Again, given that assumptions are tracked externally.

Mark	Class	Failed Rule(s)	Error
$\langle\!\lfloor e \rfloor\!\rangle_{\tau' \not\sim \tau}$	type inconsistency	Subsumption	e synthesises τ' inconsistent with analysis type τ
$\langle\!\lfloor e \rfloor\!\rangle_{\blacktriangleright \to}$	shape	Function Application	e is in a function position, but is not a function
$\langle\!\lfloor e \rfloor\!\rangle_{\blacktriangleright +}$	shape	Case Expression	e is a case scrutinee, but is not a sum type
$\langle\!\lfloor e \rfloor\!\rangle_{\blacktriangleright \times}$	shape	Product Projection	e is projected as a product, but is not a product
$\langle\!\lfloor e \rfloor\!\rangle_{\blacktriangleright \forall}$	shape	Type Application	e is in a type function position, but is not a type function
$\langle\!\lfloor \lambda x.\, e \rfloor\!\rangle_{\sim\to}$	mode limitation	Lambda Synthesis	Unannotated lambda in synthesis position
$\langle\!\lfloor \lambda x.\, e \rfloor\!\rangle_{\sim\to}$	mode limitation	Lambda Analysis	Unannotated lambda analysed against a non-function type
$\langle k \rangle \Rightarrow$	scope	Variables	Variable k has no binding in scope

Table 1. Error Mark Forms Classified

The validity of this method is formalised by two theorems: *totality* ensures marks account for every possible typing error, and *erasure* ensures adding marks does not change a term's structure.

Theorem 7.1 (Totality of Marking). *For every expression e and assumptions $\Delta;\Gamma$, there exist a marked expression $\check{e}$ and type τ such that $\Delta;\Gamma \vdash e \rightsquigarrow \check{e} \Rightarrow \tau$. And, for every type τ', there exists $\check{e}'$ such that $\Delta;\Gamma \vdash e \rightsquigarrow \check{e}' \Leftarrow \tau'$.*

Theorem 7.2 (Erasure of Marking). *If $\Delta;\Gamma \vdash e \rightsquigarrow \check{e} \Rightarrow \tau$ then* $\mathrm{erase}(\check{e}) \equiv e$*, and likewise for analysis.*

7.2 Marked Context Typing

In this style, we extend context typing (§5) to *marked* context typing $\Delta;\Gamma \vdash \mathcal{C} \rightsquigarrow \check{\mathcal{C}}$ **at** $p \triangleright \Delta';\Gamma'\ [\,m\,]$, threading a marked context $\check{\mathcal{C}}$ alongside $\mathcal{C}$ (with context marks defined analogously to expressions), adding one error rule per mark form of Table 1. We have proven analogous totality and soundness theorems hold for marked contexts, mirroring the unmarked plug decomposition and composition (Theorems 5.2 and 5.3).

7.3 The Debugging Recipe by Example

We may now relate the tools developed over the last four sections to explain type errors:

(1) Place the focus (cursor) at some marked term. This splits the marked program into a marked context $\check{\mathcal{C}}$ and marked focus $\check{e}$, with $\check{\mathcal{C}}[\check{e}]$ recovering the program; a marked context typing at the focus is then derivable by totality of marked context typing.
(2) Synthesis slice the internal type information, the term at the *focus*, $\check{e}$, if the focus mode m is in synthesis ($\Rightarrow\ \tau$).
(3) Analysis slice the external type information, the *context typing* itself, if the focus mode m is in analysis ($\Leftarrow\ \tau$).
(4) Pick queries to explore the error. For type-inconsistency marks *both* the synthesis and analysis slices exist (two context typings exist from step 1). From these, *only* the inconsistent parts need be queried to explain the error. For shape errors, simply query just the outermost shape of the synthesised type; mode-limitation errors are analogous on the analysis side.

We give some worked examples, notating with a marked term, a synthesis slice, and/or an analysis slice. 'Purely structural' slice regions which do not add or manipulate type information (i.e. bindings) are in a lighter shade.

7.3.1 Type inconsistency: $(\!|\cdot|\!)_{\tau' \nsim \tau}$. Consider the program

$$\textbf{let } p : 1 \to 1 = (\texttt{true}, \texttt{false}) \textbf{ in } p.$$

The RHS $(\texttt{true}, \texttt{false})$ synthesises $\texttt{Bool} \times \texttt{Bool}$ but is analysed against $1 \to 1$. Marking inserts $(\!|\cdot|\!)_{\texttt{Bool}\times\texttt{Bool} \nsim 1\to 1}$ around the RHS:

$$\textbf{let } p : 1 \to 1 = (\!|(\texttt{true}, \texttt{false})|\!)_{\texttt{Bool}\times\texttt{Bool} \nsim 1\to 1} \textbf{ in } p.$$

Marking would ordinarily report the error $(\!|\cdot|\!)_{\texttt{Bool}\times\texttt{Bool} \nsim 1\to 1}$ with the entire term highlighted. However, in reality, only the outermost type constructors are actually causing the (initial) type inconsistency, and type slicing allows query refinements of the full types. In full:

Decomposition. The (context, focus) pair the marking factors into is

$$\mathcal{C} \;=\; \textbf{let } p : 1 \to 1 = \circ \textbf{ in } p, \qquad e \;=\; (\texttt{true}, \texttt{false}),$$

with the focus at $\Leftarrow (1 \to 1)$. The two debugging queries run on the two sides of this split.

