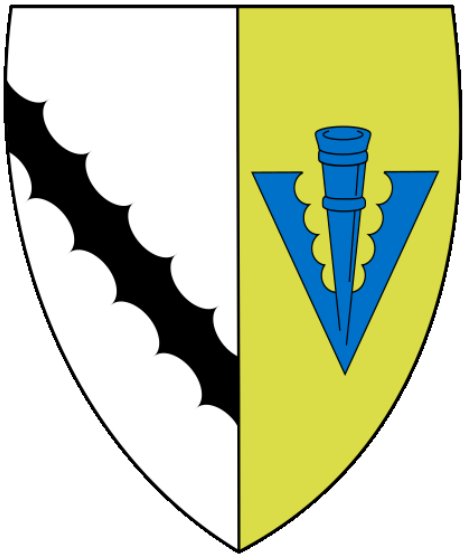
Max Carroll

Polymorphic Type Slicing

Computer Science Tripos, Part III



Sidney Sussex College
University of Cambridge
May 21, 2026

A dissertation submitted to the University of Cambridge in partial fulfilment for a Master of Engineering in Computer Science

Total page count: 62
Main chapters (excluding front-matter, references and appendix): 49 pages (pp 5–53)
Main chapters word count: 11936
Methodology used to generate that word count:

```
$ texcount -inc -sum=1,1,1,1,0,0,0 -0 PolymorphicTypeSlicing.tex
```

The frontmatter, bibliography, and appendix are excluded.

Declaration

I, Max Carroll of Sidney Sussex College, being a candidate for Part III of the Computer Science Tripos, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this report, I adhered to the Department of Computer Science and Technology AI Policy. I am content for my report to be made available to the students and staff of the University. AI tools were used to formulaically refactor *existing* mechanised proofs from the type lattice to the expression and context lattices. Some of the proof mechanisation is partially based on existing mechanisations; these are referenced accordingly by 'Mechanisation Provenance' notes in the dissertation body.

Signed: Max Carroll

Date: May 21, 2026

Abstract

This dissertation formalises and mechanises *type slicing* – a technique that incrementally explains program types by highlighting minimal program fragments. Using this, a programmer can select any sub-term of a program and incrementally explore complex types and concisely explain type errors. Crucially, this includes highlighting context *surrounding* the term, differing form typical type error highlighting which simply states a term has the wrong type, but not what about the surrounding contexts explains *why* it has this type.

I formalise this in a bidirectional, gradually-typed core calculus with explicit System F style polymorphism, products, and sum types. Additionally, a novel *context classification* judgement to formalise the notion of splitting type information into that derived from a selected sub-term and that coming from the surrounding context, which can be applied to arbitrary bidirectional type systems. Further, the theory extends formally to ill-typed programs with error marks building upon the marked lambda calculus of Zhao et al.

Mechanisation legend: ◯ fully mechanised, ◬ mechanised with postulates, △ not mechanised.

Contents

1 Introduction

Programmers report that debugging and testing consumes 20–60% of their development time [4], with a small fraction of difficult bugs absorbing a disproportionate share of that effort [1]. In statically typed languages, many such bugs are caught statically at compile time by *type errors*.

These type errors are conventionally reported at a single location, yet most errors genuinely involve more than one source location [36]. The reported location is often imprecise in *two* directions: often including code that does *not* directly contribute to the error, yet at the same time misses regions of the surrounding context that *do* (for example, non-local type information sourced from bindings). As a result, existing highlighting systems show both too much and too little context to the programmer for any given error. The error itself states only that the term has a differing type that was *expected*, not *why* the system expected it.

This dissertation develops the theory of *type slicing*: a decomposable highlighting system that allows terms to be selected and the type explained in a *bidirectional* locally inferred type system. A programmer selects an expression and queries a part of the type, v, that they wish to explain. The explanation picks out a minimal portion of the program that suffices to justify the queried type.

A *synthesis slice* (section 4) of a selected term answers "which parts of the term caused it to have type v?"; an *analysis slice* (section 5) of a selected term answers the complementary question, "which parts of the surrounding program enforced that this term have type v?".

Slices can be *refined*: refining a query, say from an entire function type to just its codomain, gives a smaller highlighted region for the refined query, supporting interactive exploration complex types, where the programmer can incrementally focus into sub-parts of the type that are of interest. For example, maybe only a small region of a large type involed in a type error is erroneous.

The type slicing machinery applies even beyond errors (section 6); it is possible to query any program regions responsible for any type information of interest, even well-typed code. Hence, type slicing is a tool to explain *any* types in a program.

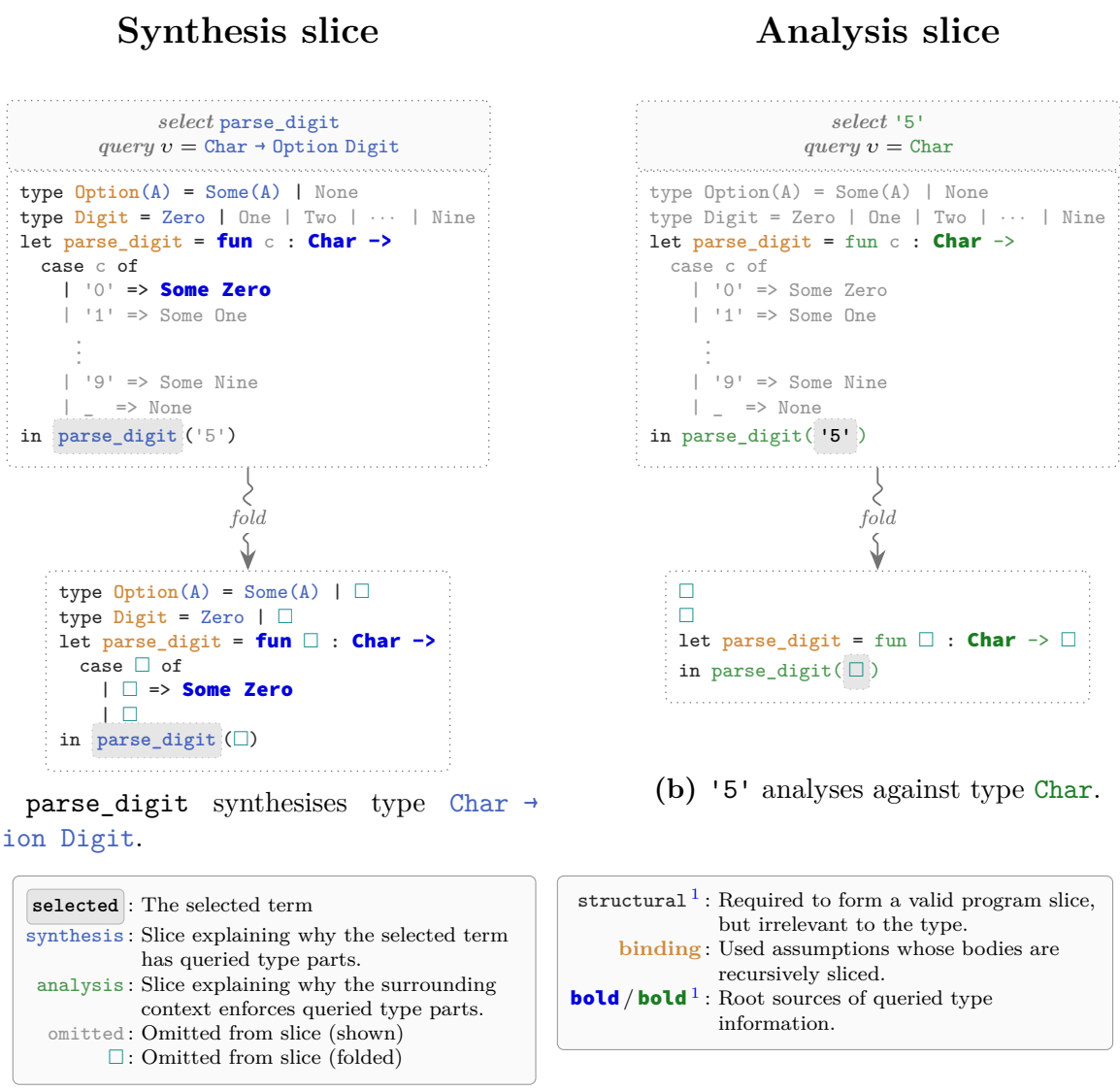
1.1 Explanation by Example

This section previews the slicing apparatus formally introduced in later chapters through three worked examples.

Figure 1 demonstrates a small example of a function parsing characters to single digits. In **(a)** the function is queried for its *synthesised* type, `Char → Option Digit`, which is explained by just taking a single branch from the function, and the corresponding part of the `Digit` type.

Secondly, **(b)** asks why the argument is *required* to be a `Char`, explained by an *analysis* slice retaining just the function application and the `Char` annotation on the function.

The formal mathematics of these two concepts are introduced in sections 4 and 5 explored upon a core calculus specified in section 3 which includes sum types, products, and explicit polymorphism.



(a) `parse_digit` synthesises type `Char → Option Digit`.

(b) `'5'` analyses against type `Char`.

`selected` : The selected term
`synthesis` : Slice explaining why the selected term has queried type parts.
`analysis` : Slice explaining why the surrounding context enforces queried type parts.
`omitted` : Omitted from slice (shown)
□ : Omitted from slice (folded)

`structural`[1] : Required to form a valid program slice, but irrelevant to the type.
`binding` : Used assumptions whose bodies are recursively sliced.
bold / **bold**[1] : Root sources of queried type information.

Figure 1: Parsing a character into an `Option Digit`.

[1] The distinction between purely structural parts of a slice and bolded type sources is not formalised, but is easy to implement in practice by tagging the leaves of the derivations during slicing.

These slices can be queried on *refined* types, investigating only a subpart of the full type. Figure 2 demonstrates the power of this by debugging a static type error: it defines a sequencing (`seq`) parser combinator to string together parsers sequentially, used to parse 4-digit pins.

(1) The `seq` combinator is implemented incorrectly: the annotation and usage of the `f` argument forgets to take the remaining unparsed string `s'`. However, as this function is well-typed (but incorrect according to the programmers mental model), the error is localised to the passed argument at the usage of `seq`.

(2) The synthesis slice explains the type of the term in the marked error.

(2b) We can *refine* the synthesis slice to only show the *outermost inconsistency* in the type: that the argument provided is a curried function.

The sequence combinator should indeed take a two argument function, so this is correct. We instead look to the analysis slice for the error.

But, in general, if the error were inside the error squiggle, this slice shows that the squiggle highlights *more* than required to demonstrate the outermost error (the function itself, not the terms within).

(2) The analysis slice explains the expected type, which is derived from the *incorrect annotation*.

The original error squiggle does not alert the programmer that the annotation itself could have been incorrect, despite this being a common occurrence (49% of the time [21]), at least when annotating existing code, which is common in hybrid static-dynamic typed language.

(3b) The refined analysis slice refines the explanation to just the single argument function to *some* option type, which is what actually causes the conflict.

(4) Because the slices do not *arbitrarily* blame the usage site like the error mark does, we inspect the refined analysis slice to locate and fix the error in the annotation.

After fixing the annotation, a secondary error at the application of `f` will be unveiled and correctly localised by the standard method.

This situation could arise naturally after refactoring the argument passed into `seq` to take an additional string argument. Here, the analysis slice points you directly to the corresponding annotation that must be changed to perform the refactor.

Section 6 concerns the formal theory applying type slicing to ill-typed programs with error marks, allowing for debugging of static type errors.

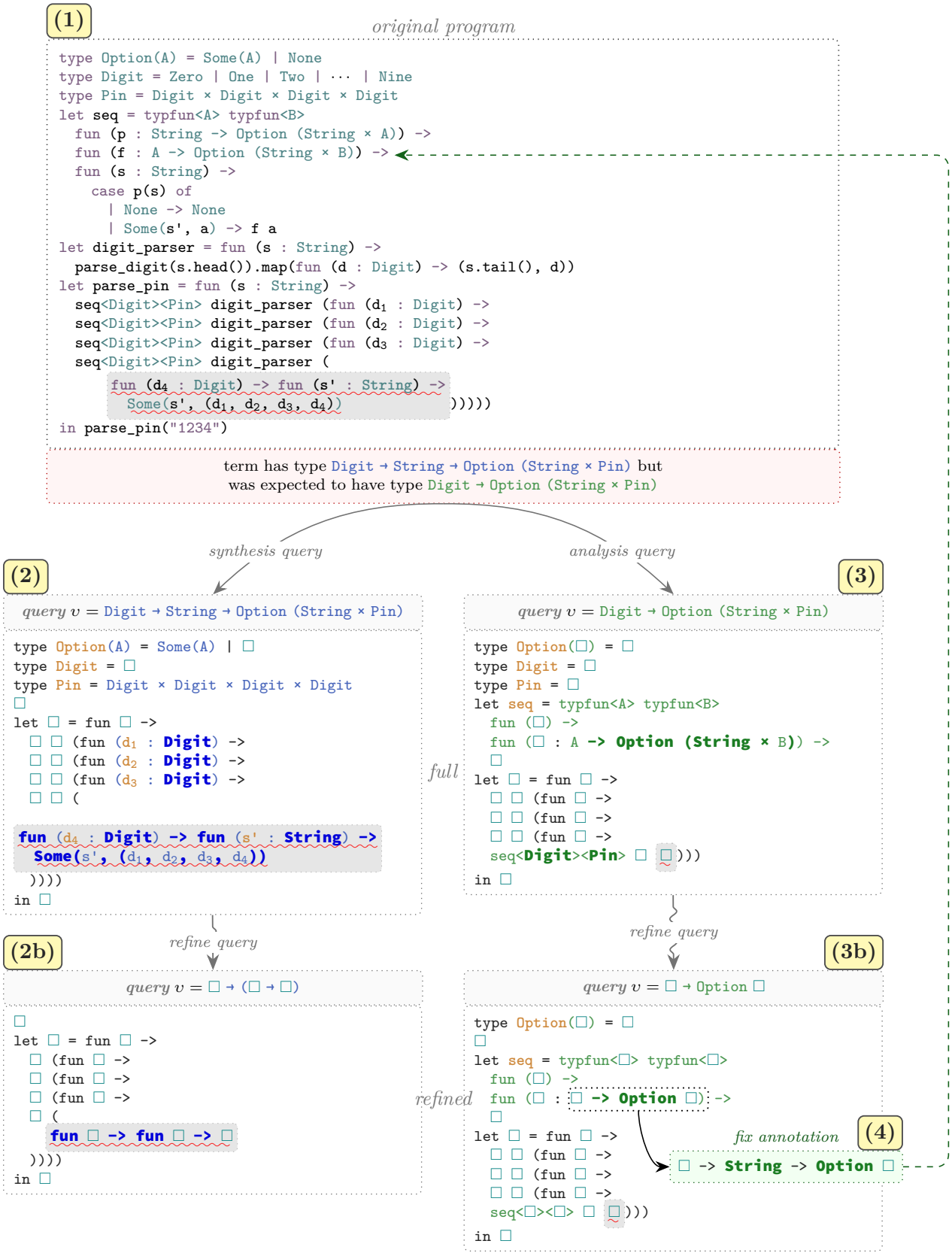


Figure 2: Debugging a static type error in a parser for 4 digit PINs.

Figure 3 demonstrates the power of type slicing when using *gradual types* by parsing a character to a digit, and otherwise falling back to a string, i.e. `Char → Digit + String`. Using the dynamic type we can infer an unannotated injection `Left(a)` into a sum type with unknown right injection `(A + ☐)` and `Right(b)` at `☐ + B`.

Therefore, in an if statement we can join two injections `(A + ☐) ⊔ (☐ + B)` to determine a full static type `A + B`, with the slices identifying the branch from which each type was inferred. No sum type annotations were required. This behaviour is similar to union type inference, which is crucial in languages like TypeScript.

Importantly, we can use refined slices to query where each part of the sum type came from. In the example we find that the `Digit` came from the `then` branch, and the `String` from the right branch.

These situations are common in advanced type systems with set-theoretic types, where types are combined non-trivially by code branches. The example shows promise that type slicing could be especially applicable in explaining components of these set types by identifying the branches from which each type component is sourced. However, the current theory only explores combining dynamic code in branches and retrieving the corresopnding static regions from the appropriate branches.

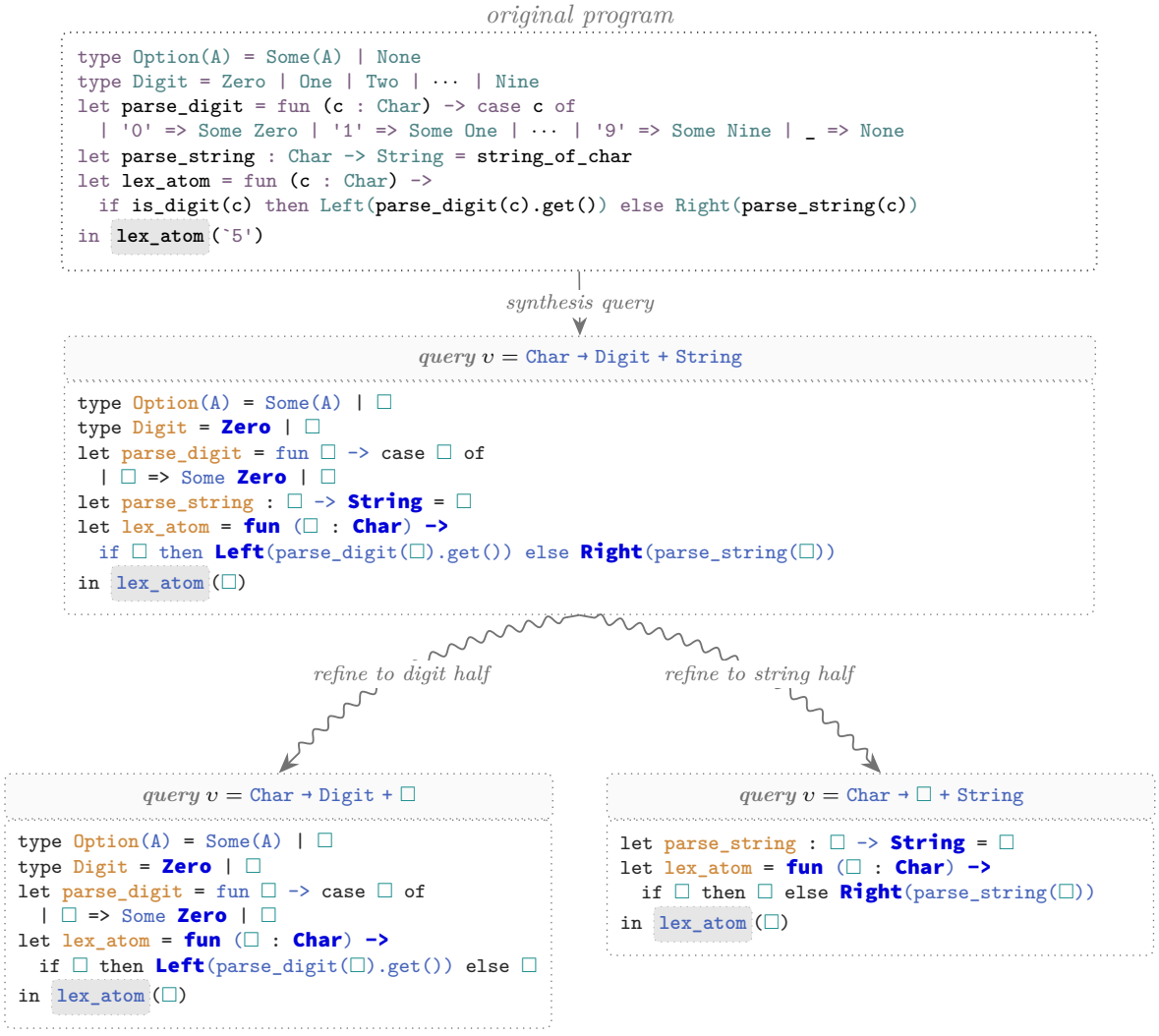


Figure 3: Lexing a character into a digit or string token.

2 Background

In this chapter I introduce the key concepts underlying the core calculus formalised: bidirectional typing, explicit polymorphism (System F), and gradual typing. I assume familiarity with standard type systems expressed via judgements, the simply-typed lambda calculus, and basic set notation.

2.1 Bidirectional Types

Type slicing is formalised for *bidirectional type systems* [14]. 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 [26] resulting in annotations only at redexes (for *most*[2] expressions).[3]

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):

$$\Delta;\, \Gamma \vdash e \Uparrow \tau$$

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

$$\Delta;\, \Gamma \vdash e \Downarrow \tau$$

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

When designing such a system, care must be taken to ensure *mode correctness* [7]: an input must never need to be *guessed*. For example, the following function application rule is *not* mode correct:

$$\frac{\Delta;\, \Gamma \vdash e_1 \Downarrow \tau_1 \to \tau_2 \qquad \Delta;\, \Gamma \vdash e_2 \Downarrow \tau_1}{\Delta;\, \Gamma \vdash e_1(e_2) \Downarrow \tau_2}$$

Here I try to check e_2 with input τ_1, which is not known from either an output of any premise or from the input to the conclusion τ_2. On the other hand, the following *is* mode correct:

$$\frac{\Delta;\, \Gamma \vdash e_1 \Uparrow \tau_1 \to \tau_2 \qquad \Delta;\, \Gamma \vdash e_2 \Downarrow \tau_1}{\Delta;\, \Gamma \vdash e_1(e_2) \Uparrow \tau_2}$$

Now τ_1 is known, being synthesised from the first premise. We never have to "guess" any types.

Such languages have two fundamental rules:

$$\text{SVar}\frac{x : \tau \in \Gamma}{\Delta;\, \Gamma \vdash x \Uparrow \tau} \qquad \text{Sub}\frac{\Delta;\, \Gamma \vdash e \Uparrow \tau}{\Delta;\, \Gamma \vdash e \Downarrow \tau}$$

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] Not always in the presence of binding elimination forms, e.g. case expressions. See section 4.5.4 of [14].
[3] i.e. a function application must use an annotated function.