Synthesis Slice. *MinSynSlice* at the minimal type-slice $v \sqsubseteq \texttt{Bool} \times \texttt{Bool}$ that still witnesses $v \nsim (1 \to 1)$:

$$e \;\rightsquigarrow\; (\,\texttt{true}\,,\,\texttt{false}\,) \qquad \text{with} \qquad v = \square \times \square.$$

Analysis Slice. *MinAnaSlice* at the minimal $\psi \sqsubseteq 1 \to 1$ that still witnesses $(\texttt{Bool} \times \texttt{Bool}) \nsim \psi$:

$$\mathcal{C} \;\rightsquigarrow\; \textbf{let } p : 1 \to 1 = \circ \textbf{ in } p \qquad \text{with} \qquad \psi = \square \to \square.$$

Full view: The slices pick out exactly what each side contributes to the failure.

synthesis query $v = \texttt{Bool} \times \texttt{Bool}$ $(\equiv \square \times \square)$
analysis query $\psi = 1 \to 1$ $(\equiv \square \to \square)$

$$\textbf{let } p : 1 \to 1 = (\!|(\,\texttt{true}\,,\,\texttt{false}\,)|\!)_{\texttt{Bool}\times\texttt{Bool} \nsim 1\to 1} \textbf{ in } p.$$

We can see that this recipe does not *arbitrarily* blame either the synthesising term or the analysing context for the error; after all, errors are commonly in annotations, especially when assigning types in a gradual language (49%) [30].

However, type slicing is flexible. A user who *does* want to trust the surrounding context (annotations), as is often done in conventional type error highlighting, can simply only query the synthesis-side. Or vice versa, to trust the implementation, which is useful when refactoring code to a different type, which conflicts with existing annotations.

7.3.2 Shape mismatches: $(\!|\cdot|\!)_{\blacktriangleright\to}$, $(\!|\cdot|\!)_{\blacktriangleright+}$, $(\!|\cdot|\!)_{\blacktriangleright\times}$, $(\!|\cdot|\!)_{\blacktriangleright\forall}$. The shape-mismatch marks all occur in similar situations: the syntactic position of the focus requires a type kind, but the focus's synthesised type does not match this.

A representative example is $e = ((), ())(\texttt{true})$, where $((),())$ is syntactically in a function position, but synthesises 1×1, which is *not* a function, i.e. does not join with $\square \to \square$. We can therefore query just why it is a product, and not a function.

synthesis query $v = 1 \times 1$
analysis query $\psi = -$
$(\!|((),())|\!)_{\blacktriangleright\to}\,(\texttt{true})$.

7.3.3 Unannotated Lambda against Non-Arrow: $(\!|\cdot|\!)_{\rightsquigarrow}$. Unannotated lambdas $\lambda y.\, e$ have no synthesis rule; their type can only be determined by the surrounding context. Consequently it has no synthesis type, so type inconsistencies are not marked via subsumption, but instead by a mode limitation mark. In this situation all the relevant type information is derived from the surrounding context, so we use analysis slices.

Consider the program **let** b : Bool $= \lambda y.\, y$ **in** b. The RHS $\lambda y.\, y$ is a bare lambda analysed against Bool. Taking the *MinAnaSlice* on the enclosing context witnesses the context enforcing a non-function type (Bool):

synthesis query $v = -$
analysis query $\psi =$ Bool

$$\textbf{let}\ b : \texttt{Bool} = (\!|\lambda y.\, y|\!)_{\rightsquigarrow}\ \textbf{in}\ b.$$

7.3.4 Unbound variable: $\langle k \rangle \Rightarrow$. The free-variable mark records a *scope* failure. There are no types involved, so type slicing is inapplicable.

8 Optimising Type Slice Calculation

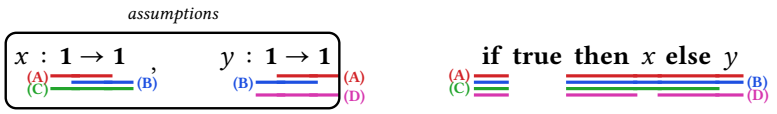
The theory of synthesis and analysis slices (§4, §6) establishes *which* slices exist, and provides a brute-force algorithm. This section explores these structures further to improve the brute-force algorithm.

8.1 Products of Synthesis Slices

We earlier suggested recursively calculating minimal slices of subterms and composing them together (§4.5). However, this cannot be done naively: the product of two minimal slices is not necessarily minimal. Consider the following if statement:

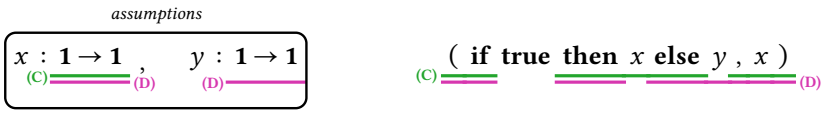
$$\Gamma = (x : \mathbf{1} \to \mathbf{1},\ y : \mathbf{1} \to \mathbf{1}), \qquad e = \textbf{if true then } x \textbf{ else } y, \qquad \tau = \mathbf{1} \to \mathbf{1},$$

and ask for slices of $D : \Delta; \Gamma \vdash e \Rightarrow \tau$ at the full query $v = \mathbf{1} \to \mathbf{1}$. There are *four* incomparable minimal full slices, displayed together as underlines:



- **(A)** Both branches are retained; the assumption slice keeps x's domain only and y's codomain only (joining to $1 \to 1$).
- **(B)** Symmetric to (A), taking the opposite choices for the assumption slice
- **(C)** **Else**-branch omitted.
- **(D)** Symmetric to (C): **then**-branch omitted.

Slices (A) and (B) are minimal because each variable is sliced *differently* across the two uses, and none are *program* slices of each other. However, notice that (A) and (B) are not subslices of the product with x, where only 2 minima exist:



When this expression is embedded in the product (**if true then** x **else** y, x), the two assumptions are joined. However, the right projection's use of x is strictly stronger than that of slices (A), (B), and (D); we are *aliasing* this variable's assumption. Slices (A) and (B) are no longer minimal under this aliased assumption, meaning they are not subslices of the overall product. Slice (D) is still valid because it does not actually use x.

8.2 Term-Minimal Slices

To rule out slices like (A) and (B) (§8.1), we strengthen the notion of slices. We first calculate the minimal expression under *maximal* typing assumptions, and then minimise the assumptions afterwards. By graduality, this is a minimal synthesis slice.

Definition 8.1 (Term-minimal slice). A *term-minimal slice* of $D : \Delta; \Gamma \vdash e \Rightarrow \tau$ on $v \in \lfloor \tau \rfloor$, written $\mathit{TermMin}\ D \blacktriangleleft v$, is a term slice $\varsigma \in \lfloor e \rfloor$ such that $\Delta; \Gamma \vdash \varsigma \ \Rightarrow\ \psi$ for some $\psi \sqsupseteq v$ (with $\psi \in \lfloor \tau \rfloor$ by graduality, Theorem 3.5).

Minimality lifts directly from the slice lattice on e alone, and existence and monotonicity follow analogously to synthesis slices (Theorems 4.5 and 4.6).

Definition 8.2 (Minimal Term-Minimal Slices). A $\mathit{MinTermMin}\ D \blacktriangleleft v$ is a $\mathit{TermMin}\ D \blacktriangleleft v$ minimal in $\mathit{TermMin}\ D \blacktriangleleft v$ under term-slice precision on $\lfloor e \rfloor$.

Crucially, the two notions of minimality do not coincide: term-minimality is strictly stronger. Not all minimal slices are term-minimal under maximal assumptions.

> Counterexample 8.3 (Minimal ⇏ Term-Minimal). *Slices (A) and (B) (§8.1) are minimal synthesis slices but are **not** term-minimal.*

8.3 Calculating Term-Minimal Slices

The rest of this section develops a structural calculus that characterises minimal term slices via a judgement:

$$\Gamma \vdash e \Rightarrow \tau \blacktriangleleft v \rightsquigarrow \varsigma \Rightarrow \psi \dashv \gamma,$$

read "the typing $\Gamma \vdash e \Rightarrow \tau$ at query v produces term slice ς, synthesises ψ, and uses assumptions γ". The fourth output γ is a minimal assumption slice to retrieve a minimal slice corresponding to the term-minimal slice.
The cases that deserve commentary are products, the binding constructs, function application, and type application.

8.3.1 Products. With term minimality, we now have that two minimal synthesis slices which are also term-minimal can be composed into a term-minimal, minimal synthesis slice. For a query $v_1 \times v_2$ on the product, the projections are queried with v_1 and v_2 individually:

$$\Rightarrow\mathtt{Pair}\ \frac{\Delta; \Gamma \vdash e_1 \Rightarrow \tau_1 \blacktriangleleft v_1 \rightsquigarrow \varsigma_1 \Rightarrow \psi_1 \dashv \gamma_1 \qquad \Delta; \Gamma \vdash e_2 \Rightarrow \tau_2 \blacktriangleleft v_2 \rightsquigarrow \varsigma_2 \Rightarrow \psi_2 \dashv \gamma_2}{\Delta; \Gamma \vdash (e_1, e_2) \Rightarrow \tau_1 \times \tau_2 \blacktriangleleft v_1 \times v_2 \rightsquigarrow (\varsigma_1, \varsigma_2) \Rightarrow \psi_1 \times \psi_2 \dashv \gamma_1 \sqcup \gamma_2}$$

8.3.2 Function Application. All the synthesised type information from a function application is derived from the function itself. We may simply omit the argument, and slice the function by $\square \to v$:

$$\Rightarrow\mathtt{App}\ \frac{\Delta; \Gamma \vdash e_2 \Leftarrow \tau_1 \qquad v \neq \square \qquad \Delta; \Gamma \vdash e_1 \Rightarrow \tau_1 \to \tau_2 \blacktriangleleft \square \to v \rightsquigarrow \varsigma_{fn} \Rightarrow \psi_1 \dashv \gamma}{\Delta; \Gamma \vdash e_1\, e_2 \Rightarrow \tau_2 \blacktriangleleft v \rightsquigarrow \varsigma_{fn}(\square) \Rightarrow \mathrm{cod}(\psi_1) \dashv \gamma}$$

8.3.3 Binding Constructs. Rules which bind variables are more complex, as they abstract a variable used in the minimised body. The general method is to slice them by first slicing the *body*, extracting the assumption it needed on the bound variable from the minimal assumptions γ, then slice the definition / type annotation using this.

Notice that this is the *opposite* direction to typechecking, e.g. let bindings type the definition first, then use the result to type the body, but slicing slices the body first to determine the minimal assumption to slice the definition with.

8.3.4 Type Application. Type application works by matching the type function's type against the query to collect a list of constraints on the type variable, the *join* of which gives the sliced type argument $\hat{\sigma}$. The type function is then sliced with a matched query $\hat{v}$, where each position corresponding to the type variable α in the type function's type is replaced by α itself and other positions are carried over from the query unchanged. Figure 11 illustrates the construction on a query whose two subparts are matched with an α with differing constraints.

$$e \Rightarrow \forall\alpha.(\alpha \times \alpha \times \mathbf{Bool}), \quad \text{applied at} \quad \sigma = \mathbf{1} \to (\mathbf{Bool} \times \mathbf{Bool}):$$
$$\text{So } e\langle\sigma\rangle \Rightarrow (\mathbf{1} \to (\mathbf{Bool} \times \mathbf{Bool})) \times (\mathbf{1} \to (\mathbf{Bool} \times \mathbf{Bool})) \times \mathbf{Bool}.$$
$$\textbf{Query} \quad v = \underbrace{\mathbf{1} \to \square}_{\alpha} \times \underbrace{\square \to (\mathbf{Bool} \times \square)}_{\alpha} \times \square$$
$$\textbf{Matched query} \quad \hat{v} = \alpha \times \alpha \times \square$$
$$\textbf{Constraints} \quad \mathbf{1} \to \square \sqsubseteq \alpha \qquad \square \to (\mathbf{Bool} \times \square) \sqsubseteq \alpha$$
$$\textbf{Constraints Join} \quad \hat{\sigma} = \mathbf{1} \to (\mathbf{Bool} \times \square)$$

Fig. 11. Type application query constraints.

8.4 Case Expressions

The case rule is the difficult one, appearing to require an iterative fixed point algorithm. This is because the two branch types join to produce the resulting type which is being queried upon. As synthesis slices are not exact, a query may be split into two disjoint types, yet a minimal slice of one branch may produce a more precise type than the query (overflow), which is also produced by a sibling branch. The sibling branch may then admit further slicing, as it no longer needs to provide this redundant part of the type. Lack of an upwards version of monotonicity (Fig. 8b) makes this difficult to solve in general without brute force.