2.2 Example: System F

System F [25, ch. 23] extends the simply-typed lambda calculus with *parametric polymorphism*: type functions $\Lambda\alpha.\ e$, universal types $\forall\alpha.\ \tau$, and type application $e\,\langle\sigma\rangle$. A polymorphic function like the identity $\Lambda\alpha.\ \lambda x : \alpha.\ x$ has type $\forall\alpha.\ \alpha \Rightarrow \alpha$ and can be instantiated at any type.

System F can be expression within the bidirectional framework:

$$\text{S}\Lambda\ \frac{\Delta;\ \Gamma \vdash e \Uparrow \tau}{\Delta;\ \Gamma \vdash \Lambda\alpha.\ e \Uparrow \forall\alpha.\ \tau} \qquad \text{STApp}\ \frac{\Delta;\ \Gamma \vdash e \Uparrow \forall\alpha.\ \tau'}{\Delta;\ \Gamma \vdash e\,\langle\sigma\rangle \Uparrow [0 \mapsto \sigma], \tau'}$$

Type functions synthesis a $\forall$-type when the body synthesises under an extended type variable context which may now reference the newly bound type variable. Type application substitutes the bound type variable for a provided type argument.

2.3 Gradual Types

A *gradual type system* [30, 31] combines static and dynamic typing. Terms may be annotated with the dynamic type $\square$, marking regions of code omitted from static type-checking but still *interoperable* with statically typed code. For example, the following type-checks:

```
let x : □ = 10 in x ^ "str"
```

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

2.3.1 Consistency

This is enabled by a *consistency* relation $\tau_1 \sim \tau_2$, a weakening of type equality. Consistency replaces equality in the typing rules, allowing types to be used interchangeably when they are consistent. Every type is consistent with the dynamic type $\square$, and compound types are consistent when their sub-parts are:

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

However, consistency is *not transitive*: $\mathit{Int} \sim \square \sim \mathit{Bool}$ does not imply $\mathit{Int} \sim \mathit{Bool}$. This non-transitivity is essential – preventing the dynamic type from collapsing all type distinctions.

After type-checking, explicit *casts* are inserted into partially annotated programs by elaboration. Then, at runtime, type errors which were masked by dynamic-type annotations are caught as *cast errors*.

2.3.2 Graduality

A true gradual type system satisfies a *gradual guarantee* property, first specified by Siek et al. in their refined criteria [31]. To specify this we first need to introduce the notion of *type precision*, written $\tau \sqsubseteq \tau'$, which orders types by their dynamic content: $\square$ is the least precise type, and concrete types like $\mathit{Int} \to \mathit{Bool}$ are maximally precise, while compound types can be partially dynamic, e.g. $\square \Rightarrow \mathit{Bool}$.

The only relevant part of graduality to Type Slicing is the downwards direction of the static gradual guarantee.

Downwards graduality. If a well-typed program is made *less precise* (type annotations replaced by $\square$), it remains well-typed, possibly with a less precise type. In a bidirectional setting: if $\Delta; \Gamma \vdash e \Uparrow \tau$ and $\Gamma' \sqsubseteq \Gamma$, $e' \sqsubseteq e$, then $\exists \tau'.\, \Gamma' \vdash e' \Uparrow \tau'$ with $\tau' \sqsubseteq \tau$.

2.4 Lattice Background

The foundations for program highlighting in Type Slicing are built upon lattices, which I introduce here.

A *partial order* $(L, \sqsubseteq)$ is a set L equipped with a reflexive, transitive, and antisymmetric relation $\sqsubseteq$. For example, precision in gradual typing forms a partial order, with $a \sqsubseteq b$ intuitively meaning "b carries at least as much type information as a".

A *lattice* is a partial order in which every pair of elements in L have both a greatest lower bound (a *meet*) and a least upper bound (a *join*) under the partial order. A lattice is *bounded* if it has a bottom element $\bot$ ($\bot \sqsubseteq x$ for all x) and a top element $\top$ ($x \sqsubseteq \top$ for all x). A *meet-semilattice* is a partial order in which every pair has a meet but not necessarily a join; dually for a *join-semilattice*. A bounded lattice is *distributive* if $a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup (a \sqcap c)$ for all a, b, c.

The meet and join operations have natural interpretations under the precision interpretation:

- The *meet* $a \sqcap b$ is the most precise element that agrees with both a and b, retaining the type information they share (Figure 4a).
- The *join* $a \sqcup b$ is the least precise element that is still as precise as both a and b, combining all the type information present in either (Figure 4b).

Heyting algebras. A bounded distributive lattice is a *Heyting algebra* [5, 13] if it has an *implication* operation $a \to b$ satisfying the adjunction:

$$c \sqcap a \sqsubseteq b \quad \text{iff} \quad c \sqsubseteq (a \to b)$$

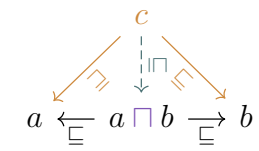
That is, $a \to b = \bigsqcup \{\, c \mid c \sqcap a \sqsubseteq b \,\}$ is the largest element whose meet with a is below b. The *pseudocomplement* of a is $\neg a = a \to \bot$. Again, when interpreting $\sqsubseteq$ as type precision, $a \to b$ intuitively means: "what is the most precise element such that, combined with a, the result is no more precise than b?" (Figure 4c.)

Co-Heyting algebras. Dually, a bounded distributive lattice is a *co-Heyting algebra* [27, 13] if it has a *subtraction* operation $a \setminus b$ satisfying:

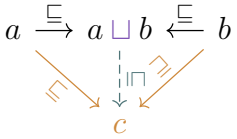
$$a \sqsubseteq b \sqcup c \quad \text{iff} \quad a \setminus b \sqsubseteq c$$

i.e. $a \setminus b = \bigsqcap \{\, c \mid a \sqsubseteq b \sqcup c \,\}$ is the smallest element whose join with b is above a; the *supplement* of a is $\sim a = \top \setminus a$. Reading $\sqsubseteq$ as type precision, $a \setminus b$ answers: "what type information in a is missing from b?" (Figure 4d.)

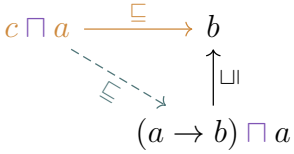
Bi-Heyting algebras. A *bi-Heyting algebra* [27, 28] is simultaneously a Heyting algebra and a co-Heyting algebra. Every *finite* bounded distributive lattice is a bi-Heyting algebra, because the required suprema and infima are over finite sets [13].



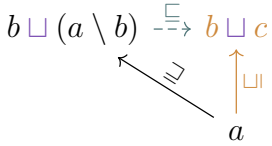
(a) Meet as product.



(b) Join as coproduct.



(c) Implication as exponential.



(d) Subtraction as co-exponential.

Figure 4: Lattice operations. Where assuming some relations (solid arrows) induces additional relations (dashed arrows).

Product lattices. Given two lattices $(L_1, \sqsubseteq_2)$ and $(L_2, \sqsubseteq_2)$, their *product* $L_1 \times L_2$ is a lattice defined componentwise: $(a_1, a_2) \sqsubseteq (b_1, b_2)$ iff $a_1 \sqsubseteq_1 b_1$ and $a_2 \sqsubseteq_2 b_2$ (Figure 5). Meets and joins are similarly defined componentwise. If both lattices are bounded, distributive, or bi-Heyting, so is the product.

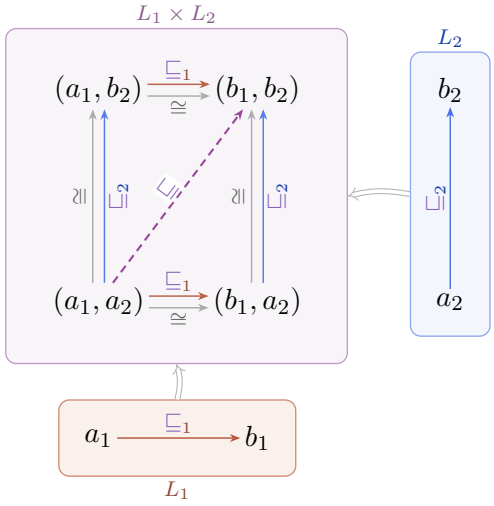


Figure 5: Product Lattice Precision

2.5 Well-Foundedness, Monotonicity, and Fixed Points

A partial order $(L, \sqsubseteq)$ is *well-founded* if it admits no infinite *strictly* descending chain $x_0 \sqsupset x_1 \sqsupset x_2 \sqsupset \cdots$. This is immediate on finite sets: every chain is bounded by the finite cardinality $|L|$.

A function $f : L \to L$ is *monotone* if $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$, and *anti-monotone* if $x \sqsubseteq y$ implies $f(y) \sqsubseteq f(x)$.

Lemma 2.1 (Monotonicity of co-Heyting subtraction). *In a co-Heyting algebra $(L, \sqsubseteq, \sqcup, \setminus)$, the binary operation $\setminus$ is monotone in its first argument and anti-monotone in its second: for all $a, a', b, b' \in L$,*

$$a \sqsubseteq a' \;\Rightarrow\; a \setminus b \sqsubseteq a' \setminus b, \qquad b \sqsubseteq b' \;\Rightarrow\; a \setminus b' \sqsubseteq a \setminus b.$$

Proof. By the universal property of $\setminus$, $x \setminus y \sqsubseteq z \iff x \sqsubseteq y \sqcup z$. *First argument:* take $z = a' \setminus b$. Then $a' \sqsubseteq b \sqcup z$, so $a \sqsubseteq a' \sqsubseteq b \sqcup z$, giving $a \setminus b \sqsubseteq z = a' \setminus b$. *Second argument:* take $z = a \setminus b$. Then $a \sqsubseteq b \sqcup z \sqsubseteq b' \sqcup z$, giving $a \setminus b' \sqsubseteq z = a \setminus b$. □

Theorem 2.2 (Kleene Fixed Point, finite case). *Let $(L, \sqsubseteq, \bot)$ be a finite partial order with bottom element $\bot$, and let $f : L \to L$ be monotone. Then f has a least fixed point $\mu f = f^n(\bot)$ for some $n \leq |L|$.*

Proof. The chain $\bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq \cdots$ is ascending by monotonicity. Since L is finite, this chain must stabilise: there is some $n \leq |L|$ with $f^n(\bot) = f^{n+1}(\bot)$. So $f^n(\bot)$ is a fixed point. Further, $f^n(\bot)$ is a least fixed point: let x be any fixed point of f. We show $f^k(\bot) \sqsubseteq x$ by induction on k. Base: $\bot \sqsubseteq x$ since $\bot$ is the bottom. Step: $f^{k+1}(\bot) = f(f^k(\bot)) \sqsubseteq f(x) = x$ by monotonicity of f and the induction hypothesis. □

$$\bot \xrightarrow{\sqsubseteq} f(\bot) \xrightarrow{\sqsubseteq} f^2(\bot) \longrightarrow \cdots \xrightarrow{=} \mu f$$

Figure 6: Kleene iteration to the least fixed point μf.

3 The Core Calculus

Legend: ◯ fully mechanised, ◬ mechanised with postulates, △ not mechanised, ▭ important theorem, ▭ counterexample.

In this chapter I present the core calculus which Type Slicing builds upon. It is a bidirectional, gradually typed lambda calculus based on the Hazel calculus [20], extended with products, sums, and explicit System F polymorphism. Then, upon this, I define and prove several bi-Heyting lattices under a precision relation.

Core/*/Base

3.1 ◯ Syntax & Relations

The syntax of the core calculus is given in Figure 7. $*$ 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.

Syntax Types τ and expressions e

$$\tau ::= \square \mid * \mid \tau \Rightarrow \tau \mid \tau \times \tau \mid \tau + \tau \mid \forall\alpha.\ \tau \mid \alpha$$

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

Derived forms Definable in terms of the core syntax

$$(e : \tau) \triangleq (\lambda x : \tau.\ x)(e) \qquad \texttt{Bool} \triangleq * + *$$
$$\texttt{true} \triangleq \iota_1\ * \qquad \texttt{false} \triangleq \iota_2\ *$$
$$\textbf{if}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \triangleq (\textbf{case}\ e\ \textbf{of}\ x.\ e_1 \mid y.\ e_2) \qquad (x \notin \mathrm{FV}(e_1),\ y \notin \mathrm{FV}(e_2))$$

Figure 7: Syntax of the core calculus up to α-equivalence (on variables x). $\mathrm{FV}(\cdot)$ denotes the set of free variables of an expression.

Core/Typ/Consistency

3.1.1 ◯ Consistency

Type consistency (Figure 8) is exactly as presented in standard gradual type systems (section 2.3). 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.

$\tau_1 \sim \tau_2$ τ_1 is consistent with τ_2

$$\sim\square_1\frac{}{\tau \sim \square} \qquad \sim\square_2\frac{}{\square \sim \tau} \qquad \sim *\frac{}{* \sim *} \qquad \sim\Rightarrow\frac{\tau_1 \sim \tau_1' \qquad \tau_2 \sim \tau_2'}{\tau_1 \Rightarrow \tau_2 \sim \tau_1' \Rightarrow \tau_2'}$$

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

Figure 8: Type consistency.

Core/*/ Precision

3.1.2 ◯ Precision

Precision (Figure 9) 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. Figure 10 illustrates this with a small example.

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

$$\sqsubseteq \square \frac{}{\square \sqsubseteq \tau} \qquad \sqsubseteq * \frac{}{* \sqsubseteq *} \qquad \sqsubseteq \alpha \frac{}{\alpha \sqsubseteq \alpha} \qquad \sqsubseteq \Rightarrow \frac{\tau_1 \sqsubseteq \tau_1' \quad \tau_2 \sqsubseteq \tau_2'}{\tau_1 \Rightarrow \tau_2 \sqsubseteq \tau_1' \Rightarrow \tau_2'}$$

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

Figure 9: Type precision.

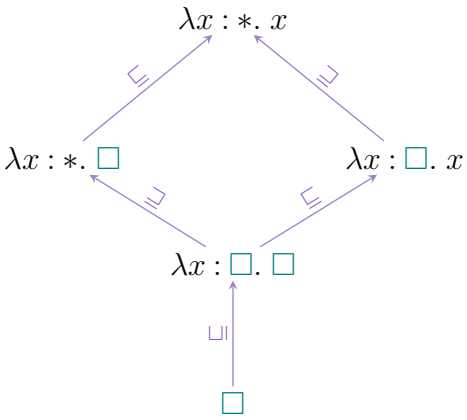


Figure 10: Hasse diagram of slices less precise than $\lambda x{:}*\,.\,x$, with each edge directed from a less-precise slice up to a more-precise slice.

Core/Assms/

Precision on Typing Assumptions. In this calculus, I 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 (Assumptions vs Contexts Terminology). Typical literature refers to these *typing assumptions* as 'typing contexts'. I avoid this terminology as I use the *context* to refer to the syntactic context around a subterm, i.e. in the same sense as used in contextual dynamics (Harper [16, Chapter 5]).

Strictly descending on successively less precise terms will always terminate at an 'empty' term in finite time. This is a useful property for calculating type slices (theorem 4.6).

Proposition 3.2 (△ Downward Well-Foundedness of Precision). *The strict precision ordering $\sqsubset$ is downward well-founded on types, expressions, and assumptions. Each family has a bottom element under precision:*

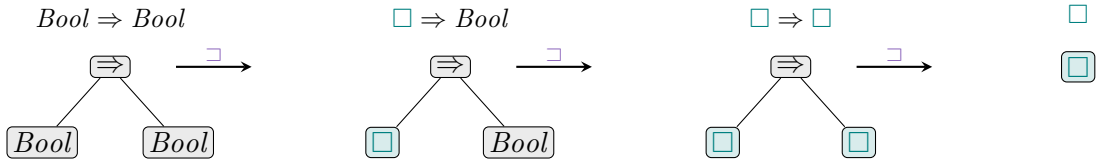


Figure 11: A maximal descending chain in $\lfloor Bool \Rightarrow Bool \rfloor$.

- *Types and expressions:* $\bot = \square$.
- *Assumptions:* $\bot = \emptyset$.

Proof. Each object is a finite tree over a finite alphabet of constructors. A strict descent in precision must replace at least one non-$\square$ position with $\square$, as illustrated in Figure 11. Since there are finitely many positions, any descending chain has length bounded by the number of positions in the original term. And for assumptions, the finiteness of $\mathrm{dom}(\Gamma)$ bounds the number of variables available. □

Core/*/ Lattice

3.2 ◯ Lattice Properties

3.2.1 ◯ Meets and Joins

The *meet* $\tau_1 \sqcap \tau_2$ is the greatest lower bound (GLB) of two types under precision. It is defined inductively: types of the same 'kind'[4] take meets of their components, and $\square$ is an *annihilator* ($\square \sqcap \tau = \square$). The meet is *always* defined: when the outermost kinds differ, $\tau_1 \sqcap \tau_2 = \square$ is a valid (and greatest) lower bound. Since all all pairs have a meet, we have that types form a *meet-semilattice* with lower bound $\square$(Section 2.4). Expression meet is defined analogously. Assumption meets are defined pointwise and by intersecting the domains.

The *join* $\tau_1 \sqcup \tau_2$ is the least upper bound (LUB). It is similarly defined inductively and componentwise, but with $\square$ acting as the *identity* ($\square \sqcup \tau = \tau$, $\tau \sqcup \square = \tau$). However, unlike meets, the join is only defined for consistent types (i.e. types of the same kind): when τ_1 and τ_2 have different kinds, no upper bound exists.

Remark 3.3. One might hope to quotient types by consistency and form a join-semilattice. But, this fails because consistency is *not transitive*: $Int \sim \square \sim \texttt{Bool}$ does not imply $Int \sim \texttt{Bool}$.

Core/Instances

3.2.2 ◯ The Slice Lattice

For a fixed type τ, the *slice lattice* of τ is the *lower set* under precision:

$$\lfloor \tau \rfloor \;=\; \{\, \upsilon \mid \upsilon \sqsubseteq \tau \,\}$$

$\lfloor e \rfloor$, $\lfloor \mathcal{C} \rfloor$, 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 are simply lift unchanged into the slice lattice. Importantly all terms in the slice lattice are consistent, so joins are always defined:

Core/Typ/ Precision

Lemma 3.4 (◯ Slices are Mutually Consistent). *If* $\upsilon_1 \sqsubseteq \tau$ *and* $\upsilon_2 \sqsubseteq \tau$, *then* $\upsilon_1 \sim \upsilon_2$.

[4] Top level type constructor

Hence, these slice lattices are *bounded lattice*. We also get that the lattice is *distributive*. And, they are on finite lattices:

Theorem 3.5 (△ Slice Lattices are Finite). *Each slice lattice $\lfloor\tau\rfloor$, $\lfloor e\rfloor$, $\lfloor\mathcal{C}\rfloor$, and $\lfloor\Gamma\rfloor$ is finite. For a term with n positions, the slice lattice has at most 2^n elements.*

Every finite distributive lattice is a bi-Heyting algebra [13], so the slice lattices are bi-Heyting: in a finite lattice the implication and subtraction operations below can be defined directly as a finite join / meet, respectively.

Core/*/Lattice

typ-slice-BiHeyting

Theorem 3.6 (◯ Slice Lattices are Bounded Distributive Bi-Heyting Algebras). *Each slice lattice $\lfloor\tau\rfloor$, $\lfloor e\rfloor$, $\lfloor\mathcal{C}\rfloor$, and $\lfloor\Gamma\rfloor$ is a bounded distributive bi-Heyting algebra:*

- ◯ ***Bounded**: bottom $\bot = \square$ (or $\emptyset$ for assumptions), top $\top = \tau$ (or e, $\mathcal{C}$, Γ).*
- ◯ ***Distributive**: meet distributes over join, $\upsilon_1 \sqcap (\upsilon_2 \sqcup \upsilon_3) = (\upsilon_1 \sqcap \upsilon_2) \sqcup (\upsilon_1 \sqcap \upsilon_3)$ (and dually).*
- ***Bi-Heyting**: equipped with*
  - ◯ ***Implication**: $\upsilon_1 \to \upsilon_2 = \bigsqcup\{\, \upsilon \mid \upsilon \sqcap \upsilon_1 \sqsubseteq \upsilon_2 \,\}$ – the most precise slice compatible with refining υ_1 to at most υ_2;*
  - ◯ ***Subtraction**: $\upsilon_1 \setminus \upsilon_2 = \bigsqcap\{\, \upsilon \mid \upsilon_1 \sqsubseteq \upsilon_2 \sqcup \upsilon \,\}$ – the least precise slice whose join with υ_2 recovers υ_1.*

Inductive Definition: Of course, calculating those joins / meets directly is very inefficient. We can define both inductively on slice lattices of the sub-terms, as demonstrated by example below for the lattice $\lfloor\tau_1 \Rightarrow \tau_2\rfloor$ (inductively on $\lfloor\tau_1\rfloor$ and $\lfloor\tau_2\rfloor$).

- *Implication* $\cdot \to \cdot$

$$\begin{aligned} \square \to \upsilon &= \top \\ \upsilon \to \square &= \square \qquad \text{(when } \upsilon \sqsupset \square\text{)} \\ * \to * &= * \\ (\upsilon_1 \Rightarrow \upsilon_2) \to (\upsilon_1' \Rightarrow \upsilon_2') &= (\upsilon_1 \to \upsilon_1') \Rightarrow (\upsilon_2 \to \upsilon_2') \end{aligned}$$

- *Subtraction* $\cdot \setminus \cdot$

$$\begin{aligned} \square \setminus \upsilon &= \square \\ \upsilon \setminus \square &= \upsilon \\ * \setminus * &= \square \\ (\upsilon_1 \Rightarrow \upsilon_2) \setminus (\upsilon_1' \Rightarrow \upsilon_2') &= \begin{cases} \square & \text{if } \upsilon_1 \setminus \upsilon_1' = \square \text{ and } \upsilon_2 \setminus \upsilon_2' = \square \\ (\upsilon_1 \setminus \upsilon_1') \Rightarrow (\upsilon_2 \setminus \upsilon_2') & \text{otherwise} \end{cases} \end{aligned}$$

Remark 3.7 (Mechanisation of Bi-Heyting operators). The implication and subtraction are only fully mechanised for the type-slice lattice $\lfloor\tau\rfloor$, which is the only lattice for which they are actually *used* in the forthcoming calculi.