However, when the branch slices exactly synthesise disjoint type queries, then there is no redundancy and the result is minimal. The scrutinee can be sliced using the join of the minimal assumptions on the branches, much like with let bindings.

To express disjoint types we can use *co-Heyting subtraction* $v_1 \setminus v_2$, which is the least slice whose join with v_2 recovers v_1, i.e. "the part of v_1 not already covered by v_2", e.g. $1 \times 1 \setminus 1\times\square = \square\times 1$.

Then, a disjoint cover of query v is one such that $v \setminus v_1 = v_2$ and $v \setminus v_2 = v_1$. The case expression will necessarily synthesise $v_1 \sqcup v_2$ which equals v by standard co-Heyting algebra rules. Note that the lattice is *not* a boolean algebra, so $v_1 \sqcap v_2 \neq \square$ in general.

The Hazel implementation simply does a disjoint split and ignores the overflow issue to work in linear time complexity. This is why it is an approximation. But, it is perfectly feasible to use this approximation and then continue with the brute force algorithm, as we expect the approximation to be very close to the minimum in almost all cases. Further, as overflow only occurs due to variable aliasing, the brute force search need only consider aliased variables at the leaves.

8.5 Calculating Analysis Slices

Analysis slices are calculated directly from synthesis slices. The only rule that creates a type checking judgement is function application. Consider the analysis slice of a function argument with query type v_1; it is just the function application and a slice of the function under $\square \to v_1$. That is, to explain why a function argument must be a given type, we ask why the corresponding function has that type as its domain.

However, to recursively calculate the analysis slice of an application when the focus is deep inside the argument, the analysis slice of the argument itself must be calculated, but its outer derivation mode is in *analysis*. The analysis slices defined earlier (§6.1) do not apply here.

We need an additional notion of *inner* analysis slices. Dually to regular (outer) analysis slices, which treat the synthesis mode as an *input*, the inner slices treat the analysis derivation mode as an *output*, and attempt to minimise it.

Intuitively, an inner analysis slice gives the minimal assumptions, context, and additionally the minimum *analysis type* to be imposed on it by the outer context, such that the queried analysis type holds at its focus. In particular, slicing the function in an application would use this minimal outer analysis mode type on the argument to slice the domain type of the function.

Definition 8.4. An *inner analysis slice* of $Cls : \Delta; \Gamma \vdash \mathcal{C}$ **at** $\Leftarrow \tau_p \triangleright \Delta'; \Gamma' [\Leftarrow \tau]$ on $v \in \lfloor \tau \rfloor$, written $InnerAnaSlice\ Cls \blacktriangleleft v$, is a triple $(\kappa, \gamma, v_{\text{outer}}) \in \lfloor \mathcal{C}, \Gamma, \tau_p \rfloor$ such that

$$\Delta; \gamma \vdash \kappa \textbf{ at } \Leftarrow v_{\text{outer}} \triangleright \Delta''; \Gamma'' [\Leftarrow \phi]$$

for some inner-type witness $\phi \sqsupseteq v$ (with $\phi \in \lfloor \tau \rfloor$ by graduality, Theorem 5.5).
A minimal inner analysis slice minimises the whole triple, v_{outer} included (so that an application slices the function's domain by no more than the argument requires).

9 Full Type Slices – Slicing Assumption Definitions

Synthesis slices are explained using a set of *minimal assumptions*. However, in real code, these are actually *provided by the surrounding context* (or from external code). Providing these minimal assumptions via a full slice of this context can be very meaningful. For example, when debugging a type error, the error itself may be *non-local*, within a variable definition elsewhere, giving one of the used variables the wrong type. In this situation the error is in the definition, and we could debug it by inspecting (and slicing) that definition.

A *full type slice* is a fully sliced decomposition of a program around a focus, which can be pieced back together to form a well-typed program with the same (or stronger) typing properties at the focus as some given query:

Definition 9.1 (Type slices). A *type slice* of $Cls : \Delta; \Gamma \vdash \mathcal{C}$ **at** $\Rightarrow \tau_p \triangleright \Delta'; \Gamma' [\Rightarrow \tau]$ and $D : \Delta'; \Gamma' \vdash e \Rightarrow \tau$ on $v \in \lfloor \tau \rfloor$, written $TypeSlice^{\Rightarrow}\ (Cls, D) \blacktriangleleft v$, is a pair $((\kappa, \gamma), (\gamma_f, \varsigma)) \in \lfloor \mathcal{C}, \Gamma \rfloor \times \lfloor \Gamma', e \rfloor$ such that $(\gamma_f, \varsigma) : SynSlice\ D \blacktriangleleft v$, and $\Delta; \gamma \vdash \kappa$ **at** $\Rightarrow \psi \triangleright \Delta'; \gamma_f' [\Rightarrow \phi']$ for some ψ, and $\gamma_f' \sqsupseteq \gamma_f$ such that $\Delta'; \gamma_f' \vdash \varsigma \Rightarrow \phi'$.

Importantly, the pair of slices forms a well-typed program: by context typing soundness, $\kappa[\varsigma]$ synthesises ψ. Then, a *minimal* type slice is one in which the context κ and the focused expression ς are jointly minimised alongside minimal external typing assumptions of the context typing γ. Notably, we do *not* minimise the assumptions to the focused term, as its assumptions are now explicitly provided by the surrounding context (and γ).

As slices can be inexact, efficiently calculating a minimal context providing sufficient assumptions to a minimal *SynSlice* at the focus is non-trivial. The Hazel implementation instead closely approximates this by slicing the definition / annotation on each binding by its required type as per the minimal assumptions of the focused term. When each slice of these definitions is exact, the resulting type slice is minimal.

For terms in analysis mode, the full type slice is just the analysis slice, as the empty focus (with type $\square$) analyses successfully against any type.

10 Minimum-Size Slices

Minimal slices are not unique, and some are actually larger than others in the sheer number of terms highlighted. So, one might rank them by *size*: the number of highlighted positions, asking for a *minimum-size* slice. For this core calculus, finding such a slice is **NP-hard** in general. This implies that an algorithm attempting to also reduce the absolute size of slices should use heuristics to avoid or reduce the likelihood of cases with exponential time complexity.

We can construct a reduction from the NP-hard set cover optimisation problem as follows, and shown in Fig. 12:

- Encode a universe $U = \{1, \ldots, n\}$ as a type $\tau = 1 \times \cdots \times 1$ (n factors, one position per element)
- Encode each of m-many sets S_j in the set cover problem by an assumption $x_j : T_j \sqsubseteq \tau$ in Γ with T_j defined by the ith projection of T_j being 1 if and only if $i \in S_j$.
- Construct a case tree with empty scrutinees and placing all (x_j, I_j), defining I_j as the m-ary product with jth projection being 1 and all others $\square$.

Then, if a synthesis slice is queried on this case tree with $\tau \times 1 \times \cdots \times 1$, each I_j must be retained, so the whole tree structure must be retained, and the only variance in the size of the slices is purely dependent on which variables x_j were retained.

As we queried τ in the first projection then $\bigsqcup_{j=1}^{m} T_j = \tau$, which implies there are m subsets[9] $S'_j \subseteq S_j$ such that $\bigcup_{j=1}^{m} S'_j = U$. Hence, the retained S_j form a set-cover. Finally, the only variance in the size of the slice was in the number of variables x_j chosen, i.e. the number of sets chosen; hence, a minimum-size synthesis slice corresponds to a minimum set cover.

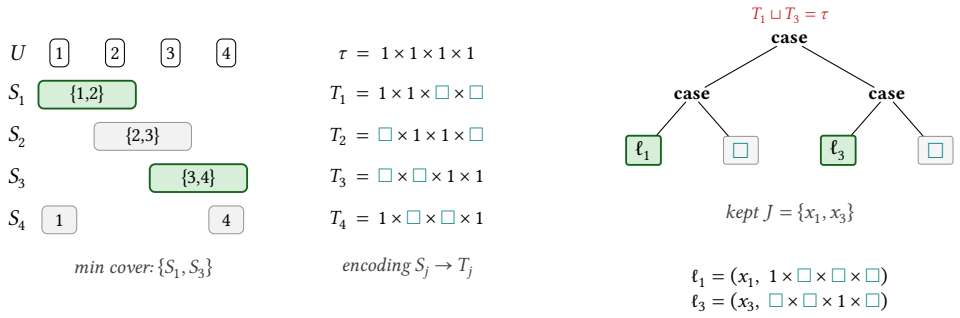


Fig. 12. NP-Hardness of Minimum-Size Slicing

11 Related Work

Program slicing methods have been explored extensively, in various forms and for many applications [46]. Here we compare type slicing with these, being loosely related in foundation to *Galois slicing* [34], and in application to *type error slicing* [19, 50], before relating to the broader type debugging literature (§11.5).

11.1 Classical Program Slicing

Classical program slicing [46] was first introduced by Weiser [51]. Here, a slice is a subprogram preserving the value of a chosen variable at a chosen program point: a 'criterion'. Similarly to type slicing, the subprogram is a syntactically well-formed program which can be analysed (evaluated).

[9] Subsets, as a minimal slice can use less precise assumptions x_j, i.e. $x_j : T'_j \sqsubseteq T_j$

The criterion is superficially similar to type slicing queries, with two significant differences. First, a type query is itself a slice of a type, which has *compound* structure, rather than a set of plain variables. Second, type slices cannot in general *preserve* all queries exactly (Theorem 4.2), so we ask only for *validity* (Theorem 4.1), which remains a 'sufficient' explanation.

Classical slicing distinguishes *static* slices (preserved over all executions) from *dynamic* slices (over one fixed execution) [24]. Type slicing can be viewed as a *dynamic* slicing method, as it fixes a particular typing derivation; however, most type systems, including ours, have no non-trivial non-determinism, making this distinction largely irrelevant.

11.2 Galois Slicing

Galois slicing [34] is a lattice-theoretic approach to dynamic program slicing that has been applied and related to a wide range of calculi and user interfaces [3, 35, 36, 39]. Type slicing shares the foundational use of lattices and precision, but differs decisively in the lack of a *Galois connection*.

In Perera's setting, a program evaluation lifts to a *meet-preserving* map $\mathit{eval}_{\rho,e} : \lfloor \rho, e \rfloor \to \lfloor v \rfloor$ between the slice lattices of partial programs and partial values. By an adjoint functor theorem [15, Prop. 7.34], such a map is the upper adjoint of a unique Galois connection, whose lower adjoint *uneval* is the 'backward slice function' sending a partial-value query to a *least* program slice evaluating to it.

For total type systems assigning unique types like the core calculus (§3), the natural type-slicing analogue of $\mathit{eval}_{\rho,e}$ maps partial assumptions and expressions to partial types by type checking.

$$\phi_D : \lfloor \Gamma, e \rfloor \to \lfloor \tau \rfloor$$

However, this 'forward map' ϕ_D is *not meet-preserving* in general. Consider $D : \Delta; \Gamma \vdash (\mathbf{case}\ e\ \mathbf{of}\ x_1.e_1 \mid x_2.e_2) \Rightarrow \tau$ in which both branches independently synthesise the same type $\tau \sqsupset \square$, and take the two slices s_1 keeping only the first branch and s_2 keeping only the second. Both are defined, yielding $\phi_D(s_1) = \phi_D(s_2) = \tau$, but their meet $s_1 \sqcap s_2$ omits both branches, giving $\phi_D(s_1 \sqcap s_2) = \square \neq \tau = \phi_D(s_1) \sqcap \phi_D(s_2)$.

11.3 Type Error Slicing

Type error slicing, like type slicing, is a static analysis on a typing problem. Both share the broad goal of attributing type-level information back to source positions, but differ in language target, the underlying mathematical structure, and the scope.