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

The lattice operations have direct visual interpretations with this notation: meet $\sqcap$ is the intersection of highlighted regions, join $\sqcup$ is the union, subtraction $\setminus$ keeps the source's (LHS) highlights that are absent from the target (RHS), and implication $\to$ keeps positions outside the source's highlights and those inside the target's. For example, in $\lfloor (* \Rightarrow *) \times * \rfloor$:

$$\begin{aligned}
(* \Rightarrow *) \times * \;&\sqcap\; (* \Rightarrow *) \times * &&= (* \Rightarrow *) \times * \\
(* \Rightarrow *) \times * \;&\sqcup\; (* \Rightarrow *) \times * &&= (* \Rightarrow *) \times * \\
(* \Rightarrow *) \times * \;&\to\; (* \Rightarrow *) \times * &&= (* \Rightarrow *) \times * \\
(* \Rightarrow *) \times * \;&\setminus\; (* \Rightarrow *) \times * &&= (* \Rightarrow *) \times *
\end{aligned}$$

No Boolean Lattice. The pseudo-complement $\neg v = v \to \square$ and the supplement $\sim v = \top \setminus v$ do *not* coincide on slice lattices, so they are not Boolean algebras. Take $v = * \Rightarrow *$ in $\lfloor * \Rightarrow * \rfloor$:

$$\begin{aligned}
\neg v &= (* \Rightarrow *) \to * \Rightarrow * = * \Rightarrow *, \\
\sim v &= (* \Rightarrow *) \setminus (* \Rightarrow *) = * \Rightarrow *.
\end{aligned}$$

The two disagree ($* \Rightarrow * \neq * \Rightarrow *$). Consequentially, the Boolean complementation laws fail on either side: the Law of the Excluded Middle fails for pseudo-negation, and the Principle of Contradiction fails for the supplement:

$$v \sqcup \neg v = (* \Rightarrow *) \sqcup * \Rightarrow * = (* \Rightarrow *) \neq * \Rightarrow *$$

$$v \sqcap \sim v = (* \Rightarrow *) \sqcap (* \Rightarrow *) = * \Rightarrow * \neq * \Rightarrow *.$$

3.2.3 △ Well-Foundedness of Slice Lattices

Proposition 3.8 (△ Upward Well-Foundedness of Slice Lattices). *For each of the four families, the reversed strict precision ordering $\sqsupset$ on $\lfloor \cdot \rfloor$ is well-founded.*

Proof. Each slice lattice is finite (Theorem 3.5). A finite poset with a top element admits no infinite ascending chains. □

Combined with downwards well-foundedness (Proposition 3.2), this gives well-foundedness in both directions within each slice.

Semantics/Statics/Typing

3.3 ◯ Typing Rules

The complete bidirectional typing rules are given in Figures 12 and 13, where types may contain variables which must be within a type variable context Δ. That is, a type τ is *well-formed* under Δ, written $\Delta \vdash \tau\ \mathsf{wf}$, if every free type variable in τ is a member of Δ.

The rules are mostly standard, but I highlight notable features:

$\boxed{\Delta;\ \Gamma \vdash e \Uparrow \tau}$ e synthesises type τ under type variable context Δ and typing assumptions Γ

$$\Uparrow\Box\ \frac{}{\Delta;\ \Gamma \vdash \Box \Uparrow \Box} \qquad \Uparrow *\ \frac{}{\Delta;\ \Gamma \vdash * \Uparrow *}$$

$$\Uparrow \mathtt{Var}\ \frac{x : \tau \in \Gamma}{\Delta;\ \Gamma \vdash x \Uparrow \tau}$$

$$\Uparrow \lambda\mathtt{Ann}\ \frac{\Delta \vdash \tau_1\ \mathsf{wf} \qquad \Delta;\ \Gamma, x : \tau_1 \vdash e \Uparrow \tau_2}{\Delta;\ \Gamma \vdash \lambda x : \tau_1.\ e \Uparrow \tau_1 \Rightarrow \tau_2} \qquad \Uparrow \mathtt{let}\ \frac{\Delta;\ \Gamma \vdash e_1 \Uparrow \tau_1 \qquad \Delta;\ \Gamma, x : \tau_1 \vdash e_2 \Uparrow \tau_2}{\Delta;\ \Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \Uparrow \tau_2}$$

$$\Uparrow \Lambda\ \frac{\Delta, \alpha;\ \Gamma \vdash e \Uparrow \tau}{\Delta;\ \Gamma \vdash \Lambda\alpha.\ e \Uparrow \forall\alpha.\ \tau} \qquad \Uparrow \mathtt{TApp}\ \frac{\Delta \vdash \sigma\ \mathsf{wf} \qquad \Delta;\ \Gamma \vdash e \Uparrow \tau \qquad \tau \sqcup \forall\alpha.\ \Box \equiv \forall\alpha.\ \tau'}{\Delta;\ \Gamma \vdash e\langle\sigma\rangle \Uparrow [\alpha \mapsto \sigma]\,\tau'}$$

$$\Uparrow \mathtt{App}\ \frac{\Delta;\ \Gamma \vdash e_1 \Uparrow \tau \qquad \tau \sqcup (\Box \Rightarrow \Box) \equiv \tau_1 \Rightarrow \tau_2 \qquad \Delta;\ \Gamma \vdash e_2 \Downarrow \tau_1}{\Delta;\ \Gamma \vdash e_1(e_2) \Uparrow \tau_2}$$

$$\Uparrow \&\ \frac{\Delta;\ \Gamma \vdash e_1 \Uparrow \tau_1 \qquad \Delta;\ \Gamma \vdash e_2 \Uparrow \tau_2}{\Delta;\ \Gamma \vdash (e_1,\ e_2) \Uparrow \tau_1 \times \tau_2}$$

$$\Uparrow \pi_1\ \frac{\Delta;\ \Gamma \vdash e \Uparrow \tau \qquad \tau \sqcup (\Box \times \Box) \equiv \tau_1 \times \tau_2}{\Delta;\ \Gamma \vdash \pi_1\ e \Uparrow \tau_1} \qquad \Uparrow \pi_2\ \frac{\Delta;\ \Gamma \vdash e \Uparrow \tau \qquad \tau \sqcup (\Box \times \Box) \equiv \tau_1 \times \tau_2}{\Delta;\ \Gamma \vdash \pi_2\ e \Uparrow \tau_2}$$

$$\Uparrow \iota_1\ \frac{\Delta;\ \Gamma \vdash e \Uparrow \tau}{\Delta;\ \Gamma \vdash \iota_1\ e \Uparrow \tau + \Box} \qquad \Uparrow \iota_2\ \frac{\Delta;\ \Gamma \vdash e \Uparrow \tau}{\Delta;\ \Gamma \vdash \iota_2\ e \Uparrow \Box + \tau}$$

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

Figure 12: Synthesis rules.

$\boxed{\Delta;\ \Gamma \vdash e \Downarrow \tau}$ e analyses against type τ under Δ and Γ

$$\Downarrow \mathtt{Sub}\ \frac{\Delta;\ \Gamma \vdash e \Uparrow \tau' \qquad \tau \sim \tau'}{\Delta;\ \Gamma \vdash e \Downarrow \tau}$$

$$\Downarrow \lambda\ \frac{\tau \sqcup (\Box \Rightarrow \Box) \equiv \tau_1 \Rightarrow \tau_2 \qquad \Delta;\ \Gamma, x : \tau_1 \vdash e \Downarrow \tau_2}{\Delta;\ \Gamma \vdash \lambda x.\ e \Downarrow \tau}$$

$$\Downarrow \lambda\mathtt{Ann}\ \frac{\begin{array}{c}\Delta \vdash \tau_1\ \mathsf{wf} \qquad \tau \sim \tau_1 \Rightarrow \Box \\ \tau \sqcup \tau_1 \Rightarrow \Box \equiv \tau_1' \Rightarrow \tau_2 \qquad \Delta;\ \Gamma, x : \tau_1 \vdash e \Downarrow \tau_2\end{array}}{\Delta;\ \Gamma \vdash \lambda x : \tau_1.\ e \Downarrow \tau}$$

$$\Downarrow \mathtt{case}\ \frac{\begin{array}{c}\Delta;\ \Gamma \vdash e \Uparrow \tau' \qquad \tau' \sqcup (\Box + \Box) \equiv \tau_1 + \tau_2 \\ \Delta;\ \Gamma, x : \tau_1 \vdash e_1 \Downarrow \tau \qquad \Delta;\ \Gamma, y : \tau_2 \vdash e_2 \Downarrow \tau\end{array}}{\Delta;\ \Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\ x.\ e_1 \mid y.\ e_2 \Downarrow \tau}$$

$$\Downarrow \iota_1\ \frac{\tau \sqcup (\Box + \Box) \equiv \tau_1 + \tau_2 \qquad \Delta;\ \Gamma \vdash e \Downarrow \tau_1}{\Delta;\ \Gamma \vdash \iota_1\ e \Downarrow \tau} \qquad \Downarrow \iota_2\ \frac{\tau \sqcup (\Box + \Box) \equiv \tau_1 + \tau_2 \qquad \Delta;\ \Gamma \vdash e \Downarrow \tau_2}{\Delta;\ \Gamma \vdash \iota_2\ e \Downarrow \tau}$$

$$\Downarrow \&\ \frac{\tau \sqcup (\Box \times \Box) \equiv \tau_1 \times \tau_2 \qquad \Delta;\ \Gamma \vdash e_1 \Downarrow \tau_1 \qquad \Delta;\ \Gamma \vdash e_2 \Downarrow \tau_2}{\Delta;\ \Gamma \vdash (e_1,\ e_2) \Downarrow \tau}$$

$$\Downarrow \mathtt{let}\ \frac{\Delta;\ \Gamma \vdash e_1 \Uparrow \tau_1 \qquad \Delta;\ \Gamma, x : \tau_1 \vdash e_2 \Downarrow \tau}{\Delta;\ \Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \Downarrow \tau}$$

Figure 13: Analysis rules. $\Downarrow\mathtt{Sub}$ is the unique bridge from analysis to synthesis.

- **Join-based matching**: Terms can synthesise $\square$ which is not syntactically a function, but should be treated as such (it is consistent with the function type). We can extract the argument and return types with a join: $\tau \sqcup (\square \Rightarrow \square) \equiv \tau_1 \Rightarrow \tau_2$. Similarly for other compound types.[5]
- **Case expressions** can be type-checked in synthesis mode ($\Uparrow$`case`), require the branch types τ_1 and τ_2 to be *consistent*, synthesising the *join* $\tau_1 \sqcup \tau_2$.
- **Dynamic sum types**: Sum injections have synthesis rules which infer the absent component of the sum with $\square$: $\iota_1\ e$ synthesises $\tau + \square$ where e synthesises τ (and dually for ι_2). This allows sum types to be inferred across case branches by joining types without annotations, like the behaviour shown in fig. 3 (section 1.1).
- **Annotated vs. unannotated lambdas**: Annotated lambdas, $\lambda x : \tau_1.\ e$, can synthesise their type using the annotation on the argument, while $\lambda x.\ e$ can only analyse as there is no way to determine the type of the argument at the definition site.
- **Unannotated Let bindings**: Let bindings, **let** $x = e_1$ **in** e_2, *synthesise* e_1 and use this to type the body, without requiring annotation.

Core/Ctx/Base, Core/Ctx/Precision, Semantics/Statics/CtxTyping

3.4 ◯ Context Classification

Context classification is a novel construction for bidirectional type systems. It is inspired by one-hole contexts (zippers) of algebraic data types [17], decomposing a data structure into a focused element and its surrounding context. I apply the same idea applies to *typing derivations*: given a derivation of an expression e and focusing on a sub-term e' in e, the classification decomposes it into the *surrounding context* (carrying all typing information *outside* the focused sub-term e') and the *focused sub-derivation* (carrying all typing information from *inside* the focused sub-term e').

The classification also records 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). Both of these are necessary to allow type checking of the focus directly from the context classification derivation.

Remark 3.9. This construction replaces and generalises the notion of *checking contexts* from the previous type slicing theory.

3.4.1 ◯ Context Syntax

First, I define one-hole contexts on terms in Figure 14. The focus of the context is notated ◯ and is restricted to a single position in an expression.

3.4.2 ◯ Context Precision and Plugging Contexts

Then, for any context, you can substitute a term into the focus to produce a complete term. This is notated by $\mathcal{C}[e]$, plugging e into context $\mathcal{C}$.

Precision on contexts, $\mathcal{C}' \sqsubseteq \mathcal{C}$, requires the two contexts to share the focus in the same structural position, with componentwise precision on sub-terms and sub-contexts. Within

[5] This is equivalent to the standard use of 'matching functions' in the gradual typing literature, typically notated $\tau \blacktriangleright \tau_1 \Rightarrow \tau_2$.

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

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

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

$$\begin{aligned}\bigcirc[e] &= e \\ (\lambda x : \tau.\ \mathcal{C})[e] &= \lambda x : \tau.\ \mathcal{C}[e] \\ (\mathcal{C}_1(e_2))[e] &= \mathcal{C}_1[e](e_2) \qquad \text{(similarly for all other constructors)}\end{aligned}$$

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

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

Figure 14: One-hole expression contexts. $\bigcirc$ marks the hole; $\mathcal{C}[e]$ plugs it.

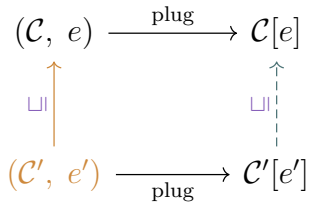


Figure 15: Plug preserves precision (Lemma 3.10).

each slice lattice $\lfloor\mathcal{C}\rfloor$ (Section 3.2.2), the bottom is the *purely structural context* $\square_{\mathcal{C}}$: all sub-terms omitted, preserving only the focus location (Figure 14).

Downwards well-foundedness holds within each $\lfloor\mathcal{C}\rfloor$ by the same argument as Proposition 3.2.

Importantly, context plugging preserves precision:

Lemma 3.10 ($\bigcirc$ Plug Preserves Precision). *If* $\mathcal{C}' \sqsubseteq \mathcal{C}$ *and* $e' \sqsubseteq e$, *then* $\mathcal{C}'[e]' \sqsubseteq \mathcal{C}[e]$. *(Figure 15)*

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

3.4.3 $\bigcirc$ The Classification Judgement

The context classification judgement is notated:

$$\Delta;\ \Gamma \vdash \mathcal{C}\ \textbf{at}\ d\ \triangleright\ \Gamma'\ [\,m\,]$$

reading as: "under type variable context Δ and assumptions Γ, context $\mathcal{C}$ in a derivation of outer mode d has its focus typed under Δ'; Γ' 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 ($d = \Uparrow \tau$) or an analysis derivation against τ ($d = \Downarrow \tau$). The *focus mode* m records what the focus e must satisfy: either $m = \Uparrow \tau'$ (the focus must synthesise type τ') or $m = \Downarrow \tau'$ (the focus must analyse against τ'). Both the outer derivation mode and focus mode carry a type, enabling the classification to track type flow through the derivation.

Further, we record the *focus assumptions* Γ' and the *focus mode* m as mentioned previously. Finally, several rule require *extra premises* which systematically match with the premises of the typing rule that each context classification rule represents.

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

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

I give a few representative rules of context classification in Figure 16. The construction of these rules is systematic, one for each typing rule and focus position. The only exception being the identity contexts $\texttt{s}\bigcirc$ and $\texttt{a}\bigcirc$ (the base cases) saying that 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 classifies its focus under $\Delta'; \Gamma'$ in focus mode m

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

$$\texttt{s}\bigcirc\ \frac{}{\Delta; \Gamma \vdash \bigcirc \textbf{ at } \Uparrow \tau \triangleright \Delta; \Gamma\,[\Uparrow \tau\,]} \qquad \texttt{a}\bigcirc\ \frac{}{\Delta; \Gamma \vdash \bigcirc \textbf{ at } \Downarrow \tau \triangleright \Delta; \Gamma\,[\Downarrow \tau\,]}$$

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

$$\texttt{so}_1\ \frac{\Delta; \Gamma \vdash \mathcal{C} \textbf{ at } \Uparrow \tau \triangleright \Delta'; \Gamma'\,[m] \qquad \tau \sqcup (\square \Rightarrow \square) \equiv \tau_1 \Rightarrow \tau_2 \qquad \Delta; \Gamma \vdash e_2 \Downarrow \tau_1}{\Delta; \Gamma \vdash \mathcal{C}(e_2) \textbf{ at } \Uparrow \tau_2 \triangleright \Delta'; \Gamma'\,[m]}$$

$$\texttt{so}_2\ \frac{\Delta; \Gamma \vdash e_1 \Uparrow \tau \qquad \tau \sqcup (\square \Rightarrow \square) \equiv \tau_1 \Rightarrow \tau_2 \qquad \Delta; \Gamma \vdash \mathcal{C} \textbf{ at } \Downarrow \tau_1 \triangleright \Delta'; \Gamma'\,[m]}{\Delta; \Gamma \vdash e_1(\mathcal{C}) \textbf{ at } \Uparrow \tau_2 \triangleright \Delta'; \Gamma'\,[m]}$$

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

$$\texttt{aSub}\ \frac{\Delta; \Gamma \vdash \mathcal{C} \textbf{ at } \Uparrow \tau' \triangleright \Delta'; \Gamma'\,[m] \qquad \tau \sim \tau'}{\Delta; \Gamma \vdash \mathcal{C} \textbf{ at } \Downarrow \tau \triangleright \Delta'; \Gamma'\,[m]}$$

Figure 16: Selected context classification rules.

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

plug-decompose

Theorem 3.11 (◯ Plug Decomposition – Totality). *For any context $\mathcal{C}$, expression e, and outer derivation mode d:*

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

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

plug-compose

Theorem 3.12 (◯ Plug Composition – Soundness). *For any context $\mathcal{C}$, expression e, outer derivation 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 $\Delta; \Gamma$.*

Semantics/ Graduality

3.5 ◯ Metatheory: Graduality & Unicity

Finally, the static gradual guarantee (section 2.3), standard in gradual typing systems, is an essential monotonicity theorem used to ensure efficient computability of type slices.

static-gradual-syn

Theorem 3.13 (◯ Static Gradual Guarantee – Synthesis). *If $\Gamma' \sqsubseteq \Gamma$, $e' \sqsubseteq e$, and $\Delta; \Gamma \vdash e \Uparrow \tau$, then there exists τ' such that $\Delta; \Gamma' \vdash e' \Uparrow \tau'$ and $\tau' \sqsubseteq \tau$.*

static-gradual-ana

Theorem 3.14 (◯ Static Gradual Guarantee – Analysis). *If $\Gamma' \sqsubseteq \Gamma$, $e' \sqsubseteq e$, $\tau' \sqsubseteq \tau$, and $\Delta; \Gamma \vdash e \Downarrow \tau$, then $\Delta; \Gamma' \vdash e' \Downarrow \tau'$.*

The same monotonicity extends to context classifications, where $m' \sqsubseteq m$ lifts type precision through the focus mode constructors: $\Uparrow \tau' \sqsubseteq \Uparrow \tau$ iff $\tau' \sqsubseteq \tau$, and likewise for $\Downarrow$. The focus modes itself (analysis or synthesis) is propagated unchanged through every classification rule.

static-gradual-syn-cls

Theorem 3.15 (◯ Static Gradual Guarantee – Context, Synthesis Position). *If $\Gamma' \sqsubseteq \Gamma$, $\mathcal{C}' \sqsubseteq \mathcal{C}$, and $\Delta; \Gamma \vdash \mathcal{C}$ **at** $\Uparrow \tau_p \triangleright \Delta_f; \Gamma_f [m]$, 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** $\Uparrow \tau'_p \triangleright \Delta_f; \Gamma'_f [m']$.*

static-gradual-ana-cls

Theorem 3.16 (◯ 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** $\Downarrow \tau_p \triangleright \Delta_f; \Gamma_f [m]$, then there exist Γ'_f and m' with $\Gamma'_f \sqsubseteq \Gamma_f$, $m' \sqsubseteq m$, and $\Delta; \Gamma' \vdash \mathcal{C}'$ **at** $\Downarrow \tau'_p \triangleright \Delta_f; \Gamma'_f [m']$.*

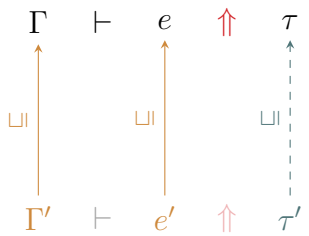


Figure 17: The static gradual guarantee – Synthesis (Theorem 3.13).

Theorem 3.17 (◯ Synthesis Unicity). *If* $\Delta; \Gamma \vdash e \Uparrow \tau_1$ *and* $\Delta; \Gamma \vdash e \Uparrow \tau_2$, *then* $\tau_1 = \tau_2$.

Corollary 3.18 (◯ Synthesis Precision). *If* $\Gamma' \sqsubseteq \Gamma$, $e' \sqsubseteq e$, $\Delta; \Gamma \vdash e \Uparrow \tau$, *and* $\Delta; \Gamma' \vdash e' \Uparrow \tau'$, *then* $\tau' \sqsubseteq \tau$.

Mechanisation provenance. The Agda mechanisation of graduality on expression typing (theorems 3.13 and 3.14) is loosely based on `hazelgrove/hazelnut-polymorphism-agda`, but had to be re-proven from scratch to integrate with the Type Slicing core calculus.[6] The graduality theorems for context classification (theorems 3.15 and 3.16) are new to this work, since context classification itself is novel.