Wand [50] and Haack and Wells [19] work in a globally inferred Hindley-Milner system; the program is reduced to a system of equality constraints over type variables. Wand's instrumented unification reports the constraint sites contributing to a failure, with no minimality guarantee; Haack and Wells compute a type error slice as a *minimal* unsatisfiable subset of the constraint set, projected back to the corresponding minimal subset of source positions. Skalpel [38] matures and scales this approach to Standard ML, and the Chameleon debugger [45] underlines the same minimal location sets interactively.

These slices are not structure-preserving programs but flat sets of constraints and sources, so the results cannot themselves be type checked or executed. Tip and Dinesh [47] are the exception: slicing explicitly typed languages by dependence tracking in a rewriting-based checker, their slices are programs guaranteed to reproduce the same error when type checked.

The scope also differs: type error slicing answers *"why does this program fail to type-check?"*, whereas type slicing answers *"why does this expression have this type?"*. Via marking (§7) [56], which elaborates every well-formed expression into a totally typed derivation, type slicing applies also to ill-typed programs. So, here, the question of "why this type?" subsumes "why this fails to type-check?".

11.4 Type-Directed Slicing

Nguyen et al. [27] recently independently introduced *type-directed slicing*, implemented for Java in the tool Specimin. Their motivation and application is more specific than ours: their slices serve the *maintainers of a typechecker*, reducing a large program exhibiting a known bug to a small test case reproducing it.

Given a target program element and a typechecker specified by its type rules, a type-directed slicer statically computes a smaller program on which the typechecker behaves identically at the target: essentially type slicing restricted to maximal queries (no partial queries). However, the approach, theory, and proofs differ greatly, as do the languages considered, with ours extending to bidirectionally inferred types, applying to programs with fewer annotations and better support for first-class functions. On the other hand, their work applies in practice to a wider range of analyses, including exceptions, crashes, and nullness analysis.

Their work was empirically evaluated on historical bugs of `javac`, NullAway [5], and the Checker Framework [31], suggesting our theory is similarly applicable to such real world use cases; conversely, partial queries may be a worthwhile extension to their tools, especially when a type checker produces multiple cascading bugs.

11.5 Type Debuggers & Type Explanation

Beyond slicing, there is a broad literature on explaining, localising, and repairing type errors. The oldest line *explains* types: Beaven and Stansifer [6] and Duggan and Bent [16] generate textual explanations from instrumented unification, Yang et al. [53] explain inferred polymorphic types of well-typed programs, and Chitil [14] navigates compositional explanation graphs via algorithmic debugging. These systems answer the same question as a synthesis slice—why does this term have this type?—but produce prose or inference traces, whereas a type slice is itself a minimal well-formed program that can be re-checked and further refined. Interactive debuggers similarly answer such queries: Chameleon [45] over a constraint store for Haskell, reporting errors as minimal location sets as in §11.3, and Argus [18] for Rust trait errors in the IDE.

A larger body of work improves error *reports* for globally inferred languages: constraint-solving heuristics in Helium [21], ranking likely error causes over constraint graphs [54, 55], minimum error sources via MaxSMT [32, 33], black-box correcting sets [26], subtyping-constraint flow messages [7], dynamic witnesses [41], and learned localisation [42] and repair [40]. These techniques answer "where is the error?" or "how might it be fixed?" rather than "why this type?"; empirically, most type errors implicate multiple program points [52], which type slicing surfaces directly.

As slices are themselves programs, these tools compose with type slicing (§1).

12 Further Extensions & Applications

12.1 Set-Theoretic Types

Type slicing looks especially promising for common gradual type systems in which *control flow and types are intertwined*: an expression's type depends non-trivially on the branches inside a function, or the control-flow path in which the expression lies. Explaining a type then genuinely requires considering the branches within the code, rather than just annotations and literals; as type slices necessarily *retain* program structure, they can retain the appropriate control flow structure.

Set theoretic types extend the type language with union, intersection, and negation, letting a term carry several types simultaneously, typing overloaded functions and typecase constructs common in real world hybrid typed languages (e.g. TypeScript, or Elixir [10]). Relevant foundations, including graduality proofs, have been explored [2, 11, 12], ready for type slicing to build upon.[10]

[10] After adapting into a bidirectional setting with expression holes.

To demonstrate this application, consider generalising sum types of the core calculus (§3) to allow sum injections to synthesise partially dynamic types: when $\Delta; \Gamma \vdash e \Rightarrow \tau$, let $\Delta; \Gamma \vdash \iota_1\ e \Rightarrow \tau + \square$ and $\Delta; \Gamma \vdash \iota_2\ e \Rightarrow \square + \tau$. Then an if statement $pred \triangleq \textbf{if}\ x > 0\ \textbf{then}\ \iota_1(x-1)\ \textbf{else}\ \iota_2(())$ synthesises a sum type $\mathbb{N} + 1$ without requiring annotations, much as union type inference works (i.e. union type $\mathbb{N} \mid 1$), and type slicing can explain where each component comes from: querying $\square + 1$ highlights exactly the failing branch, and $\mathbb{N} + \square$ the succeeding one. We mechanised this example.

12.2 Occurrence Typing

Occurrence typing [48] is another common technique in hybrid type systems. Inside the true branch of `if (typeof x = int) ... else ...`, the type of `x` is refined as an `int`. These explicitly require exploring the surrounding control flow, which can be extensive, to reason about the types. As type slices necessarily retain program structure, they can provide a reduced slice with minimal control flow to reason within. Formally applying type slicing to such systems will, however, need extensive integration work with bidirectional typing and static graduality.

12.3 Implicit Polymorphism

Globally inferred languages have much more confusing typing derivations and errors than locally inferred ones, so extending the calculus to implicit polymorphism would expand the application significantly. Explaining even well-typed code is particularly useful here: error marks are often mis-localised, leaving the real error in well-typed code whose types conflict with the programmer's expectation. Adapting the constraint slicing of Wand [50] and Haack and Wells [19] to integrate with type slicing is worth exploring.

12.4 Cast Slicing & Dynamics

A gradually-typed language elaborates programs into a cast calculus and runs the casts dynamically. Type slices can be attached to those casts, letting casts explain their insertion by pointing back to code: the source is a slice of the term being cast, the target a slice of its context.

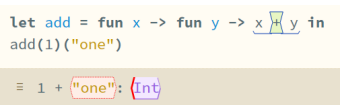


Fig. 13. Type slice associated with a cast error, from a preliminary workshop paper [9] built on differing foundations [8].

This is useful for debugging dynamic type errors, which would not otherwise point back to code: gradual typing's blame [49] identifies *which* cast failed but implicates only the cast site; a slice widens this to every source fragment contributing to either side.

However, slices are no longer slices of the original program once dynamic reductions are performed. Tracking dynamic execution, perhaps integrating ideas from Perera [34, 35], would allow tracing casts back to source with stronger guarantees; dynamic slices measurably aid diagnosis of run-time type errors in gradual languages [4].

13 Conclusions

This paper formulated type slicing theory for a bidirectional calculus with explicit System F polymorphism, bindings, products, and sums, mechanised in Agda, and formally integrated it with the error-marking calculus of Zhao et al. [56], applying it to ill-typed programs and justifying its *sound* use in debugging *all* static type errors. This positions type slicing as a promising tool for practical languages, especially gradually typed ones, explaining both static and, in follow-up work, *dynamic* types.

References

[1] [n. d.]. Hazel: A Live Functional Programming Environment with Typed Holes. https://hazel.org. Accessed 2026-07-08.

[2] Pedro Jorge Fernandes Ângelo. 2024. *Gradual Intersection Types*. Ph. D. Dissertation. Faculdade de Ciências, Universidade do Porto.

[3] Robert Atkey and Roly Perera. 2025. Galois Slicing as Automatic Differentiation. arXiv preprint arXiv:2511.09203.

[4] Felipe Bañados Schwerter, Ronald Garcia, Reid Holmes, and Karim Ali. 2025. Dynamic Program Slices Change How Developers Diagnose Gradual Run-Time Type Errors. *The Art, Science, and Engineering of Programming* 10, 1, Article 8 (2025).

[5] Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. 2019. NullAway: Practical Type-Based Null Safety for Java. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 740–750. doi:10.1145/3338906.3338919

[6] Mike Beaven and Ryan Stansifer. 1993. Explaining Type Errors in Polymorphic Languages. *ACM Letters on Programming Languages and Systems* 2, 1–4 (1993), 17–30. doi:10.1145/176454.176460

[7] Ishan Bhanuka, Lionel Parreaux, David Binder, and Jonathan Immanuel Brachthäuser. 2023. Getting into the Flow: Towards Better Type Error Messages for Constraint-Based Type Inference. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2, Article 237 (2023). doi:10.1145/3622812

[8] Max Carroll. 2025. Decomposable Type Highlighting for Bidirectional Type Slicing. Undergraduate dissertation, University of Cambridge.

[9] Max Carroll, Anil Madhavapeddy, and Patrick Ferris. 2025. Decomposable Type Highlighting for Bidirectional Type and Cast Systems. Submitted to the HATRA workshop, SPLASH.

[10] Giuseppe Castagna, Guillaume Duboc, and José Valim. 2024. The Design Principles of the Elixir Type System. *The Art, Science, and Engineering of Programming* 8, 2 (2024), 4:1–4:34. doi:10.22152/programming-journal.org/2024/8/4

[11] Giuseppe Castagna and Victor Lanvin. 2017. Gradual Typing with Union and Intersection Types. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 41:1–41:28. doi:10.1145/3110285

[12] Giuseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G. Siek. 2019. Gradual Typing: A New Perspective. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 16:1–16:32. doi:10.1145/3290329

[13] Sheng Chen and Martin Erwig. 2014. Counter-Factual Typing for Debugging Type Errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 583–594. doi:10.1145/2535838.2535863

[14] Olaf Chitil. 2001. Compositional Explanation of Types and Algorithmic Debugging of Type Errors. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 193–204. doi:10.1145/507635.507659

[15] Brian A. Davey and Hilary A. Priestley. 2002. *Introduction to Lattices and Order* (2nd ed.). Cambridge University Press. doi:10.1017/CBO9780511809088

[16] Dominic Duggan and Frederick Bent. 1996. Explaining Type Inference. *Science of Computer Programming* 27, 1 (1996), 37–83. doi:10.1016/0167-6423(95)00007-0

[17] Jana Dunfield and Neelakantan R. Krishnaswami. 2022. Bidirectional Typing. *Comput. Surveys* 54, 5 (2022), 98:1–98:38. doi:10.1145/3450952

[18] Gavin Gray, Will Crichton, and Shriram Krishnamurthi. 2025. An Interactive Debugger for Rust Trait Errors. *Proceedings of the ACM on Programming Languages* 9, PLDI (2025). doi:10.1145/3729302

[19] Christian Haack and J. B. Wells. 2003. Type Error Slicing in Implicitly Typed Higher-Order Languages. In *Programming Languages and Systems (ESOP) (LNCS, Vol. 2618)*. Springer, 284–301. doi:10.1007/3-540-36575-3_20

[20] Robert Harper. 2016. *Practical Foundations for Programming Languages* (2 ed.). Cambridge University Press. doi:10.1017/cbo9781316576892

[21] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. 2003. Scripting the Type Inference Process. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 3–13. doi:10.1145/944705.944707

[22] Gérard Huet. 1997. The Zipper. *Journal of Functional Programming* 7, 5 (1997), 549–554. doi:10.1017/S0956796897002864

[23] Amy J. Ko and Brad A. Myers. 2008. Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*. ACM, 301–310. doi:10.1145/1368088.1368130

[24] Bogdan Korel and Janusz Laski. 1988. Dynamic Program Slicing. *Inform. Process. Lett.* 29, 3 (1988), 155–163. doi:10.1016/0020-0190(88)90054-3

[25] Benjamin S. Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. 2007. Searching for Type-Error Messages. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 425–434. doi:10.1145/1250734.1250783

[26] Calvin Loncaric, Satish Chandra, Cole Schlesinger, and Manu Sridharan. 2016. A Practical Framework for Type Inference Error Explanation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 781–799. doi:10.1145/2983990.2983994