[6] Which is sufficiently similar to Hazel for this provenance to be useful.

4 Synthesis Slices

Legend: ◯ fully mechanised, ⓐ mechanised with postulates, △ not mechanised, ▭ important theorem, ▭ counterexample.

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

Slicing/Synthesis/Synthesis

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)[7], all of its type information is contained within the synthesis derivation:

$$D \;:\; \Delta; \Gamma \vdash e \Uparrow \tau$$

Synthesis slices work in the following manner: Given the synthesis derivation D, we may *query* the type τ by taking a slice υ in $\lfloor \tau \rfloor$. A resulting 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 $\upsilon = \mathit{Int} \Rightarrow \square$ for $\tau = \mathit{Int} \Rightarrow \mathit{Bool}$ asks "which parts of the program account for the domain being *Int*?". Importantly, the resulting program slice might not include *all* the program regions relevant to the query type, but will necessarily provide a consistent subset which suffice to totally *explain* (independently synthesise) the query.

Definition 4.1 (◯ Synthesis slice)**.** A *synthesis slice* of $D : \Delta; \Gamma \vdash e \Uparrow \tau$ on $\upsilon \in \lfloor \tau \rfloor$, written $\mathit{SynSlice}\ D \blacktriangleleft \upsilon$, is a pair $(\gamma, \varsigma) \in \lfloor \Gamma, e \rfloor$ such that $\Delta; \gamma \vdash \varsigma \Uparrow \phi$ for some $\phi \sqsupseteq \upsilon$ (with $\phi \in \lfloor \tau \rfloor$ by graduality, theorem 3.13).

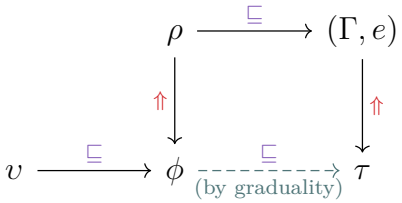


Figure 18: Diagram of a synthesis slice; $\upsilon \sqsubseteq \phi$ is the validity witness.

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.

Exactness. Note that synthesis slices do not strictly demand $\upsilon = \phi$. I terms such slices that do as *exact* synthesis slices. Exact slices are an intuitive way to define synthesis slices, but we shall see that exactness is *too strong* a condition.

[7] 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.

Definition 4.2 (◯ Exact synthesis slice)**.** *ExactSynSlice* $D \blacktriangleleft \upsilon$ is a synthesis slice (γ, ς) : *SynSlice* $D \blacktriangleleft \upsilon$ whose synthesised type ϕ satisfies $\phi \sqsubseteq \upsilon$.

However, exact synthesis slices are too strong a constraint to use in general. They need not exist:

Slicing/Counterexamples ¬ syn-preserves-join

Counterexample 4.3 (◯ Non-Existence of all Exact Synthesis Slices). *Consider* $x : * \Rightarrow * \vdash (x, x) \Uparrow (* \Rightarrow *) \times (* \Rightarrow *)$*, and query* $\upsilon = (* \Rightarrow \square) \times (\square \Rightarrow *)$.
Any strictly smaller slice of either $x : * \Rightarrow *$ *or* (x, x) *synthesises a type strictly less precise that* υ*. Yet the original program synthesises* $(* \Rightarrow *) \times (* \Rightarrow *)$*, strictly more precise than* υ.

Slicing/Synthesis/Synthesis

4.2 ◯ Minimality

Synthesis slices give us *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 that further slicing any part of the program makes the slice invalid (synthesising a type insufficient to cover the query).

Definition 4.4 (◯ Minimality)**.** For an element a of a poset, a is minimal, $\mathsf{IsMinimal}\,(a)$, when all a' with $a' \sqsubseteq a$ are in fact equal to a, have $a' = a$.

Definition 4.5 (◯ Minimal Synthesis Slices)**.** A *MinSynSlice* $D \blacktriangleleft \upsilon$ is a *SynSlice* $D \blacktriangleleft \upsilon$ minimal in *SynSlice* $D \blacktriangleleft \upsilon$ under program-slice precision on $\lfloor \Gamma, e \rfloor$.

Slicing/Synthesis/Synthesis

4.3 △ Existence of Minimal Slices

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

minExists

Theorem 4.6 (△ Existence of minimal slices). *For every* s : *SynSlice* $D \blacktriangleleft \upsilon$*, there exists* m : *MinSynSlice* $D \blacktriangleleft \upsilon$ *with* $m \sqsubseteq s$.

Proof. The slice lattice $\lfloor \Gamma, e \rfloor$ is finite. Hence, $\{\, s' : \textit{SynSlice}\ D \blacktriangleleft \upsilon \mid s' \sqsubseteq s \,\}$ is finite, which can be decided simply by enumerating all $s' : \textit{SynSlice}\ D \blacktriangleleft \upsilon$ and deciding $s' \sqsubseteq s$. If this set is empty, then s' is minimal. □

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

The proof above is not a practical algorithm. In reality, for any term, you can efficiently compute the list of maximal slices that are strictly less precise than some given term (by omitting exactly *one* leaf from the AST). Then pick the first maximal strict slice that satisfies the query and recurse; if none of the slices satsify the query, then you have found a minimal slice. 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 19 for an example lattice to explore, marking all minima (of which there may be multiple, Counterexample 4.7).

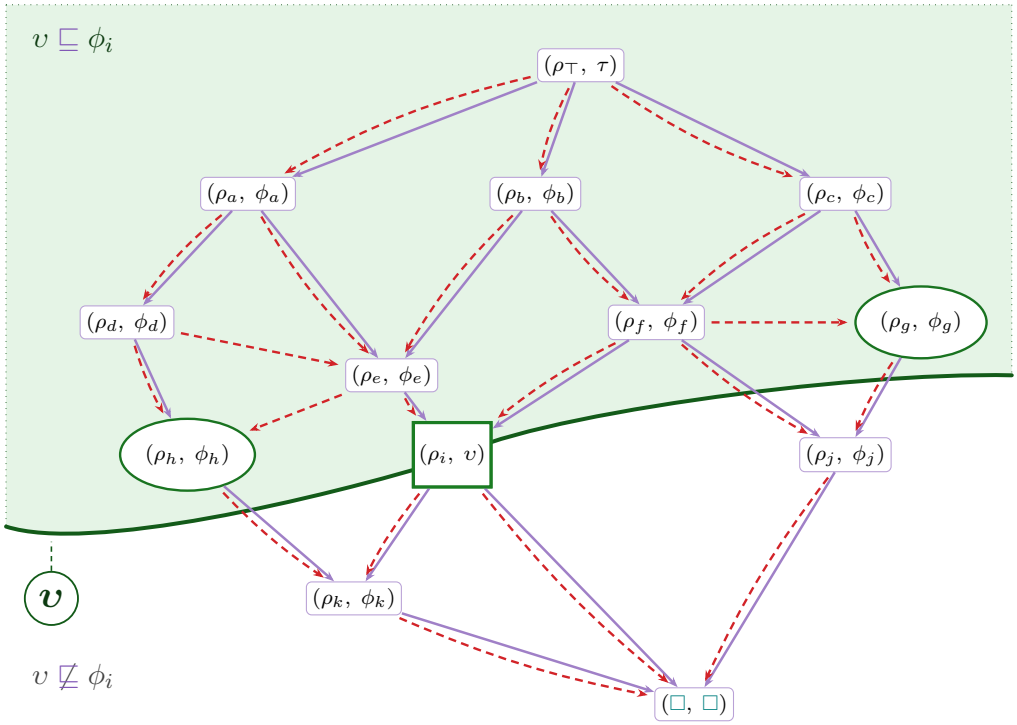


Figure 19: An example Hasse diagram of program slices in $\lfloor \rho_\top \rfloor$ overlayed with type precision relations of their synthesised types (mostly induced by graduality). Classified into a shaded region of *valid synthesis slices*, synthesising $\phi_i \sqsupseteq v$.
Purple: program precision $\sqsubset$. Red: type precision $\sqsubseteq$. Ellipses: minimal synthesis slices. Square: an exact minimal slice

Counterexample 4.7 (△ Non-Uniqueness of Minimal Slices). *The program* **if true then** $*$ **else** $*$ *has two incomparable minimal slices for* $v = *$*:* **if** $\square$ **then** $*$ **else** $\square$ *and* **if** $\square$ **then** $\square$ **else** $*$.

Refining a query v to a smaller $v' \sqsubseteq v$ descends further in the lattice, finding minima which are less precise, but consistent with, the previous slice for v.

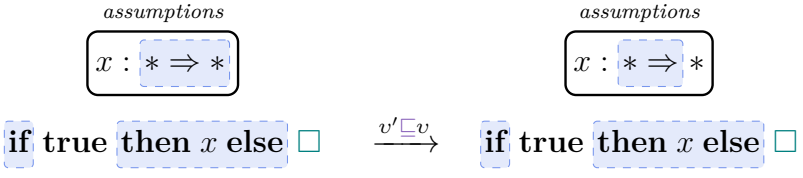
minMono **Theorem 4.8** (△ Monotonicity of minimal slices). *If* $v_1 \sqsubseteq v_2$ *in* $\lfloor \tau \rfloor$ *and* $m_2 : \mathit{MinSynSlice}\ D \blacktriangleleft v_2$*, then there exists* $m_1 : \mathit{MinSynSlice}\ D \blacktriangleleft v_1$ *with* $m_1 \sqsubseteq m_2$.

Slicing/Synthesis/FixedAssmsCalc

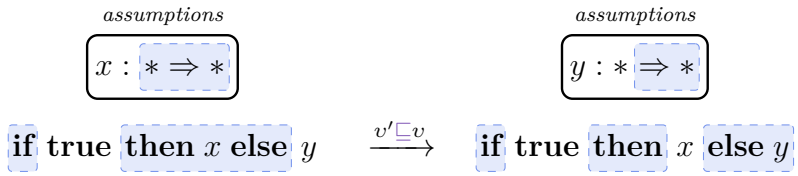
4.4 ◯ Sub-slices

In practice, a user may not arrive at a query atomically. One might look at a synthesised type, then refine the query to explore one component of it, then into a sub-component. For this use case we get successive queries forming a descending chain $v \sqsupseteq v' \sqsupseteq v'' \sqsupseteq \cdots$. Due to monotonicity, this chain can be associated with a chain of program slices $\rho \sqsupseteq \rho' \sqsubseteq \cdots$.

Example. Refining $v = * \Rightarrow *$ to $v' = * \Rightarrow \square$ on $e =$ **if true then** x **else** y:



Branch hopping. This descending chain of program slices is useful because it intuitively aligns with the users thinking of *refining* an existing slice, and retrieving a refined program consistent with the last query. Picking random minimal slices will not necessarily form such a chain, giving rise to 'branch hopping' where highlighted regions switch between branches when refining the query, which is undesirable from a UX perspective. For example, refining $\upsilon = * \Rightarrow *$ to $\upsilon' = \square \Rightarrow *$:



Slicing/Synthesis/Decompositions

4.5 ◯ Decompositions

Instead of calculating slices by brute force (section 4.3), it would be ideal if we could calculate minimal slices of a term from minimal subslices of its subterms. This section proves that any minimal slice on a compound can indeed be created by composing minimal slices on its subterm, suggesting that such approaches are indeed possible; these are further explored in the optimisations of section 7.

Firstly, each compound typing rule must come with a *slice combinator*: an operation that builds a synthesis slice from slices of the typing rule's premises, joining their shared assumptions. Then, we must be able to decompose a compound minimal slice into minimal slices combined using the corresponding combinator.

I give the two of the cases: the product rule and the let binding rule. The remaining cases are similar.

4.5.1 ◯ Products

Given slices $(\gamma_1, \varsigma_1) : \mathit{SynSlice}\ D_1 \blacktriangleleft \upsilon_1$ and $(\gamma_2, \varsigma_2) : \mathit{SynSlice}\ D_2 \blacktriangleleft \upsilon_2$ for the two components of a pair $D = \Uparrow \mathtt{Pair}\ D_1\ D_2$ (with synthesised types ϕ_1, ϕ_2), we form their product slice $(\gamma_1, \varsigma_1) \times (\gamma_2, \varsigma_2) : \mathit{SynSlice}\ D \blacktriangleleft \upsilon_1 \times \upsilon_2$ 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)[8] types $\phi_1' \times \phi_2' \sqsupseteq \phi_1 \times \phi_2 \sqsupseteq \upsilon_1 \times \upsilon_2$. For this combinator, we get the following decomposition theorem:

min-prod-decomposability

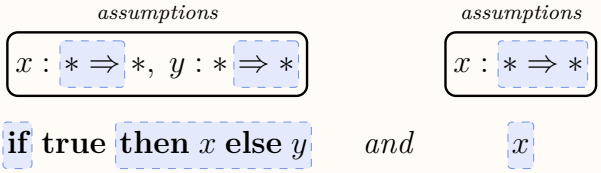
Theorem 4.8.1 (◯ *Case: Products*)**.** *Let* $m : \mathit{MinSynSlice} \Uparrow \mathtt{Pair}\ D_1\ D_2 \blacktriangleleft \upsilon_1 \times \upsilon_2$. *Then there exist minimal* $m_1 : \mathit{MinSynSlice}\ D_1 \blacktriangleleft \upsilon_1$ *and* $m_2 : \mathit{MinSynSlice}\ D_2 \blacktriangleleft \upsilon_2$ *with* $m = m_1 \times m_2$.

◯ **Converse Failure** However, the converse is not true:

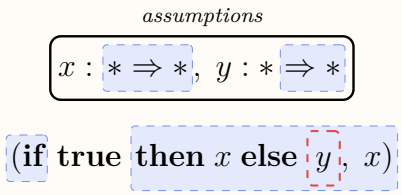
[8] Due to $\gamma_1 \sqcup \gamma_2$ being larger than γ_1 and γ_2

¬&syn-preserves-minimality

Counterexample 4.9 (◯ Paired minimal slices are not necessarily minimal). *Consider two minimal slices, both on query* $* \Rightarrow *$*:*



Note that left slice is minimal due to the assumption slice, omitting either x *or* y *would no longer synthesise* $* \Rightarrow *$*. However, the pair combinator, which pairs the expression slices and joins the assumption slices, produces:*



Which is not minimal, the y *outlined in red can be removed while still synthesising* $(* \Rightarrow *) \times (* \Rightarrow *)$ *(the natural product of the queries of the components).*

Hence, we cannot simply recurse through the typing derivation to calculate a slice on $v_1 \times v_2$ from sub-slices of the components on v_1 and v_2 respectively. Section 7 solves this issue.

4.5.2 ◯ Let Bindings

A more subtle case is of let-bindings, which has difference assumptions for its two premises:

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

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:

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

min-def-decomposability **Theorem 4.8.2** (*Case: Bindings*). *Let* m : *MinSynSlice* $\Uparrow\texttt{Let}\ D_1\ D_2 \blacktriangleleft v$. *Then there exists* $\phi_1 \in \lfloor \tau_1 \rfloor$ *and minimal slices* m_1 : *MinSynSlice* $D_1 \blacktriangleleft \phi_1$ *(synthesising* ϕ_1'*) and* $m_2 = (\gamma_2, \varsigma_2)$: *MinSynSlice* $D_2 \blacktriangleleft v$ *satisfying* $\gamma_2(x) \sqsubseteq \phi_1'$, *with* $m = m_1\ \mathsf{def}\ m_2$.

Slicing/Synthesis/Synthesis

4.6 ◯ Contribution Slices

A natural way represent a 'least' slice that retains *all* information relevant to a query type query, rather than just enough to consistently explain the type, would be to *join* all minimal slices for the slices. These joins are necessarily unique and are 'least' in the sense of being a least upper bound of actual minimal slices.

For this to make sense, closure of synthesis slices under joins is the crucial, joining two synthesis slices explaining υ should still explain υ fully. This is a second instance where exact synthesis slices are too strong, we do *not* get that joining two exact slices synthesises exactly the joined query:

¬⊔syn-closed

Counterexample 4.10 (◯ Joins do not preserve exactness). *Consider* $D : (x : *) \vdash x \Uparrow *$. *For the query* $\upsilon = \square$, *consider two exact (but not minimal) slices:* $x : \square \vdash x \Uparrow \square$ *and* $x : * \vdash \square \Uparrow \square$.
However, their join synthesises $*$: $x : * \vdash x \Uparrow *$, *despite the join of the queries being* $\square$. *This synthesis is not exact.*

Even if we restrict this the exact slices to be minimal:

¬⊔syn-preserves-join

Counterexample 4.11 (◯ Joins do not preserve exactness, even under minimality). *Consider* $D = x : * \Rightarrow * \vdash (x, x) \Uparrow (* \Rightarrow *) \times (* \Rightarrow *)$. *Then for queries* $\upsilon_1 = \square \times (\square \Rightarrow *)$ *and* $\upsilon_2 = (* \Rightarrow \square) \times \square$, *we get minimal slices* $(x : \square \Rightarrow *,\ (\square, x))$ *and* $(x : * \Rightarrow \square),\ (x, \square)$ *respectively.*
Joining these minimal slices gives the original derivation D, *which synthesises* $(* \Rightarrow *) \times (* \Rightarrow *) \sqsupset (* \Rightarrow \square) \times (\square \Rightarrow *) = \upsilon_1 \times \upsilon_2$.

However, the weaker validity condition for regular synthesis slices $\upsilon \sqsubseteq \phi$, does allow for join closure.

⊔syn

Theorem 4.12 (◯ Closure of Synthesis Slices under Join). *For* $(\gamma_1, \varsigma_1) : \mathit{SynSlice}\ D \blacktriangleleft \upsilon_1$ *and* $(\gamma_2, \varsigma_2) : \mathit{SynSlice}\ D \blacktriangleleft \upsilon_2$, *then* $(\gamma_1, \varsigma_1) \sqcup (\gamma_2, \varsigma_2) : \mathit{SynSlice}\ D \blacktriangleleft \upsilon_1 \sqcup \upsilon_2$, *with the joined slice taken pointwise:* $(\gamma_1 \sqcup \gamma_2,\ \varsigma_1 \sqcup \varsigma_2)$.

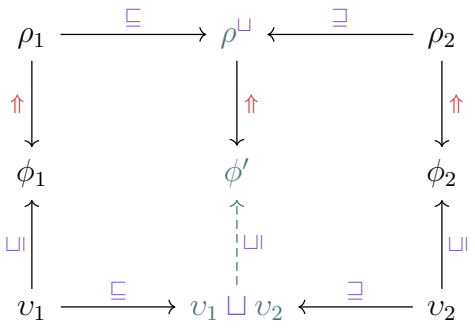


Figure 20: Theorem 4.12.

5 Analysis Slices

Legend: ◯ fully mechanised, ◬ mechanised with postulates, △ not mechanised, ▭ important theorem, ▭ counterexample.

Analysis slices explain why a sub-terms is *analysed* against an expected type. The principal difference from synthesis slices is the object being sliced: where a synthesis slice sliced a derivation $\Delta; \Gamma \vdash e \Uparrow \tau$, an analysis slice slices a *context classification* $\Delta; \Gamma \vdash \mathcal{C}$ **at** $d \triangleright \Delta'; \Gamma' [m]$ around the selected term. The expression slice ς is replaced by a context slice $\kappa \in \lfloor \mathcal{C} \rfloor$; everything else (minimality, existence, monotonicity) proceeds analogously to Section 4.

Slicing/Analysis/Analysis

5.1 ◯ Analysis Slices

Analysis slices concern the classifications with focus mode $m = \Downarrow\tau$. A query $\upsilon \in \lfloor \tau \rfloor$ specifies the part of τ that we wish to explain.

Unlike synthesis slices, there are two possible outer derivations modes d. Analysis slices are for sub-terms within a synthesising expression, but the alternative concept, *inner* analysis slices, are for when the whole term is being analysed.

Definition 5.1 (◯ Analysis slice)**.** An *analysis slice* of $Cls : \Delta; \Gamma \vdash \mathcal{C}$ **at** $\Uparrow\tau_p \triangleright \Delta'; \Gamma' [\Downarrow\tau]$ on $\upsilon \in \lfloor \tau \rfloor$, written $AnaSlice\ Cls \blacktriangleleft \upsilon$, is a pair $(\kappa, \gamma) \in \lfloor \mathcal{C}, \Gamma \rfloor$ such that $\Delta; \gamma \vdash \kappa$ **at** $\Uparrow\psi \triangleright \Delta''; \Gamma'' [\Downarrow\upsilon]$ for some outer-type witness $\psi \sqsupseteq \upsilon$ (with $\psi \in \lfloor \tau_p \rfloor$ by graduality, theorem 3.15).

Definition 5.2 (◯ Inner analysis slice)**.** An *inner analysis slice* of $Cls : \Delta; \Gamma \vdash \mathcal{C}$ **at** $\Downarrow\tau_p \triangleright \Delta'; \Gamma' [\Downarrow\tau]$ on $\upsilon \in \lfloor \tau \rfloor$, written $InnerAnaSlice\ Cls \blacktriangleleft \upsilon$, is a pair $(\kappa, \gamma) \in \lfloor \mathcal{C}, \Gamma \rfloor$ such that, for some chosen outer-type $\upsilon_{\text{outer}} \in \lfloor \tau_p \rfloor$ then $\Delta; \gamma \vdash \kappa$ **at** $\Downarrow\upsilon_{\text{outer}} \triangleright \Delta''; \Gamma'' [\Downarrow\upsilon]$.

Slicing/Analysis/Analysis

5.2 ◯ Minimality

$MinAnaSlice\ Cls \blacktriangleleft \upsilon$ defined analogously to Definition 4.4, minimising over the context slice (κ, γ). Similarly for $MinInnerAnaSlice\ Cls \blacktriangleleft \upsilon$, except υ_{outer} is also minimsed, i.e. we additionally find a minimal outer type to the whole expressions check against, that still checks the focus at our query υ.

As befoe, minimal analysis slices need not be unique, but this non-determinism is wholly inherited from synthesis.

Slicing/Analysis/Analysis

5.3 ◬ Existence of Minimal Slices

minExists/minExistsPos

Theorem 5.3 (◬ Existence of minimal analysis slices). *For every $s : AnaSlice\ Cls \blacktriangleleft \upsilon$, there exists $m : MinAnaSlice\ Cls \blacktriangleleft \upsilon$ with $m \sqsubseteq s$; and analogously for InnerAnaSlice.*

Proof. The slice lattice $\lfloor \mathcal{C}, \Gamma \rfloor$ is finite, so $\{(\kappa', \gamma') \sqsubseteq (\kappa, \gamma) \mid (\kappa', \gamma') \text{ is a slice}\}$ is finite. Enumerate it and descend; the chain terminates at a minimal element. □

mono/monoPos

Theorem 5.4 (◬ Monotonicity of minimal analysis slices). *If $\upsilon_1 \sqsubseteq \upsilon_2$ in $\lfloor \tau \rfloor$ and $m_2 : MinAnaSlice\ Cls \blacktriangleleft \upsilon_2$, then there exists $m_1 : MinAnaSlice\ Cls \blacktriangleleft \upsilon_1$ with $m_1 \sqsubseteq m_2$; analogously for InnerAnaSlice.*

Semantics/Marking/

6 Error Marking & Type Error Debugging

Legend: ◯ fully mechanised, ⚠ mechanised with postulates, △ not mechanised, ▭ important theorem, ▭ counterexample.

This chapter associates the tools of this formalism with the error squiggles of fig. 2 (section 1.1). 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. [37], which I briefly recap and extend to the type slicing core calculus (section 3) in the next subsection.

Semantics/Marking/ Judgment

6.1 ◯ Error Marking

A marking algorithm is a pair of bidirectional judgements,

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

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, so cannot be debugged by type slicing.

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
$\langle\!\lfloor e \rfloor\!\rangle_{\blacktriangleright \Rightarrow}$	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	Typfun Application	e is in a typfun positon, but is not a typfun
$\langle\!\lfloor \lambda x.\, e \rfloor\!\rangle_{\sim \Rightarrow}$	mode limitation	Lambda Synthesis	Unannotated lambda in synthesis position
$\langle\!\lfloor \lambda x.\, e \rfloor\!\rangle_{\sim \Rightarrow}$	mode limitation	Lambda Analysis	Unannotated lambda analysed against a non-function type
$\langle k \rangle \Uparrow$	scope	Variables	Variable k has no binding in scope

Table 1: Error Mark Forms Classified

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

The validity of this method is formalised by two theorems: *totality* ensures marks account for every possible typing error, and *erasure* ensures adding marks doesn't change the structure of the terms.

Semantics/Marking/Metatheory **Theorem 6.1** (◯ Totality of Marking). *For every expression e and assumptions $\Delta;\Gamma$. There exists marked expression $\check{e}$ and type τ such that $\Delta;\Gamma \vdash e \looparrowright \check{e} \Uparrow \tau$. And, for every type τ', there exists $\check{e}'$ such that $\Delta;\Gamma \vdash e \looparrowright \check{e}' \Downarrow \tau$.*

Semantics/Marking/Metatheory **Theorem 6.2** (◯ Erasure of Marking). *If $\Delta;\Gamma \vdash e \looparrowright \check{e} \Uparrow \tau$ then $\mathrm{erase}(\check{e}) \equiv e$, and likewise for analysis.*

Mechanisation provenance. The Agda development is loosely based on the mechanisation accompanying Zhao et al. [37] – `hazelgrove/error-localization-agda` – but the proofs are not direct ports. They were re-derived from scratch against the core calculus of section 3, following the recipe proposed in the paper.

Semantics/Marking/CtxMarking

6.2 ◯ Marked Context Classification

Context classification (section 3.4) gives a reading of where a one-hole context C sits in a derivation. This can be extended to marked context classification $\Delta;\Gamma \vdash C \looparrowright \check{C}$ **at** $p \triangleright \Delta';\Gamma'\,[m]$ threads a marked context $\check{C}$ alongside C (with context marks defined analogously to expressions), with one error rules per mark form in table 1.

For example, the type inconsistency mark can be inserted into context classification with the following rule:

$$maSub\Uparrow \frac{\Delta;\Gamma \vdash C \looparrowright \check{C} \textbf{ at } \Uparrow \tau' \triangleright \Delta';\Gamma'\,[m] \qquad \neg(\tau \sim \tau')}{\Delta;\Gamma \vdash C \looparrowright (\!|\check{C}|\!)_{\tau' \not\sim \tau} \textbf{ at } \Downarrow \tau \triangleright \Delta';\Gamma'\,[m]}$$

The marked context wraps the inner marked context $\check{C}$ in an inconsistency mark when the analysed and synthesised types of the *full plugged expression* differ: $\tau' \not\sim \tau$. The full rule set is collected in section A.

◯ **A Worked Decomposition** Take the expression:

$$\textbf{let } p : * \times (* \Rightarrow *) = (\,*,\ \texttt{true}\,) \textbf{ in } p.$$

The pair's right component synthesises `Bool` but is analysed against $* \Rightarrow *$ (from the annotation), so marking should produce:

$$\textbf{let } p : * \times (* \Rightarrow *) = (\,*,\ (\!|\texttt{true}|\!)_{\mathtt{Bool} \not\sim * \Rightarrow *}\,) \textbf{ in } p.$$

Placing the focus on the marked term `true` gives:

$$C \;=\; \textbf{let } p : * \times (* \Rightarrow *) = (\,*,\ ◯\,) \textbf{ in } p, \qquad e \;=\; \texttt{true}.$$

The classification builds $\check{C}$ from the inside out, first marking the focus with the type inconsistency:

Step (rule)	$\check{C}$ so far
ms◯	◯
$maSub\Uparrow$	$(\!|◯|\!)_{\mathtt{Bool} \not\sim * \Rightarrow *}$
ma&$_2$	$(\,*,\ (\!|◯|\!)_{\mathtt{Bool} \not\sim * \Rightarrow *}\,)$
malet …	$\textbf{let } p : * \times (* \Rightarrow *) = (\,*,\ (\!|◯|\!)_{\mathtt{Bool} \not\sim * \Rightarrow *}\,) \textbf{ in } p$

Plugging gives $\check{C}[\texttt{true}]$:

$$\textbf{let } p : * \times (* \Rightarrow *) = (\ *,\ (\!|\texttt{true}|\!)_{\texttt{Bool} \not\sim * \Rightarrow *}\ ) \textbf{ in } p.$$

Marked context classification is validated with a pair of theorems mirroring the unmarked *plug* totality and soundness theorems (theorems 3.11 and 3.12).

mplug-decompose

Theorem 6.3 (◯ Marked Plug Decomposition – Totality). *For any context $\mathcal{C}$, expression e, and outer derivation mode d:*

- *If $\Delta; \Gamma \vdash \mathcal{C}[e] \leadsto \check{e}_\star \Uparrow \tau$, then there exist Δ', Γ', $\check{\mathcal{C}}$, $\check{e}$, and m such that $\Delta; \Gamma \vdash \mathcal{C} \leadsto \check{\mathcal{C}}$ **at** $\Uparrow \tau \triangleright \Delta'; \Gamma'\ [m]$, e marks to $\check{e}$ at focus mode m, and $\check{e}_\star = \check{\mathcal{C}}[e]$.*
- *If $\Delta; \Gamma \vdash \mathcal{C}[e] \leadsto \check{e}_\star \Downarrow \tau$, then there exist Δ', Γ', $\check{\mathcal{C}}$, $\check{e}$, and m such that $\Delta; \Gamma \vdash \mathcal{C} \leadsto \check{\mathcal{C}}$ **at** $\Downarrow \tau \triangleright \Delta'; \Gamma'\ [m]$, e marks to $\check{e}$ at focus mode m, and $\check{e}_\star = \check{\mathcal{C}}[e]$.*

Here "e marks to $\check{e}$ at focus mode m" means: if $m = \Uparrow \tau'$ then $\Delta'; \Gamma' \vdash e \leadsto \check{e} \Uparrow \tau'$; if $m = \Downarrow \tau'$ then $\Delta'; \Gamma' \vdash e \leadsto \check{e} \Downarrow \tau'$.

mplug-compose

Theorem 6.4 (◯ Marked Plug Composition – Soundness). *For any context $\mathcal{C}$, marked context $\check{\mathcal{C}}$, outer derivation mode d, and focus mode m: if $\Delta; \Gamma \vdash \mathcal{C} \leadsto \check{\mathcal{C}}$ **at** $d \triangleright \Delta'; \Gamma'\ [m]$ and e marks to $\check{e}$ at focus mode m, then $\mathcal{C}[e]$ marks to $\check{\mathcal{C}}[\check{e}]$ at outer derivation mode d.*

6.3 The Debugging Recipe by Example

We may now relate the tools developed over the last 4 sections to explain type errors:

1. Place the focus (cursor) at some marked term. This splits the marked program into a marked context $\check{C}$ and marked focus $\check{e}$, with $\check{C}[\check{e}]$ recovering the program; a marked context classification at the focus is then derivable by totality (theorem 6.3).
2. Synthesis slice the internal type information, the term at the *focus*, $\check{e}$, if the focus mode m is in synthesis ($\Uparrow \tau$).
3. Analysis slice the external type information, the *context classification* itself, if the focus mode m is in analysis ($\Downarrow \tau$).
4. Pick queries to explore the error. For type-inconsistency marks both the synthesis and analysis slices exist (context classifications for both steps 2 and 3 exist), and you can query *only* the inconsistent parts. For shape errors, query just the outermost shape of the synthesised type; mode-limitation errors are analogous on the analysis side.

I give some worked examples, with the following notation:

- The marked term itself is wrapped in a red dashed box.
- The synthesis slice of the focus is highlighted in blue.
- The analysis slice of the surrounding context is highlighted in green. With purely structural parts which don't contribute to the type (i.e. bindings) in a lighter shade.[9]

[9] This idea of 'purely' structural parts is not formalised here, but is easy to implement in practice.

6.3.1 Type inconsistency: $(\!|\cdot|\!)_{\tau' \not\sim \tau}$

Consider the program

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

The RHS $(\texttt{true}, \texttt{false})$ synthesises $\texttt{Bool} \times \texttt{Bool}$ but is analysed against $* \Rightarrow *$ – inconsistent head constructors. Marking inserts $(\!|\cdot|\!)_{\texttt{Bool}\times\texttt{Bool} \not\sim *\Rightarrow *}$ around the RHS:

$$\textbf{let } p : * \Rightarrow * = (\!|(\texttt{true}, \texttt{false})|\!)_{\texttt{Bool}\times\texttt{Bool} \not\sim *\Rightarrow *} \textbf{ in } p.$$

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

$$C \;=\; \textbf{let } p : * \Rightarrow * = \bigcirc \textbf{ in } p, \qquad e \;=\; (\texttt{true}, \texttt{false}),$$

with the focus at $\Downarrow (* \Rightarrow *)$. The two debugging queries run on the two sides of this split.

Synthesis side (blue, on the focus e). *MinSynSlice* at the minimal type-slice $\upsilon \sqsubseteq \texttt{Bool} \times \texttt{Bool}$ that still witnesses $\upsilon \not\sim (* \Rightarrow *)$:

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

Analysis side (green, on the context C). *MinAnaSlice* at the minimal $\psi \sqsubseteq * \Rightarrow *$ that still witnesses $(* \times *) \not\sim \psi$:

$$C \;\rightsquigarrow\; \textbf{let } p : * \Rightarrow * = \bigcirc \textbf{ in } p \qquad \text{with} \qquad \psi = \square \Rightarrow \square.$$

Full view.

$$\begin{aligned} \text{synthesis query } \upsilon &= \texttt{Bool} \times \texttt{Bool} \;\; (\equiv \square \times \square) \\ \text{analysis query } \psi &= * \Rightarrow * \;\; (\equiv \square \Rightarrow \square) \end{aligned}$$

$$\textbf{let } p : * \Rightarrow * = (\!|(\,\texttt{true}\,,\,\texttt{false}\,)|\!)_{\texttt{Bool}\times\texttt{Bool} \not\sim *\Rightarrow *} \textbf{ in } p.$$

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

The recipe approach does not arbitrarily blame either the synthesising term or the analysing context for the error. A user who does want to trust the surrounding context (typically, annotations) as is often done in conventional type error highlighting can simply restrict the queries to include only the synthesis-side.

6.3.2 Shape mismatches: $(\!|\cdot|\!)_{\blacktriangleright\Rightarrow}$, $(\!|\cdot|\!)_{\blacktriangleright +}$, $(\!|\cdot|\!)_{\blacktriangleright \times}$, $(\!|\cdot|\!)_{\blacktriangleright \forall}$

The shape-mismatch marks ($(\!|\cdot|\!)_{\blacktriangleright\Rightarrow}$, $(\!|\cdot|\!)_{\blacktriangleright +}$, $(\!|\cdot|\!)_{\blacktriangleright \times}$, $(\!|\cdot|\!)_{\blacktriangleright \forall}$) all share the same simple recipe: the syntactic position of the focus requires a type kind, but the focus's synthesised type does not match this.

A representative example is $e = \texttt{true}\,(*)$: the function position synthesises $\texttt{Bool}$, which is a function, i.e. does not join with $\square \Rightarrow \square$:

$$\begin{aligned} \text{synthesis query } \upsilon &= \texttt{Bool} \\ \text{analysis query } \psi &= - \end{aligned}$$

$$(\!|\texttt{true}|\!)_{\blacktriangleright\Rightarrow}\,(*).$$

$(\!|\cdot|\!)_{\blacktriangleright +}$ (**case** $*$ **of** $\ldots$), $(\!|\cdot|\!)_{\blacktriangleright \times}$ $(\pi_1\,\texttt{true})$, and $(\!|\cdot|\!)_{\blacktriangleright \forall}$ are analogous.

6.3.3 Unannotated Lambda against Non-Arrow: $(\!|\cdot|\!)_{\sim\Rightarrow}$

Unannotated lambdas $\lambda y.\, e$ have no synthesis rule, type must be by the surrounding context. Consequentially 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:

$$\textbf{let}\ b : \texttt{Bool} = \lambda y.\, y\ \textbf{in}\ b.$$

The RHS $\lambda y.\, y$ is a bare lambda analysed against `Bool`. Take the *MinAnaSlice* on the enclosing context witnesses the context enforcing a non-function type (`Bool`):

$$\begin{aligned} \text{synthesis query } \upsilon &= - \\ \text{analysis query } \psi &= \texttt{Bool} \end{aligned}$$

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

6.3.4 Unbound variable: $\langle k \rangle \Uparrow$

The free-variable mark records a *scope* failure: there are no types involved.

7 Optimising Type Slice Calculation

Legend: ◯ fully mechanised, ◬ mechanised with postulates, △ not mechanised, ▭ important theorem, ▭ counterexample.

The theory of Sections 4–5 establishes *which* slices exist, and provide a brute-force algorithm. This section explores these structures further to improve the brute-force algorithm.

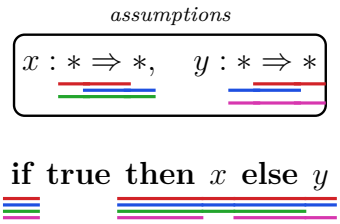
Slicing/Synthesis/FixedAssmsCalc

7.1 ◯ Products of Synthesis Slices

The product counterexample of Section 4.5.1 is worth re-examining, which stated that not all pairs of minimal slices can be paired into a minimal product slice. Consider the following if statement:

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

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



The four colours encode four minimal slices of D at υ:

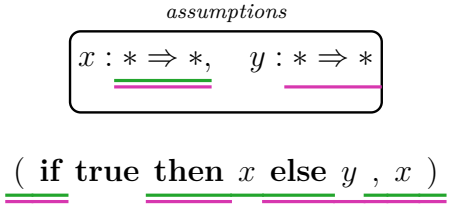
(A) **Red** : Both branches are retained; the assumption slice keeps x's domain only and y's codomain only (joining to $* \Rightarrow *$).

(B) **Blue** : Symmetric to (A), taking the opposite choices for the assumption slice

(C) **Green** : **Else**-branch omitted.

(D) **Mulberry** : 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:



Aliasing. When this expression is embedded in the product (**if true then** x **else** y, x), the outer use of x *joins* onto the assumption slice. As slices (A) (B) (C) each also use x, the assumptions are aliased, and in slices (A) (B) the assumptions now available in the if statement are strictly greater, making the expression non-minimal.

Proposal. Use a stricter notion of synthesis slices that asks only "is the term irreducible under the full assumptions Γ?". Slices (A) and (B) are ruled out, but (C) and (D) are still valid. The next subsection makes this precise.

Slicing/Synthesis/FixedAssmsSynthesis

7.2 ◯ Term Slices

To rule out slices like (A) and (B) of Section 7.1 we strengthen the notion of slice: rather than allowing the assumption context to vary across $\lfloor\Gamma\rfloor$, we fix it at the full Γ and slice the term alone. The resulting slice is a sliced term that, under the original assumptions, still synthesises a type at least as precise as the query.

Definition 7.1 (◯ Term-minimal slice)**.** A *term-minimal slice* of $D : \Delta; \Gamma \vdash e \Uparrow \tau$ on $\upsilon \in \lfloor\tau\rfloor$, written $\mathit{TermMin}\ D \blacktriangleleft \upsilon$, is a term slice $\varsigma \in \lfloor e\rfloor$ such that $\Delta; \Gamma \vdash \varsigma \Uparrow \psi$ for some $\psi \sqsupseteq \upsilon$ (with $\psi \in \lfloor\tau\rfloor$ by graduality, theorem 3.13).

Slicing/Synthesis/FixedAssmsSynthesis

7.3 ◯ Term-Minimality

Minimality lifts directly from the slice lattice on e alone:

Definition 7.2 (◯ Minimal Term-Minimal Slices)**.** A $\mathit{MinTermMin}\ D \blacktriangleleft \upsilon$ is a $\mathit{TermMin}\ D \blacktriangleleft \upsilon$ minimal in $\mathit{TermMin}\ D \blacktriangleleft \upsilon$ under term-slice precision on $\lfloor e\rfloor$.

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

¬min-syn-is-min-fixedassms

Counterexample 7.3 (◯ Minimal $\not\Rightarrow$ Term-Minimal). *Slices (A) and (B) of Section 7.1 are minimal synthesis slices but not term-minimal.*

Slicing/Synthesis/FixedAssmsSynthesis

7.4 △ Existence of Minimal Term-Minimal Slices

As with synthesis slices, every term-minimal slice has a minimal slice below it:

tmin-exists

Theorem 7.4 (△ Existence of minimal term-minimal slices). *For every $s : \mathit{TermMin}\ D \blacktriangleleft \upsilon$, there exists $m : \mathit{MinTermMin}\ D \blacktriangleleft \upsilon$ with $m \sqsubseteq s$.*

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

$$\Gamma \vdash e \Uparrow \tau \blacktriangleleft \upsilon \rightsquigarrow \varsigma \Uparrow \psi \dashv \gamma,$$

read "the typing $\Gamma \vdash e \Uparrow \tau$ at query υ 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.