[27] Loi Ngo Duc Nguyen, Tahiatul Islam, Theron Wang, Sam Lenz, and Martin Kellogg. 2025. Static Program Reduction via Type-Directed Slicing. *Proceedings of the ACM on Software Engineering* 2, ISSTA, Article ISSTA091 (2025). doi:10.1145/3728968

[28] 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. doi:10.1145/3290327

[29] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. 2017. Hazelnut: A Bidirectional Type System with Holes. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM, 86–99. doi:10.1145/3009837.3009900

[30] John-Paul Ore, Sebastian Elbaum, Carrick Detweiler, and Lambros Karkazis. 2021. An Empirical Study on Type Annotations: Accuracy, Speed, and Suggestion Effectiveness. *ACM Transactions on Software Engineering and Methodology* 30, 2, Article 20 (2021). doi:10.1145/3439775

[31] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa, Jr., Jeff H. Perkins, and Michael D. Ernst. 2008. Practical Pluggable Types for Java. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA)*. 201–212. doi:10.1145/1390630.1390656

[32] Zvonimir Pavlinovic, Tim King, and Thomas Wies. 2014. Finding Minimum Type Error Sources. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. 525–542. doi:10.1145/2660193.2660230

[33] Zvonimir Pavlinovic, Tim King, and Thomas Wies. 2015. Practical SMT-Based Type Error Localization. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 412–423. doi:10.1145/2784731.2784765

[34] Roly Perera, Umut A. Acar, James Cheney, and Paul Blain Levy. 2012. Functional Programs that Explain Their Work. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 365–376. doi:10.1145/2364527.2364579

[35] Roly Perera, Deepak Garg, and James Cheney. 2016. Causally Consistent Dynamic Slicing. In *27th International Conference on Concurrency Theory (CONCUR) (LIPIcs, Vol. 59)*. 18:1–18:15. doi:10.4230/LIPIcs.CONCUR.2016.18

[36] Roly Perera, Minh Nguyen, Tomas Petricek, and Meng Wang. 2022. Linked Visualisations via Galois Dependencies. *Proceedings of the ACM on Programming Languages* 6, POPL, Article 8 (2022). doi:10.1145/3498668

[37] Benjamin C. Pierce and David N. Turner. 2000. Local type inference. *ACM Transactions on Programming Languages and Systems* 22, 1 (Jan. 2000), 1–44. doi:10.1145/345099.345100

[38] Vincent Rahli, Joe B. Wells, John Pirie, and Fairouz Kamareddine. 2017. Skalpel: A Constraint-Based Type Error Slicer for Standard ML. *Journal of Symbolic Computation* 80 (2017), 164–208. doi:10.1016/j.jsc.2016.07.013

[39] Wilmer Ricciotti, Jan Stolarek, Roly Perera, and James Cheney. 2017. Imperative Functional Programs that Explain Their Work. *Proceedings of the ACM on Programming Languages* 1, ICFP, Article 14 (2017). doi:10.1145/3110258

[40] Georgios Sakkas, Madeline Endres, Benjamin Cosman, Westley Weimer, and Ranjit Jhala. 2020. Type Error Feedback via Analytic Program Repair. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/3385412.3386005

[41] Eric L. Seidel, Ranjit Jhala, and Westley Weimer. 2016. 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)*. 228–242. doi:10.1145/2951913.2951915

[42] Eric L. Seidel, Huma Sibghat, Kamalika Chaudhuri, Westley Weimer, and Ranjit Jhala. 2017. Learning to Blame: Localizing Novice Type Errors with Data-Driven Diagnosis. *Proceedings of the ACM on Programming Languages* 1, OOPSLA, Article 60 (2017). doi:10.1145/3138818

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

[44] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (LIPIcs, Vol. 32)*. 274–293. doi:10.4230/LIPIcs.SNAPL.2015.274

[45] Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. 2003. Interactive Type Debugging in Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*. 72–83. doi:10.1145/871895.871903

[46] Frank Tip. 1995. A Survey of Program Slicing Techniques. *Journal of Programming Languages* 3, 3 (1995), 121–189.

[47] Frank Tip and T. B. Dinesh. 2001. A Slicing-Based Approach for Locating Type Errors. *ACM Transactions on Software Engineering and Methodology* 10, 1 (2001), 5–55. doi:10.1145/366378.366379

[48] Sam Tobin-Hochstadt and Matthias Felleisen. 2010. Logical Types for Untyped Languages. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 117–128. doi:10.1145/1863543.1863561

[49] Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can't Be Blamed. In *Programming Languages and Systems (ESOP) (Lecture Notes in Computer Science, Vol. 5502)*. 1–16. doi:10.1007/978-3-642-00590-9_1

[50] Mitchell Wand. 1986. Finding the Source of Type Errors. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*. 38–43. doi:10.1145/512644.512648

[51] Mark Weiser. 1984. Program Slicing. *IEEE Transactions on Software Engineering* SE-10, 4 (1984), 352–357. doi:10.1109/TSE.1984.5010248

[52] Baijun Wu and Sheng Chen. 2017. How Type Errors Were Fixed and What Students Did? *Proceedings of the ACM on Programming Languages* 1, OOPSLA, Article 105 (2017). doi:10.1145/3133929

[53] Jun Yang, Greg Michaelson, and Phil Trinder. 2002. Explaining Polymorphic Types. *Comput. J.* 45, 4 (2002), 436–452. doi:10.1093/comjnl/45.4.436

[54] Danfeng Zhang and Andrew C. Myers. 2014. Toward General Diagnosis of Static Errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 569–582. doi:10.1145/2535838.2535870

[55] Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis, and Simon Peyton Jones. 2015. Diagnosing Type Errors with Class. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 12–21. doi:10.1145/2737924.2738009

[56] Eric Zhao, Raef Maroof, Anand Dukkipati, Andrew Blinn, Zhiyi Pan, and Cyrus Omar. 2024. Total Type Error Localization and Recovery with Holes. *Proceedings of the ACM on Programming Languages* 8, POPL, Article 68 (2024). doi:10.1145/3632910