Slicing/Synthesis/FixedAssmsCalc

Term-Minimal → Minimal Slice. A minimal slice is then obtained by pairing ς with any minimal assumption context γ still synthesising a type at least as imprecise as υ. This justifies the algorithmic strategy of the rest of the chapter: compute term-minimal slices structurally, then promote them to full minimal slices.

$$\boxed{\Delta;\, \Gamma \vdash e \Uparrow \tau \blacktriangleleft \upsilon \rightsquigarrow \varsigma \Uparrow \psi \dashv \gamma}$$

$$\Uparrow\texttt{Var}\ \frac{\Gamma(x) = \tau \quad \upsilon \neq \square}{\Delta;\, \Gamma \vdash x \Uparrow \tau \blacktriangleleft \upsilon \rightsquigarrow x \Uparrow \tau \dashv x : \upsilon}$$

$$\Uparrow\texttt{Gap}\ \frac{}{\Delta;\, \Gamma \vdash e \Uparrow \tau \blacktriangleleft \square \rightsquigarrow \square \Uparrow \square \dashv \emptyset}$$

$$\Uparrow\texttt{Const}\ \frac{}{\Delta;\, \Gamma \vdash * \Uparrow * \blacktriangleleft \top \rightsquigarrow * \Uparrow * \dashv \emptyset}$$

$$\Uparrow\texttt{Pair}\ \frac{\Delta;\, \Gamma \vdash e_1 \Uparrow \tau_1 \blacktriangleleft \upsilon_1 \rightsquigarrow \varsigma_1 \Uparrow \psi_1 \dashv \gamma_1 \qquad \Delta;\, \Gamma \vdash e_2 \Uparrow \tau_2 \blacktriangleleft \upsilon_2 \rightsquigarrow \varsigma_2 \Uparrow \psi_2 \dashv \gamma_2}{\Delta;\, \Gamma \vdash (e_1, e_2) \Uparrow \tau_1 \times \tau_2 \blacktriangleleft \upsilon_1 \times \upsilon_2 \rightsquigarrow (\varsigma_1, \varsigma_2) \Uparrow \psi_1 \times \psi_2 \dashv \gamma_1 \sqcup \gamma_2}$$

$$\Uparrow\texttt{Proj}\ \frac{\upsilon \neq \square \qquad \Delta;\, \Gamma \vdash e \Uparrow \tau_1 \times \tau_2 \blacktriangleleft \widehat{\upsilon}_i \rightsquigarrow \varsigma \Uparrow \psi_1 \dashv \gamma}{\Delta;\, \Gamma \vdash \pi_i\, e \Uparrow \tau_i \blacktriangleleft \upsilon \rightsquigarrow \pi_i\, \varsigma \Uparrow \pi_i\, \psi_1 \dashv \gamma}$$

$$\Uparrow\texttt{App}\ \frac{\Delta;\, \Gamma \vdash e_2 \Downarrow \tau_1 \qquad \upsilon \neq \square \qquad \Delta;\, \Gamma \vdash e_1 \Uparrow \tau_1 \Rightarrow \tau_2 \blacktriangleleft \square \Rightarrow \upsilon \rightsquigarrow \varsigma_{fn} \Uparrow \psi_1 \dashv \gamma}{\Delta;\, \Gamma \vdash e_1\, e_2 \Uparrow \tau_2 \blacktriangleleft \upsilon \rightsquigarrow \varsigma_{fn}(\square) \Uparrow \mathsf{cod}(\psi_1) \dashv \gamma}$$

$$\Uparrow\texttt{Lam:}\ \frac{\Delta;\, \Gamma, x{:}\tau_1 \vdash e_2 \Uparrow \tau_2 \blacktriangleleft \upsilon_2 \rightsquigarrow \varsigma_2 \Uparrow \psi_2 \dashv \gamma, x{:}\phi_1 \qquad \Delta;\, \Gamma, x{:}(\phi_1 \sqcup \upsilon_1) \vdash \varsigma_2 \Uparrow \psi_2'}{\Delta;\, \Gamma \vdash \lambda x{:}\tau_1.\, e_2 \Uparrow \tau_1 \Rightarrow \tau_2 \blacktriangleleft \upsilon_1 \Rightarrow \upsilon_2 \rightsquigarrow \lambda x{:}(\phi_1 \sqcup \upsilon_1).\, \varsigma_2 \Uparrow (\phi_1 \sqcup \upsilon_1) \Rightarrow \psi_2' \dashv \gamma}$$

$$\Uparrow\texttt{Let}\ \frac{\begin{array}{c}\upsilon_2 \neq \square \qquad \Delta;\, \Gamma, x{:}\tau_1 \vdash e_2 \Uparrow \tau_2 \blacktriangleleft \upsilon_2 \rightsquigarrow \varsigma_b \Uparrow \psi_2 \dashv \gamma_2, x{:}\upsilon_1 \\ \Delta;\, \Gamma \vdash e_1 \Uparrow \tau_1 \blacktriangleleft \upsilon_1 \rightsquigarrow \varsigma_d \Uparrow \psi_1 \dashv \gamma_1 \qquad \Delta;\, \Gamma, x{:}\psi_1 \vdash \varsigma_b \Uparrow \psi_2'\end{array}}{\Delta;\, \Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \Uparrow \tau_2 \blacktriangleleft \upsilon_2 \rightsquigarrow \mathbf{let}\ x = \varsigma_d\ \mathbf{in}\ \varsigma_b \Uparrow \psi_2' \dashv \gamma_1 \sqcup \gamma_2}$$

$$\Uparrow\texttt{TLam}\ \frac{\Delta, \alpha;\, \Gamma \vdash e \Uparrow \tau \blacktriangleleft \upsilon \rightsquigarrow \varsigma \Uparrow \psi' \dashv \gamma}{\Delta;\, \Gamma \vdash \Lambda\alpha.\, e \Uparrow \forall\alpha.\, \tau \blacktriangleleft \forall\alpha.\, \upsilon \rightsquigarrow \Lambda\alpha.\, \varsigma \Uparrow \forall\alpha.\, \psi' \dashv \gamma}$$

$\Uparrow$`TApp`: see Section 7.5.3.

$\Uparrow$`Case`: see Section 7.6.

Figure 21: Term-minimal slice calculus rules.

Slicing/Synthesis/FixedAssmsCalc

7.5 ◯ Calculating Term-Minimal Slices

As mentioned above, restricting to term-minimality allows products to be calculated by pairing sub-slices at the corresponding sub-queries, i.e. for $\upsilon_1 \times \upsilon_2$, slice the first component at υ_1, similarly for the second. More generally, if a typing rule has sub-terms which synthesise types, then recursively slice them. Otherwise, if a sub-term analyses against a type, it can be omitted entirely as it does not *provide* type information (only checking against existing type information).

The cases that deserve commentary are the binding constructs, function application, and type application; the $\Uparrow$`Lam:` and $\Uparrow$`Let` rules each carry a body re-typing premise (rightmost) which re-derives the body's typing under the binder slice $\gamma(x)$.

7.5.1 ◯ Function Application

For example, for function application, omit the argument, and slice the function by $\square \Rightarrow \upsilon$, the argument type is not required to determine the return type:

$$\Uparrow\texttt{App}\ \frac{\Delta;\,\Gamma \vdash e_2 \Downarrow \tau_1 \qquad \upsilon \neq \square \qquad \Delta;\,\Gamma \vdash e_1 \Uparrow \tau_1 \Rightarrow \tau_2 \blacktriangleleft \square \Rightarrow \upsilon \rightsquigarrow \varsigma_{fn} \Uparrow \psi_1 \dashv \gamma}{\Delta;\,\Gamma \vdash e_1\, e_2 \Uparrow \tau_2 \blacktriangleleft \upsilon \rightsquigarrow \varsigma_{fn}(\square) \Uparrow \mathsf{cod}(\psi_1) \dashv \gamma}$$

7.5.2 ◯ Lambdas, Lets, and Type Abstractions

Rules which bind variables are more complex, as they abstract a variable used in the minimised body. Slice them by first slicing the body, extracting the assumption it needed for the bound variable from γ, then slice the definition / type annotation using this. Notice that this is the *opposite* direction to typechecking.[10] The corresponding rules are $\Uparrow$`Lam:`, $\Uparrow$`Let`, and $\Uparrow$`TLam`.

7.5.3 ◯ 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 $\widehat{\upsilon}$, where each α-position of the query is replaced by α and non-α positions are carried through unchanged. Figure 22 illustrates the construction on a query whose two α-positions give differing constraints.

$$e \Uparrow \forall\alpha.\,(\alpha \times \alpha \times \mathbf{Bool}), \quad \text{applied at} \quad \sigma = * \Rightarrow (\mathbf{Bool} \times \mathbf{Bool}):$$

$$\text{So } e\langle\sigma\rangle \Uparrow (* \Rightarrow (\mathbf{Bool} \times \mathbf{Bool})) \times (* \Rightarrow (\mathbf{Bool} \times \mathbf{Bool})) \times \mathbf{Bool}.$$

$$\textbf{Query}\ \ \upsilon = \underbrace{* \Rightarrow \square}_{\alpha} \times \underbrace{\square \Rightarrow (\mathbf{Bool} \times \square)}_{\alpha} \times \square$$

$$\textbf{Matched query}\ \ \widehat{\upsilon} = \alpha \times \alpha \times \square$$

$$\textbf{Constraints}\ \ * \Rightarrow \square \sqsubseteq \alpha \qquad \square \Rightarrow (\mathbf{Bool} \times \square) \sqsubseteq \alpha$$

$$\textbf{Constraints Join}\ \ \hat{\sigma} = * \Rightarrow (\mathbf{Bool} \times \square)$$

Figure 22: Type application.

[10] e.g. let bindings type the definition first, then use the result to type the body.

Slicing/Synthesis/FixedAssmsSlicing

7.6 △ Fixed Point Algorithm for Case Expressions

This is the complex case, minimal slices must have no redundant type information from both branches, so the query must be split between the branches. However, simply splitting the query disjointly between the branches is not enough to ensure minimality.

The Overlap Problem. Suppose a case expression **case** e **of** $x.\, e_1 \mid y.\, e_2$ synthesises $\tau_1 \sqcup \tau_2$. Suppose we disjointly split a query $\upsilon \in \lfloor \tau_1 \sqcup \tau_2$ into $\upsilon_1 \in \lfloor \tau_1 \rfloor$ and $\upsilon_2 \in \lfloor \tau_2 \rfloor$. Then slice the branches, but as minimal slices are not necessarily exact (theorem 4.3), then they may synthesise strictly more prescise ψ_1 and ψ_2. In this situation, branch 1 actually only needed to cover what ψ_2 didn't cover, which is stricly less than the υ_1 it was queried under, so it may not be minimal on this, and vice versa for branch 2.

(a) Disjoint split. **(b)** Residual queries.

Figure 23: Abstract View of Overlap and Residuals.

7.6.1 Branch Residuals

When exactly are two branch slices jointly minimal? Let υ_1 and υ_2 be the branch queries, then when (see Figure 23b):

- $\upsilon_1 \sqsubseteq \upsilon \setminus \psi_2$. Branch 1 is queried on not more than is *actually* provided by branch 2 or contained within υ.
- $\upsilon_2 \sqsubseteq \upsilon \setminus \psi_1$. Branch 2 is queried on not more than is *actually* provided by branch 1 or contained within υ.
- $\upsilon \sqsubseteq \psi_1 \sqcup \psi_2$. The joined branch types actually cover the query.

It is still possible for ψ_1 and ψ_2 to overlap, but any overlap is not contained in the queries, so the branches are still minimal.

Iteration. However, these are *mutually dependent* slices, so they need iteration to solve. Let υ_1^i be the query on the ith iteration, (σ_1^i, ψ_1^i) be the resulting slice and similarly for branch 2. Then iterate setting $\upsilon_1^1 = \upsilon \setminus \psi_2^0$, then setting $\upsilon_2^1 = \upsilon \setminus \psi_1^1$. Starting at $(\psi_1^0, \psi_2^0) = (\top, \square)$. We wish to show that ψ_1 decreases monotonically and ψ_2 increases monotonically.

As co-Heyting subtraction is anti-monotone in its second argument (theorem 2.1), then as ψ_2 increases,[11] υ_1 decreases and as ψ_1 decreases, υ_2 increases. At the start, $\psi_2^0 = \square$, so ψ_2 increases on the first step, so υ_1 decreases on the first step.

By monotonicity of minimum synthesis slices (theorem 4.8), we can find σ_1 decreasing and hence ψ_1 decreasing. Hence, υ_2 increases (by anti-monotonicity). However, an upwards monotonicity theorem, which would conclude the next step, that ψ_2 monotonically increases,

[11] All use of increase/decrease here is in the monotonic sense.

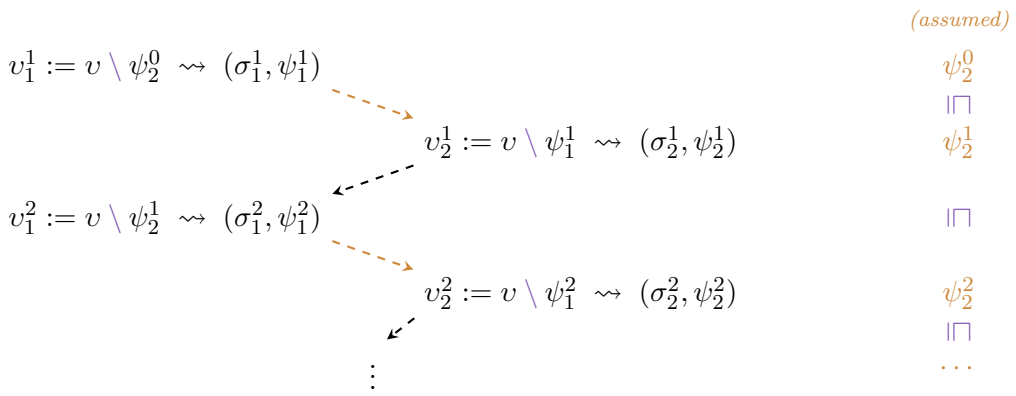


Figure 24: Branch Fixed Point Iteration

does *not* hold (theorem 7.5). But, when it does, v_1 would decrease inductively, completing the loop.

Hence, *when the chain ψ_2 monotonically increases*, then ψ_1 monotonically decreases. So, by Kleene's finite fixed point theorem (theorem 2.2) this iteration leads to a fixed point with $v_1 = v \setminus \psi_2$ and $v_2 = v \setminus \psi_1$, satisfying the first two constraints for jointly minimal branches. The third condition $v \sqsubseteq \psi_1 \sqcup \psi_2$ then follows from validity of the branch slices ($v_1 \sqsubseteq \psi_1$):

$$\begin{aligned} v &\sqsubseteq \psi_2 \sqcup (v \setminus \psi_2) && \text{(co-Heyting: } a \sqsubseteq b \sqcup (a \setminus b)\text{)} \\ &= \psi_2 \sqcup v_1 && \text{(definition of } v_1\text{)} \\ &\sqsubseteq \psi_2 \sqcup \psi_1 && \text{(by } v_1 \sqsubseteq \psi_1\text{).} \end{aligned}$$

¬upward-closure-min-fixedassms

Counterexample 7.5 (◯ Upwards Non-Monotonicity of Term-Minimal Slices). *Fix* $\Gamma = (z : * \times *)$ *and* $D : \Delta; \Gamma \vdash \mathbf{if}\ \square\ \mathbf{then}\ z\ \mathbf{else}\ (*, \square) \Uparrow * \times *$. *The less precise query* $v' = * \times \square$ *admits the following term-minimal slice:*

assumptions

$z : * \times *$

if $\square$ **then** $\square$ **else** $(*, \square)$

The more precise query $v = * \times *$, *by contrast, has a* unique *term-minimal slice – synthesising* $* \times *$ *forces use of* z*:*

assumptions

$z : * \times *$

if $\square$ **then** z **else** $\square$

The two are incomparable.

Therefore, this part of the algorithm is partial. In reality, this covers most cases, and you can revert to the brute force algorithm otherwise. Devising a complete (non-brute force) algorithm is left to future work. Another potential downside of this algorithm is that it is *biased*, it prefers to select typing information from branch 1.

7.6.2 Scrutinee Descent

The branches are now jointly minimal under the full context. But, case statements also include *bindings*, whose type are sourced by another term. However, unlike with let bindings, we can't simply take the minimal assumptions of the branches, because those may only provide enough to synthesise the residual queries υ_i, which may not cover the original query υ.[12]

As case scrutinee terms are typically small in real world code (e.g. if statements), we can simply do a brute-force descent on the scrutinee. But, if the term is significantly larger than the type, it makes sense to can perform a brute-force descent on the bound types to find a minimal pair of assumptions to cover the query υ: that is, $x : \phi_1; \Gamma \vdash \sigma_1 \vdash \psi_1'$ and $y : \phi_2; \Gamma \vdash \sigma_2 \vdash \psi_2'$ with $\upsilon \sqsubseteq \psi_1' \sqcup \psi_2'$; this can be done by descending upon the type (hence why this is only useful when the type is smaller than the term). Then slice the scrutinee according to $\phi_1 + \phi_2$ and use this as a smaller starding point to descend on.

mincase-cov ◯ **Optimisation:** However, if $\upsilon \sqsubseteq \upsilon_1 \sqcup \upsilon_2$ already, then you can do the same as the let binding case: slice the scrutinee according to the (two) minimal assumptions required for the branch bindings, and derive the minimal assumptions by joining the scrutinee and branches minimal assumptions.

7.6.3 The Assumption Slice

At this point we have a term-minimal slice, but we still need a minimal assumption slice. Again, this does not directly follow from the minimal assumptions for the branches individually, due to *aliasing* of common assumptions. However, you can join these individual minima alongside the minimal assumptions for the scrutinee (calculated by brute-force[13] or over-approximated using the slice on $\phi_1 + \phi_2$). This gives a set of valid assumptions to synthesise υ which will be *near-minimum* and work as a good starting point to brute-force from.

7.7 △ Calculating Analysis Slices

Analysis slices are calculated directly from synthesis slices. The only rule the creates a type checking judgement is function application. Consider the analysis slice of a function argument on υ_2; it is just the function application and a slice of the function under $\square \to \upsilon_2$.

The key difference is that analysis slices are *constructed from the focus upwards*. An unannotated function is an *inner* (theorem 5.2) analysis slice; so it defers calculation of its outer context by returning a minimal analysis type υ_{outer} to impose upon the unannotated function which still imposes the original query, υ, on the focus, i.e. $\upsilon_{outer} \triangleq \square \to \upsilon$. The other analysis rules are similar. The only additional complexity is for binding constructs which also must slice the bindings according to the assumptions required for the inner analysis slice.

[12] We only have that $\upsilon \sqsubseteq \psi_1 \sqcup \psi_2$.
[13] *Ascending* from $\emptyset$ being more efficient here.

8 △ Minimum-Size Slicing

Legend: ◯ fully mechanised, ◬ mechanised with postulates, △ not mechanised, ▭ important theorem, ▭ counterexample.

We can calculate a *minimal* synthesis slice in polynomial time (section 4.3). However, minimal slices are not unique, so a natural extension is to rank them by *size* and ask for a minimal slice of *minimum size*. This chapter reduces the *set cover optimisation problem* to calculating minimum size slice, meaning minimum-size slicing is **NP-hard**. I (informally) prove this for term-minimal synthesis slices, and show that the argument extends to general minimal synthesis slices and to analysis slices.

8.1 △ Definitions

Definition 8.1 (Size of an Expression Slice)**.** For an expression slice $\varsigma \in \lfloor e \rfloor$, the size $|\varsigma|$ counts non-$\Box$ syntactic positions:

$$|\Box| = 0, \qquad |*| = 1, \qquad |x| = 1, \qquad |c(\varsigma_1, \ldots, \varsigma_k)| = 1 + \textstyle\sum_i |\varsigma_i|.$$

Definition 8.2 (Minimum-Size TermMinimal Slice Problem (Min-TM-Slice))**.** Given a derivation $D : \Delta; \Gamma \vdash e \Uparrow \tau$ and query $\upsilon \in \lfloor \tau \rfloor$, the minimum-size term-minimal slice problem is to compute

$$\mathrm{minSize}(D, \upsilon) \triangleq \min\{\, |\sigma| \;:\; \sigma \in \mathit{TermMin}\ D \blacktriangleleft \upsilon \,\}.$$

Definition 8.3 (Minimum Set Cover (Optimisation) Problem (Min-Set-Cover))**.** Given a universe $U = \{1, \ldots, n\}$ and a family $S = S_1, \ldots, S_m \subseteq U$, compute

$$K^\star(U, S) \triangleq \min\{\, |J| \;:\; J \subseteq \{1, \cdots, m\},\ \textstyle\bigcup_{j \in J} S_j = U \,\}.$$

The minimum set cover optimisation problem is NP-hard [19, §16.1, p. 414], hence any polynomial reduction from it, is also NP-hard.

8.2 △ The reduction

Definition 8.4 (Reduction from Min-set-cover)**.** Given an instance $(U, S_1, \ldots, S_m)$ of Min-Set-Cover, suppose wlog. $|U| = n$ and $\bigcup_j S_j = U$. Then construct $D : \Delta; \Gamma \vdash e \Uparrow \tau$ and query $\upsilon = \tau$ as follows:

- **Target type τ:** $\tau = \underbrace{* \times (* \times (\cdots \times (* \times *)))}_{n \text{ projections}}$. Position i of τ corresponds to element i of U.
- **Encoding $S_j \mapsto T_j$:** For each j, construct type slice family $T_j \in \lfloor \tau \rfloor$ by omitting all projections i for $i \notin S_j$. Then $\bigsqcup_j T_j = \tau$ iff $\bigcup_j S_j = U$, where $\bigsqcup_j$ is the finite join of family T_j.
- **Assumption context Γ:** $\Gamma = (x_1 : T_1,\ \ldots,\ x_m : T_m)$, so each x_j synthesises T_j exactly under $\Uparrow\texttt{Var}$.
- **Let-chain leaves ℓ_j:** For each j, define a chain of $m - 1$ let-bindings that thread x_j through fresh names:

$$\ell_j \triangleq \textbf{let}\ n_1 = x_j\ \textbf{in let}\ n_2 = n_1\ \textbf{in}\ \cdots\ \textbf{let}\ n_{m-1} = n_{m-2}\ \textbf{in}\ n_{m-1}$$

ℓ_j synthesises T_j by repeated application of ⇑`Let`, and has size $2m-1$. Any strict sub-slice of ℓ_j breaks the chain and synthesises only $\square$, so ℓ_j behaves as an all-or-nothing unit.

- **Case tree e:** Construct any binary case tree with leaves $\ell_1, \ldots, \ell_m$, scrutinees $\square$, and binders x.

  By theorem 3.4 the leaves' synthesised types are mutually consistent, so the case rule applies at every node, giving $\Delta; \Gamma \vdash e \Uparrow \bigsqcup_j T_j = \Delta; \Gamma \vdash e \Uparrow \tau$. The tree has size $|e| = m(2m-1) + (m-1) = 2m^2 - 1$.

- **Query:** $\upsilon = \tau$.

This construction is trivially computable in polynomial time, $\mathcal{O}(n + m^2 + \sum_j |S_j|)$ (quadratic in m because of the per-leaf let chains).

Concretely, view each set S_j as a *variable* x_j whose type is a slice of the target carrying $*$ exactly where S_j has an element. The chain ℓ_j at each leaf makes retaining x_j in the slice cost $2m-1$ (the full chain) and omitting it cost 0, for minimal slices.[14] The per-leaf cost $2m-1$ strictly dominates the (at most) $(m-1)$ variation in retained case nodes, so the minimum-size slice agrees with the minimum set cover. Figure 25 draws this correspondence on a 4-element universe.

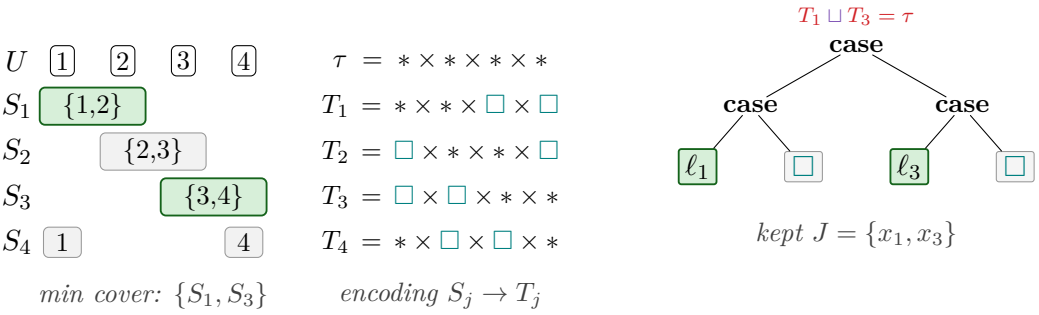


Figure 25: Set cover (left), the encoding (middle), and the induced case-tree slice (right), with a minimal set cover selected.

8.3 △ Correctness

Throughout this subsection, write $J(\sigma) = \{j : \ell_j \text{ is fully retained in } \sigma\}$ for the set of leaves whose entire let-chain is kept.

Lemma 8.5 (Validity is Set-Cover). *A term-minimal slice σ of D at $\upsilon = \tau$ is valid iff $J(\sigma)$ is a set cover of U.*

Proof. A fully retained chain ℓ_j synthesises T_j; any strict sub-slice of ℓ_j (or a fully gapped leaf) synthesises $\square$. Walking up the case tree, the case rule joins branch types pointwise, so σ synthesises $\bigsqcup_{j \in J(\sigma)} T_j$. Validity ($\sqsupseteq \tau$) holds iff every position of τ is covered: iff $\bigcup_{j \in J(\sigma)} S_j = U$. □

Lemma 8.6 (Minimum Slice Size). *For (Γ, e, τ) encoding Min-Set-Cover problem (U, S), every minimum-size valid slice σ has $|J(\sigma)| = K^\star(U, S)$. Hence $K^\star(U, S)$ is recovered from such a σ by counting retained chains, in time $\mathcal{O}(|\sigma|) = \mathcal{O}(m^2)$.*

[14] Partially slicing any part of the chain is never minimal.

Proof. A minimum-size term-minimal slice σ' is associated with the set $J(\sigma')$ by counting the retained leaves. This requires a single loop over the case tree, whose size is quadratic in m, so $J(\sigma')$ is computable in $\mathcal{O}(m^2)$ time.

Any minimal slice σ' has size $|J(\sigma')| \cdot (2m-1) + k$ where $0 < k < m$ counts the number of case statements in the slice. Therefore, suppose σ' corresponds to a non-minimal solution to set-cover, then by theorem 8.5, some σ has $|J(\sigma)| < |J(\sigma')|$, hence:

$$\begin{aligned}
|\sigma| &< |J(\sigma)| \cdot (2m-1) + m && \text{(bound on case-node count)} \\
&\leq (|J(\sigma')| - 1) \cdot (2m-1) + m && \text{(by } |J(\sigma)| < |J(\sigma')|\text{)} \\
&= |J(\sigma')| \cdot (2m-1) - m + 1 && \text{(arithmetic)} \\
&\leq |\sigma'| && \text{(bound on case-node count).}
\end{aligned}$$

Hence, σ' would not be a minimum-size slice; by contradiction, a minimum-sized σ' must therefore correspond to a minimum set cover. □

Theorem 8.7 (Minimum-Size Term-Minimal-Slicing is NP-hard). *Computing* $\mathrm{minSize}(D, \upsilon)$ *is NP-hard.*

Proof. Definition 8.4 is a polynomial-time reduction from Min-Set-Cover to Min-TM-Slice with Lemma 8.6 recovering $K^\star(U, S)$ from $\mathrm{minSize}(D, \upsilon)$ in polynomial time. Hence, NP-hardness of Min-Set-Cover [19, §16.1, p. 414] transfers onto Min-TM-Slice. □

8.4 △ Extension to Minimal Synthesis Slices

The reduction above can be embedded also into minimum synthesis slices in general, by forcing the minimal assumptions slice to be full by producting the case tree with each of the variables x_j and querying each set in this product fully. This forces every variable to be used fully in the assumption slice.

Definition 8.8 (Reduction to Min-Syn-Slice)**.** Let (Γ, e_0, τ) be the term-minimal encoding of Definition 8.4. Form the product expression and query

$$e \triangleq (e_0,\, x_1,\, x_2,\, \ldots,\, x_m), \qquad \upsilon \triangleq \tau \times T_1 \times T_2 \times \cdots \times T_m,$$

under the same assumption context $\Gamma = (x_1 : T_1, \ldots, x_m : T_m)$.

Lemma 8.9 (Maximal Assumption Slice). *For every valid slice* $\rho = (\gamma, \varsigma)$ *of* $\Delta; \Gamma \vdash e \Uparrow \upsilon$ *at* υ*,* $\gamma = \Gamma$*.*

Proof. The $(j+1)$-th projection is x_j and is queried at T_j. This projection must synthesise T_j under γ via $\Uparrow$`Var`. Hence, we require $\gamma(x_j) = T_j$, so $\gamma = \Gamma$. □

Theorem 8.10 (Minimum-Size Syn-Slice is NP-hard). *Computing* $\min\{|\rho| : \rho \in \mathit{SynSlice}\ D \blacktriangleleft \upsilon\}$ *is NP-hard.*

Proof. By Lemma 8.9, $|\gamma| = |\Gamma|$ is fixed by the input, and the outer-tuple variable positions $x_1, \ldots, x_m$ contribute a further fixed m. Hence $|\rho| = |\Gamma| + m + |\varsigma_0|$, where ς_0 slices the case-tree component e_0, and minimising $|\rho|$ reduces to the term-minimal problem on (e_0, τ). The construction is polynomial in $n + m$, and NP-hardness transfers from Theorem 8.7. □

9 Relation to Other Forms of Slicing

Program slicing of various different forms, relating to dynamic execution, data dependencies, and type errors, has been studied extensively. In this chapter I compare type slicing with three other program slicing traditions: classical *program slicing* [35, 32], *Galois slicing* [23], and *type error slicing* [34, 15]. Type slicing differs from all three, but has some similarities to Galois slicing.

9.1 Classical Program Slicing

Classical program slicing [35, 32] was developed for imperative programs. Slices are syntactically obtained by deleting lines in the code, which is less flexible than omitting arbitrary sub-terms in a syntax tree as used here. The equivalent of type queries is a 'slicing criterion', which is a (program-point, variable) pair, with the slice preserving the runtime value of the chosen variable at the chosen point. Type queries have a more complex, compound structure as compared to simple variables, and slices cannot preserve the type query in general (theorem 4.3), so we adopt the notion of *validity* instead of preservation.

Program slicing has static [35] and dynamic [18] variants, depending on whether the slicing criterion is preserved over *all* inputs to the program or a particular execution. As typing is deterministic (theorem 3.17), this distinction does not apply to type slicing.

9.2 Galois Slicing

Galois slicing [23] is a lattice-theoretic version of type slicign that has subsequently been applied and related to a wide range of calculi and user interfaces [29, 22, 24, 3].

Type slicing is related in the use of lattices and precision. However, Galois slices define a Galois connection, which unfortunately does not exist for type slicing.

9.2.1 Background: Galois connections and the adjoint functor theorem

For context, an order-theoretic Galois connection is defined as follows:

Definition 9.1 (Galois connection)**.** Let $(P, \sqsubseteq_P)$ and $(Q, \sqsubseteq_Q)$ be posets. A pair of monotone maps $f : P \to Q$ and $g : Q \to P$ is a *Galois connection*, written $f \dashv g$ with f the lower adjoint and g the upper adjoint, when for every $p \in P$ and $q \in Q$

$$f(p) \sqsubseteq_Q q \iff p \sqsubseteq_P g(q).$$

Proposition 9.2 (Adjoint Functor Theorem for Complete Lattices [13, Prop. 7.34])**.** *Let P, Q be complete lattices. A monotone map $g : Q \to P$ is the upper adjoint of a (necessarily unique) Galois connection if and only if g preserves all meets. When g preserves meets, the lower adjoint f is given pointwise by*

$$f(p) = \bigsqcap \{\, q \in Q \mid g(q) \sqsupseteq p \,\}.$$

9.2.2 Slicing

Perera et al. work in a functional setting in which a program e is evaluated under an environment ρ to a value v, written $\rho, e \Downarrow v$. For a fixed such terminating execution, they form the slice lattice $\lfloor \rho, e \rfloor$ of partial programs below (ρ, e) and the slice lattice $\lfloor v \rfloor$ of partial values

below v (Perera et al. refer to these as prefixes, Prefix(ρ, e) and Prefix(v)). Type slices have the same shape as Perera's slices: a partial program (here the slice ρ) paired with a partial value (here the synthesised type ψ), mediated by a derivation (here the typing derivation).

Restricted to the slice lattice below the executed program, evaluation lifts to a function

$$eval_{\rho,e} : \lfloor \rho, e \rfloor \to \lfloor v \rfloor$$

which is total, monotone, and crucially meet-preserving:

$$eval_{\rho,e}(p_1 \sqcap p_2) = eval_{\rho,e}(p_1) \sqcap eval_{\rho,e}(p_2).$$

By the adjoint functor theorem, $eval_{\rho,e}$ is then the upper adjoint of a unique Galois connection, with lower adjoint

$$uneval_{\rho,e} : \lfloor v \rfloor \to \lfloor \rho, e \rfloor, \qquad uneval_{\rho,e}(u) = \bigsqcap \{\, p \in \lfloor \rho, e \rfloor \mid eval_{\rho,e}(p) \sqsupseteq u \,\}$$

This $uneval_{\rho,e}$ is the 'backward slice function': it sends a partial-value query u to the meet of all slices whose evaluation suffices to produce u, and that meet is itself in the set, so $uneval_{\rho,e}(u)$ is a least slice for u.

9.2.3 Failure of Meet-Preservation for Type Synthesis

The natural type-slicing analogue of $eval_{\rho,e}$ is the synthesis function

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

which, given a slice $\rho \sqsubseteq (\Gamma, e)$, returns the unique synthesised type of ρ under D. The function is well-defined by syn-unicity, monotone by syn-precision, and exists for any derivation D.

However, the forward map ϕ_D is not meet-preserving in general. The case rule witnesses this: consider a derivation

$$D : \Delta;\, \Gamma \vdash (\mathbf{case}\ e\ \mathbf{of}\ x_1.\, e_1 \mid x_2.\, e_2) \Uparrow \tau$$

in which both branches independently synthesise the same non-gap type τ. The slice s_1 keeping only the first branch and the slice s_2 keeping only the second each yield $\phi_D = \tau$, but their meet $s_1 \sqcap s_2$ omits both branches, giving $\phi_D(s_1 \sqcap s_2) = \square \sqsubset \tau = \phi_D(s_1) \sqcap \phi_D(s_2)$.

9.3 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 [34] and Haack and Wells [15] work in globally inferred Hindley-Milner: the program is reduced to a system of equality constraints over type variables, and a type error slice is computed as a minimal unsatisfiable subset of the constraint set, projected back to the corresponding minimal subset of source positions.

These program slices are not structure preserving programs, they are flat sets of constraints and sources, rather than actual programs. This means the results themselves cannot be type checker nor can they be executed.

There is also a difference in scope. 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 (section 6) [37], which elaborates every well-formed expression into a totally typed derivation, type slicing applies also to ill-typed programs, so "why this type?" subsumes "why this fails to type-check?" as the special case of explaining the type of a marked expression.

10 Further Extensions & Applications

10.1 Further Extensions

10.1.1 Implicit Polymorphism

Globally inferred languages typically have much more confusing typing derivations and type errors than locally inferred languages. Extending the calculus to implicit polymorphism in a bidirectional setting would expand the application significantly. In particular, the ability to explain even well-typed code is particularly useful here as error marks are often localised to the wrong location, resulting the real error being in well-typed code. Adapting the constraint slicing systems of Wand [34] and Haack and Wells [15] to integrate with type slicing is worth exploring.

10.1.2 Set Theoretic Types

Set theoretic types extend the type language with union, intersection, and negation. A term can then carry several types simultaneously, allowing typing of overloaded functions, and typecase constructs, commonly used in real world hybrid typed languages (e.g. TypeScript, or Elixir [10]). These types could be added to extend the slice lattices of this paper (section 3.2), and type slicing could be used to explain the sources of union types by omitting appropriate branches (including typecase structures); similarly, irrelevant overloads to functions would be omitted.

Figure 3 demonstrated a simple example of this by inferring sum types using dynamic types, in much the same way as simple union types would be inferred over case statements.

Gradual set theoretic types have been explored by Castagna [11, 12], and more recently, Ângelo's recent thesis [2] establishes graduality specifically for intersection types with a proof of graduality.

10.1.3 Occurrence Typing

Occurrence typing [33] is another common technique in typing flexible or hybrid type systems. Inside the true branch of `if (typeof x === int) ... else ...`, the type of `x` is inferred as an `int`. These types are similarly difficult to reason about, especially in large code, and, unlike the constraint-based type error slicing surveyed in section 9.3, do genuinely require a syntactically well-formed program slice to reason about the affect of data-flow on the types. Therefore, type slicing is a particularly good, and sufficiently flexible, way to explain types in such a system. However, to my knowledge there is no existing formal research of occurrence typing in a bidirectional setting.

10.2 Further Applications

10.2.1 Type-Modifying Refactorings

Refactoring a term such that its type changes would typically scatter type errors across the refactored locations. Type slicing can be used to semi-automatically fix these type errors.

Many locations to refactor will have syntactically different surrounding contexts, but may have the identical analysis contribution slices (extending contribution slice from section 4.6 to the analogous construct on minimal analysis slices).[15] We could treat these slices as

[15] Formally, we would compare their *minimally scoped* slices, in the sense of the minimally scoped checking contexts in [9].

templates which we define a refactor for, that performs edits to the elements in the minimal slice itself, possibly moving, but not editing, entire folded terms.

These edits could be semi-automated by referring back to the user to fix each template, or otherwise fully automatic by using dedicated solvers or LLMs. In particular, the slices provide only the necessary context to an LLM, restricting edits to code regions actually relevant to the type fixes. LLM edits on incomplete programs with holes is under active research [6] in the Hazel language [20], which this core calculus (section 3) is very similar to.

Figure 26 shows two syntactically distinct usages of a refactored `helper` term who both collapse to the same minimal analysis slice. An edit on that template (e.g. replacing `Int` with `String`) fixes both sites at once.

Changing these annotations will then lead to errors at other use sites not involved in the initial refactor, and we can treat synthesis contribution slices as templates to change in much the same way. By treating these edits as new refactorings, we would loop, fixing errors at each step until none are left (or if making no progress).

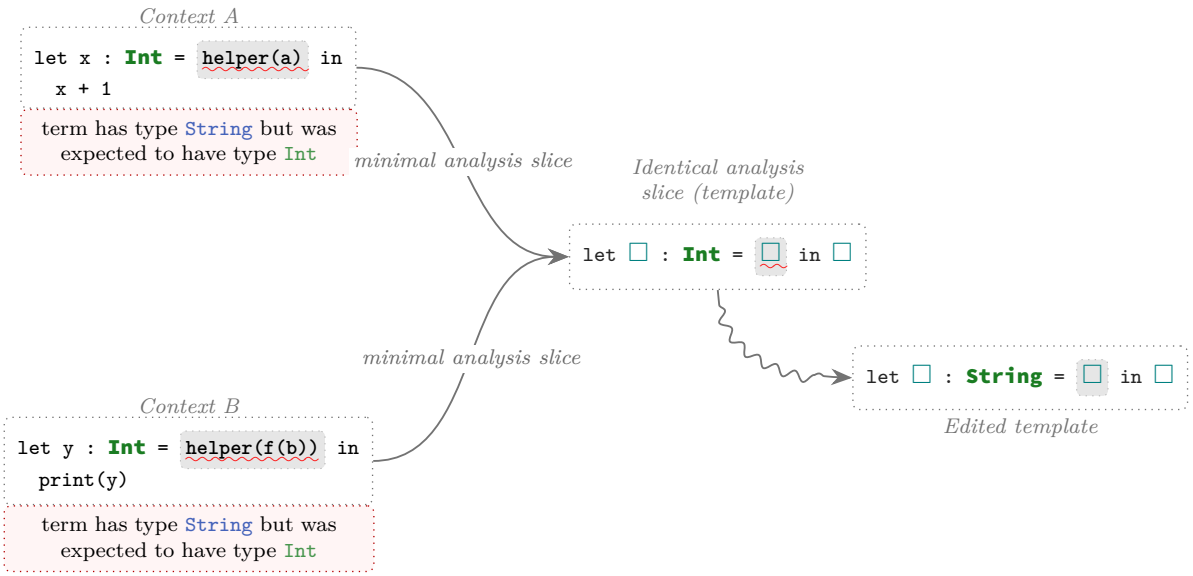


Figure 26: Two distinct contexts with the same analysis slice, performing the same (initial) fix.

10.2.2 Cast Slicing & Dynamics

A gradually-typed language elaborates source programs into a cast calculus and runs the resulting casts dynamically. Type slices can be attached directly to those casts. Hence, casts can then explain why they were inserted by pointing back to code; the source will be a slice of the term being cast, and the target a slice of its context. Approximate minimal slices can be tagged directly onto types allowing them to be decomposed at runtime, allowing dynamic casts to point back to code. This is particularly useful for debugging dynamic type errors, which do not typically point back to code, but do when carrying type slices. However, the strong properties of slices being slices of the original program is not maintained during dynamic execution. Tracking dynamic execution, perhaps integrating ideas from Perera [23, 22], would allow users to trace back casts to source code with stronger guarantees. This application is detailed thoroughly in Carroll [8].

11 Conclusions

This dissertation presents the type slicing theory for a bidirectional calculus with explicit System F polymorphism, bindings, products, and sums, mechanising a large majority of the theory in Agda. A more rigorous account of typing assumptions involved in slices slices was considered with extensive exploration of provably correct algorithms for calculating type slices, including various optimisations.

The theory is formally integrated with the error-marking calculus of Zhao et al. [37], formally connecting it to ill-typed programs, justifying its *sound* use in debugging *all* static type errors of a program.

Type slicing was discussed to be substantially different from previous program slicing techniques, of which there are several varieties, none of which being directly applicable to the problem of explaining types in bidirectional type systems. Finally, further extensions and applications were proposed, which would place type slicing as a promising tool for use in practical gradually typed languages after extensions, both for explaining static *and dynamic* types and for code refactoring.

References

[1] Abdulaziz Alaboudi and Thomas D. LaToza. *An Exploratory Study of Debugging Episodes*. 2021. arXiv: 2105.02162 [cs.SE]. url: https://arxiv.org/abs/2105.02162.

[2] Pedro Jorge Fernandes Ângelo. "Gradual Intersection Types". PhD thesis. Faculdade de Ciências, Universidade do Porto, 2024.

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

[4] Moritz Beller et al. "On the dichotomy of debugging behavior among programmers". In: Association for Computing Machinery, 2018. doi: 10.1145/3180155.3180175.

[5] Garrett Birkhoff. *Lattice Theory*. Vol. 25. Colloquium Publications. American Mathematical Society, 1940. doi: 10.1090/coll/025.

[6] Andrew Blinn et al. "Statically Contextualizing Large Language Models with Typed Holes". In: *Proceedings of the ACM on Programming Languages* 8.OOPSLA2 (Oct. 2024). issn: 2475-1421. doi: 10.1145/3689728. url: http://dx.doi.org/10.1145/3689728.

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

[8] Max Carroll. *Decomposable Type Highlighting for Bidirectional Type Slicing*. Part II Dissertation, University of Cambridge. 2024.

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

[10] Giuseppe Castagna, Guillaume Duboc, and José Valim. "The Design Principles of the Elixir Type System". In: *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. "Gradual Typing with Union and Intersection Types". In: *Proceedings of the ACM on Programming Languages* 1.ICFP (2017), 41:1–41:28. doi: 10.1145/3110285.

[12] Giuseppe Castagna et al. "Gradual Typing: A New Perspective". In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), 16:1–16:32. doi: 10.1145/3290329.

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

[14] Jana Dunfield and Neelakantan R. Krishnaswami. "Bidirectional Typing". In: *ACM Computing Surveys* 54.5 (2022), 98:1–98:38. doi: 10.1145/3450952.

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

[16] Robert Harper. *Practical Foundations for Programming Languages*. 2nd ed. Cambridge University Press, 2016. isbn: 9781107150300. doi: 10.1017/cbo9781316576892.

[17] Gérard Huet. "The Zipper". In: *Journal of Functional Programming* 7.5 (1997), pp. 549–554. doi: 10.1017/S0956796897002864.

[18] Bogdan Korel and Janusz Laski. “Dynamic Program Slicing”. In: *Information Processing Letters* 29.3 (1988), pp. 155–163. doi: 10.1016/0020-0190(88)90054-3.

[19] Bernhard Korte and Jens Vygen. *Combinatorial Optimization: Theory and Algorithms*. 6th ed. Vol. 21. Algorithms and Combinatorics. See §16.1 (p. 414) for NP-hardness of Minimum Set Cover. Springer, 2018. doi: 10.1007/978-3-662-56039-6.

[20] Cyrus Omar et al. “Live functional programming with typed holes”. In: *Proceedings of the ACM on Programming Languages* 3.POPL (Jan. 2019), pp. 1–32. issn: 2475-1421. doi: 10.1145/3290327. url: http://dx.doi.org/10.1145/3290327.

[21] John-Paul Ore et al. “Assessing the Type Annotation Burden”. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE 2018. ACM, 2018, pp. 190–201. doi: 10.1145/3238147.3238173.

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

[23] Roly Perera et al. “Functional Programs that Explain Their Work”. In: *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 2012, pp. 365–376. doi: 10.1145/2364527.2364579.

[24] Roly Perera et al. “Linked Visualisations via Galois Dependencies”. In: *Proceedings of the ACM on Programming Languages* 6.POPL (2022). doi: 10.1145/3498668.

[25] Benjamin C. Pierce. *Types and Programming Languages*. 1st. The MIT Press, 2002. isbn: 0262162091.

[26] Benjamin C. Pierce and David N. Turner. “Local type inference”. In: *ACM Transactions on Programming Languages and Systems* 22.1 (Jan. 2000), pp. 1–44. issn: 1558-4593. doi: 10.1145/345099.345100. url: http://dx.doi.org/10.1145/345099.345100.

[27] Cecylia Rauszer. “Semi-Boolean algebras and their applications to intuitionistic logic with dual operations”. In: *Fundamenta Mathematicae* 83.3 (1974), pp. 219–249. doi: 10.4064/fm-83-3-219-249.

[28] Gonzalo E. Reyes and Houman Zolfaghari. “Bi-Heyting algebras, toposes and modalities”. In: *Journal of Philosophical Logic* 25.1 (1996), pp. 25–43. doi: 10.1007/BF00357841.

[29] Wilmer Ricciotti et al. “Imperative Functional Programs that Explain Their Work”. In: *Proceedings of the ACM on Programming Languages* 1.ICFP (2017). doi: 10.1145/3110258.

[30] Jeremy G. Siek and Walid Taha. “Gradual Typing for Functional Languages”. In: *Scheme and Functional Programming Workshop*. 2006, pp. 81–92.

[31] Jeremy G. Siek et al. “Refined Criteria for Gradual Typing”. In: *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Vol. 32. LIPIcs. 2015, pp. 274–293. doi: 10.4230/LIPIcs.SNAPL.2015.274.

[32] Frank Tip. “A Survey of Program Slicing Techniques”. In: *Journal of Programming Languages* 3.3 (1995), pp. 121–189.

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

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

[35] Mark Weiser. “Program Slicing”. In: *IEEE Transactions on Software Engineering* SE-10.4 (1984), pp. 352–357. doi: 10.1109/TSE.1984.5010248.

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

[37] Eric Zhao et al. “Total Type Error Localization and Recovery with Holes”. In: *Proceedings of the ACM on Programming Languages* 8.POPL (2024). doi: 10.1145/3632910.

Semantics/Statics/CtxTyping, Semantics/Marking/CtxMarking

A ◯ Context Classification: Full Rule Set

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

$$s\bigcirc\ \frac{}{\Delta;\,\Gamma \vdash \bigcirc\ \mathbf{at} \Uparrow \tau \triangleright \Delta;\,\Gamma\,[\Uparrow \tau]} \qquad a\bigcirc\ \frac{}{\Delta;\,\Gamma \vdash \bigcirc\ \mathbf{at} \Downarrow \tau \triangleright \Delta;\,\Gamma\,[\Downarrow \tau]}$$

$$aSub\ \frac{\Delta;\,\Gamma \vdash \mathcal{C}\ \mathbf{at} \Uparrow \tau' \triangleright \Delta';\,\Gamma'\,[m] \qquad \tau \sim \tau'}{\Delta;\,\Gamma \vdash \mathcal{C}\ \mathbf{at} \Downarrow \tau \triangleright \Delta';\,\Gamma'\,[m]}$$

$$s\lambda:\ \frac{\Delta \vdash \tau_1\ \mathsf{wf} \qquad \Delta;\,\Gamma, x{:}\tau_1 \vdash \mathcal{C}\ \mathbf{at} \Uparrow \tau_2 \triangleright \Delta';\,\Gamma'\,[m]}{\Delta;\,\Gamma \vdash \lambda x{:}\tau_1.\,\mathcal{C}\ \mathbf{at} \Uparrow \tau_1 \Rightarrow \tau_2 \triangleright \Delta';\,\Gamma'\,[m]}$$

$$a\lambda:\ \frac{\tau \sim \tau_1 \Rightarrow \square \qquad \tau \sqcup \tau_1 \Rightarrow \square \equiv \tau_1 \Rightarrow \tau_2 \qquad \Delta \vdash \tau_1\ \mathsf{wf} \qquad \Delta;\,\Gamma, x{:}\tau_1 \vdash \mathcal{C}\ \mathbf{at} \Downarrow \tau_2 \triangleright \Delta';\,\Gamma'\,[m]}{\Delta;\,\Gamma \vdash \lambda x{:}\tau_1.\,\mathcal{C}\ \mathbf{at} \Downarrow \tau \triangleright \Delta';\,\Gamma'\,[m]}$$

$$a\lambda\Rightarrow\ \frac{\tau \sqcup \square \Rightarrow \square \equiv \tau_1 \Rightarrow \tau_2 \qquad \Delta;\,\Gamma, x{:}\tau_1 \vdash \mathcal{C}\ \mathbf{at} \Downarrow \tau_2 \triangleright \Delta';\,\Gamma'\,[m]}{\Delta;\,\Gamma \vdash \lambda x.\,\mathcal{C}\ \mathbf{at} \Downarrow \tau \triangleright \Delta';\,\Gamma'\,[m]}$$

Figure 27: Unmarked context classification: holes, subsumption, and lambdas.

$$s\circ_1\ \frac{\Delta;\,\Gamma \vdash \mathcal{C}\ \mathbf{at} \Uparrow \tau \triangleright \Delta';\,\Gamma'\,[m] \qquad \tau \sqcup \square \Rightarrow \square \equiv \tau_1 \Rightarrow \tau_2 \qquad \Delta;\,\Gamma \vdash e \Downarrow \tau_1}{\Delta;\,\Gamma \vdash \mathcal{C} \circ_1 e\ \mathbf{at} \Uparrow \tau_2 \triangleright \Delta';\,\Gamma'\,[m]}$$

$$s\circ_2\ \frac{\Delta;\,\Gamma \vdash e \Uparrow \tau \qquad \tau \sqcup \square \Rightarrow \square \equiv \tau_1 \Rightarrow \tau_2 \qquad \Delta;\,\Gamma \vdash \mathcal{C}\ \mathbf{at} \Downarrow \tau_1 \triangleright \Delta';\,\Gamma'\,[m]}{\Delta;\,\Gamma \vdash e \circ_2 \mathcal{C}\ \mathbf{at} \Uparrow \tau_2 \triangleright \Delta';\,\Gamma'\,[m]}$$

$$s{<}{>}_1\ \frac{\Delta;\,\Gamma \vdash \mathcal{C}\ \mathbf{at} \Uparrow \tau \triangleright \Delta';\,\Gamma'\,[m] \qquad \tau \sqcup \forall\cdot\,\square \equiv \forall\cdot\,\tau' \qquad \Delta \vdash \sigma\ \mathsf{wf}}{\Delta;\,\Gamma \vdash \mathcal{C}\langle\sigma\rangle_1\ \mathbf{at} \Uparrow [0 \mapsto \sigma]\,\tau' \triangleright \Delta';\,\Gamma'\,[m]}$$

$$s\pi_1\ \frac{\Delta;\,\Gamma \vdash \mathcal{C}\ \mathbf{at} \Uparrow \tau \triangleright \Delta';\,\Gamma'\,[m] \qquad \tau \sqcup \square \times \square \equiv \tau_1 \times \tau_2}{\Delta;\,\Gamma \vdash \pi_1\,\mathcal{C}\ \mathbf{at} \Uparrow \tau_1 \triangleright \Delta';\,\Gamma'\,[m]}$$

$$s\pi_2\ \frac{\Delta;\,\Gamma \vdash \mathcal{C}\ \mathbf{at} \Uparrow \tau \triangleright \Delta';\,\Gamma'\,[m] \qquad \tau \sqcup \square \times \square \equiv \tau_1 \times \tau_2}{\Delta;\,\Gamma \vdash \pi_2\,\mathcal{C}\ \mathbf{at} \Uparrow \tau_2 \triangleright \Delta';\,\Gamma'\,[m]}$$

$$s\Lambda\ \frac{\Delta, \alpha;\,\Gamma \vdash \mathcal{C}\ \mathbf{at} \Uparrow \tau \triangleright \Delta';\,\Gamma'\,[m]}{\Delta;\,\Gamma \vdash \Lambda\alpha.\,\mathcal{C}\ \mathbf{at} \Uparrow \forall\alpha.\,\tau \triangleright \Delta';\,\Gamma'\,[m]}$$

Figure 28: Unmarked context classification: application, type application, projections, type abstraction.

$$s\&_1 \frac{\Delta;\Gamma \vdash \mathcal{C}\ \mathbf{at}\ \Uparrow \tau_1 \triangleright \Delta';\Gamma'\,[m] \qquad \Delta;\Gamma \vdash e \Uparrow \tau_2}{\Delta;\Gamma \vdash \mathcal{C}\ \&_1\, e\ \mathbf{at}\ \Uparrow \tau_1 \times \tau_2 \triangleright \Delta';\Gamma'\,[m]}$$

$$s\&_2 \frac{\Delta;\Gamma \vdash e \Uparrow \tau_1 \qquad \Delta;\Gamma \vdash \mathcal{C}\ \mathbf{at}\ \Uparrow \tau_2 \triangleright \Delta';\Gamma'\,[m]}{\Delta;\Gamma \vdash e\ \&_2\, \mathcal{C}\ \mathbf{at}\ \Uparrow \tau_1 \times \tau_2 \triangleright \Delta';\Gamma'\,[m]}$$

$$a\&_1 \frac{\tau \sqcup \Box \times \Box \equiv \tau_1 \times \tau_2 \qquad \Delta;\Gamma \vdash \mathcal{C}\ \mathbf{at}\ \Downarrow \tau_1 \triangleright \Delta';\Gamma'\,[m] \qquad \Delta;\Gamma \vdash e \Downarrow \tau_2}{\Delta;\Gamma \vdash \mathcal{C}\ \&_1\, e\ \mathbf{at}\ \Downarrow \tau \triangleright \Delta';\Gamma'\,[m]}$$

$$a\&_2 \frac{\tau \sqcup \Box \times \Box \equiv \tau_1 \times \tau_2 \qquad \Delta;\Gamma \vdash e \Downarrow \tau_1 \qquad \Delta;\Gamma \vdash \mathcal{C}\ \mathbf{at}\ \Downarrow \tau_2 \triangleright \Delta';\Gamma'\,[m]}{\Delta;\Gamma \vdash e\ \&_2\, \mathcal{C}\ \mathbf{at}\ \Downarrow \tau \triangleright \Delta';\Gamma'\,[m]}$$

$$s\iota_1 \frac{\Delta;\Gamma \vdash \mathcal{C}\ \mathbf{at}\ \Uparrow \tau \triangleright \Delta';\Gamma'\,[m]}{\Delta;\Gamma \vdash \iota_1\, \mathcal{C}\ \mathbf{at}\ \Uparrow \tau + \Box \triangleright \Delta';\Gamma'\,[m]}$$

$$s\iota_2 \frac{\Delta;\Gamma \vdash \mathcal{C}\ \mathbf{at}\ \Uparrow \tau \triangleright \Delta';\Gamma'\,[m]}{\Delta;\Gamma \vdash \iota_2\, \mathcal{C}\ \mathbf{at}\ \Uparrow \Box + \tau \triangleright \Delta';\Gamma'\,[m]}$$

$$a\iota_1 \frac{\tau \sqcup \Box + \Box \equiv \tau_1 + \tau_2 \qquad \Delta;\Gamma \vdash \mathcal{C}\ \mathbf{at}\ \Downarrow \tau_1 \triangleright \Delta';\Gamma'\,[m]}{\Delta;\Gamma \vdash \iota_1\, \mathcal{C}\ \mathbf{at}\ \Downarrow \tau \triangleright \Delta';\Gamma'\,[m]}$$

$$a\iota_2 \frac{\tau \sqcup \Box + \Box \equiv \tau_1 + \tau_2 \qquad \Delta;\Gamma \vdash \mathcal{C}\ \mathbf{at}\ \Downarrow \tau_2 \triangleright \Delta';\Gamma'\,[m]}{\Delta;\Gamma \vdash \iota_2\, \mathcal{C}\ \mathbf{at}\ \Downarrow \tau \triangleright \Delta';\Gamma'\,[m]}$$

Figure 29: Unmarked context classification: pairs and injections.

$$scase_1 \frac{\begin{array}{c}\Delta;\Gamma \vdash e \Uparrow \tau \qquad \tau \sqcup \Box + \Box \equiv \tau_1 + \tau_2 \\ \Delta;\Gamma, x{:}\tau_1 \vdash \mathcal{C}\ \mathbf{at}\ \Uparrow \tau_1' \triangleright \Delta';\Gamma'\,[m] \qquad \Delta;\Gamma, y{:}\tau_2 \vdash e' \Uparrow \tau_2' \qquad \tau_1' \sim \tau_2'\end{array}}{\Delta;\Gamma \vdash \mathbf{case}\, e\, \mathbf{of}\, x.\mathcal{C} \mid y.e'\ \mathbf{at}\ \Uparrow \tau_1' \sqcup \tau_2' \triangleright \Delta';\Gamma'\,[m]}$$

$$scase_2 \frac{\begin{array}{c}\Delta;\Gamma \vdash e \Uparrow \tau \qquad \tau \sqcup \Box + \Box \equiv \tau_1 + \tau_2 \\ \Delta;\Gamma, x{:}\tau_1 \vdash e' \Uparrow \tau_1' \qquad \Delta;\Gamma, y{:}\tau_2 \vdash \mathcal{C}\ \mathbf{at}\ \Uparrow \tau_2' \triangleright \Delta';\Gamma'\,[m] \qquad \tau_1' \sim \tau_2'\end{array}}{\Delta;\Gamma \vdash \mathbf{case}\, e\, \mathbf{of}\, x.e' \mid y.\mathcal{C}\ \mathbf{at}\ \Uparrow \tau_1' \sqcup \tau_2' \triangleright \Delta';\Gamma'\,[m]}$$

$$acase_1 \frac{\begin{array}{c}\Delta;\Gamma \vdash e \Uparrow \tau_0 \qquad \tau_0 \sqcup \Box + \Box \equiv \tau_1 + \tau_2 \\ \Delta;\Gamma, x{:}\tau_1 \vdash \mathcal{C}\ \mathbf{at}\ \Downarrow \tau \triangleright \Delta';\Gamma'\,[m] \qquad \Delta;\Gamma, y{:}\tau_2 \vdash e' \Downarrow \tau\end{array}}{\Delta;\Gamma \vdash \mathbf{case}\, e\, \mathbf{of}\, x.\mathcal{C} \mid y.e'\ \mathbf{at}\ \Downarrow \tau \triangleright \Delta';\Gamma'\,[m]}$$

$$acase_2 \frac{\begin{array}{c}\Delta;\Gamma \vdash e \Uparrow \tau_0 \qquad \tau_0 \sqcup \Box + \Box \equiv \tau_1 + \tau_2 \\ \Delta;\Gamma, x{:}\tau_1 \vdash e' \Downarrow \tau \qquad \Delta;\Gamma, y{:}\tau_2 \vdash \mathcal{C}\ \mathbf{at}\ \Downarrow \tau \triangleright \Delta';\Gamma'\,[m]\end{array}}{\Delta;\Gamma \vdash \mathbf{case}\, e\, \mathbf{of}\, x.e' \mid y.\mathcal{C}\ \mathbf{at}\ \Downarrow \tau \triangleright \Delta';\Gamma'\,[m]}$$

Figure 30: Unmarked context classification: case.

$$sdef_1 \frac{\Delta;\Gamma \vdash \mathcal{C}\ \mathbf{at}\ \Uparrow \tau' \triangleright \Delta';\Gamma'\,[m] \qquad \Delta;\Gamma, x{:}\tau' \vdash e \Uparrow \tau}{\Delta;\Gamma \vdash \mathbf{def}\, \mathcal{C} \vdash_1 e\ \mathbf{at}\ \Uparrow \tau \triangleright \Delta';\Gamma'\,[m]}$$

$$sdef_2 \frac{\Delta;\Gamma \vdash e \Uparrow \tau' \qquad \Delta;\Gamma, x{:}\tau' \vdash \mathcal{C}\ \mathbf{at}\ \Uparrow \tau \triangleright \Delta';\Gamma'\,[m]}{\Delta;\Gamma \vdash \mathbf{def}\, e \vdash_2 \mathcal{C}\ \mathbf{at}\ \Uparrow \tau \triangleright \Delta';\Gamma'\,[m]}$$

$$adef_1 \frac{\Delta;\Gamma \vdash \mathcal{C}\ \mathbf{at}\ \Uparrow \tau' \triangleright \Delta';\Gamma'\,[m] \qquad \Delta;\Gamma, x{:}\tau' \vdash e \Downarrow \tau}{\Delta;\Gamma \vdash \mathbf{def}\, \mathcal{C} \vdash_1 e\ \mathbf{at}\ \Downarrow \tau \triangleright \Delta';\Gamma'\,[m]}$$

$$adef_2 \frac{\Delta;\Gamma \vdash e \Uparrow \tau' \qquad \Delta;\Gamma, x{:}\tau' \vdash \mathcal{C}\ \mathbf{at}\ \Downarrow \tau \triangleright \Delta';\Gamma'\,[m]}{\Delta;\Gamma \vdash \mathbf{def}\, e \vdash_2 \mathcal{C}\ \mathbf{at}\ \Downarrow \tau \triangleright \Delta';\Gamma'\,[m]}$$

Figure 31: Unmarked context classification: let (def).

$\boxed{\Delta;\,\Gamma \vdash C \looparrowright \check{C} \textbf{ at } p \triangleright \Delta';\,\Gamma'\,[m]}$ Marked context $C \looparrowright \check{C}$ at outer position p classifies its focus under $\Delta';\,\Gamma'$ in focus mode m

$$ms\bigcirc\frac{}{\Delta;\,\Gamma \vdash \bigcirc \looparrowright \bigcirc \textbf{ at } \Uparrow\tau \triangleright \Delta;\,\Gamma\,[\Uparrow\tau]} \qquad ma\bigcirc\frac{}{\Delta;\,\Gamma \vdash \bigcirc \looparrowright \bigcirc \textbf{ at } \Downarrow\tau \triangleright \Delta;\,\Gamma\,[\Downarrow\tau]}$$

$$maSub\frac{\Delta;\,\Gamma \vdash C \looparrowright \check{C} \textbf{ at } \Uparrow\tau' \triangleright \Delta';\,\Gamma'\,[m] \qquad \tau \sim \tau'}{\Delta;\,\Gamma \vdash C \looparrowright \check{C} \textbf{ at } \Downarrow\tau \triangleright \Delta';\,\Gamma'\,[m]}$$

$$maSub{\Uparrow}\frac{\Delta;\,\Gamma \vdash C \looparrowright \check{C} \textbf{ at } \Uparrow\tau' \triangleright \Delta';\,\Gamma'\,[m] \qquad \neg(\tau \sim \tau')}{\Delta;\,\Gamma \vdash C \looparrowright (\!|\check{C}|\!)_{\tau' \not\sim \tau} \textbf{ at } \Downarrow\tau \triangleright \Delta';\,\Gamma'\,[m]}$$

$$ms\lambda:\frac{\Delta \vdash \tau_1\ \mathsf{wf} \qquad \Delta;\,\Gamma, x{:}\tau_1 \vdash C \looparrowright \check{C} \textbf{ at } \Uparrow\tau_2 \triangleright \Delta';\,\Gamma'\,[m]}{\Delta;\,\Gamma \vdash \lambda x{:}\tau_1.\,C \looparrowright \lambda x{:}\tau_1.\,\check{C} \textbf{ at } \Uparrow\tau_1 \Rightarrow \tau_2 \triangleright \Delta';\,\Gamma'\,[m]}$$

$$ma\lambda:\frac{\begin{array}{c}\tau \sim \tau_1 \Rightarrow \square \qquad\qquad \tau \sqcup \tau_1 \Rightarrow \square \equiv \tau_1 \Rightarrow \tau_2 \\ \Delta \vdash \tau_1\ \mathsf{wf} \qquad \Delta;\,\Gamma, x{:}\tau_1 \vdash C \looparrowright \check{C} \textbf{ at } \Downarrow\tau_2 \triangleright \Delta';\,\Gamma'\,[m]\end{array}}{\Delta;\,\Gamma \vdash \lambda x{:}\tau_1.\,C \looparrowright \lambda x{:}\tau_1.\,\check{C} \textbf{ at } \Downarrow\tau \triangleright \Delta';\,\Gamma'\,[m]}$$

Figure 32: Marked context classification: holes, subsumption, annotated lambda.

$$ma\lambda{\Rightarrow}\frac{\tau \sqcup \square \Rightarrow \square \equiv \tau_1 \Rightarrow \tau_2 \qquad \Delta;\,\Gamma, x{:}\tau_1 \vdash C \looparrowright \check{C} \textbf{ at } \Downarrow\tau_2 \triangleright \Delta';\,\Gamma'\,[m]}{\Delta;\,\Gamma \vdash \lambda x.\,C \looparrowright \lambda x.\,\check{C} \textbf{ at } \Downarrow\tau \triangleright \Delta';\,\Gamma'\,[m]}$$

$$ma\lambda{\Rightarrow}{\Uparrow}\frac{\forall\tau_1,\tau_2.\,\tau \sqcup \square \Rightarrow \square \not\equiv \tau_1 \Rightarrow \tau_2 \qquad \Delta;\,\Gamma \vdash C \looparrowright \check{C} \textbf{ at } \Downarrow\square \triangleright \Delta';\,\Gamma'\,[m]}{\Delta;\,\Gamma \vdash \lambda x.\,C \looparrowright (\!|\lambda x.\,\check{C}|\!)_{\sim\Rightarrow} \textbf{ at } \Downarrow\tau \triangleright \Delta';\,\Gamma'\,[m]}$$

$$ms\lambda{\Rightarrow}{\Uparrow}\frac{\Delta;\,\Gamma \vdash C \looparrowright \check{C} \textbf{ at } \Downarrow\square \triangleright \Delta';\,\Gamma'\,[m]}{\Delta;\,\Gamma \vdash \lambda x.\,C \looparrowright (\!|\lambda x.\,\check{C}|\!)_{\sim\Rightarrow} \textbf{ at } \Uparrow\square \triangleright \Delta';\,\Gamma'\,[m]}$$

$$ms\circ_1\frac{\Delta;\,\Gamma \vdash C \looparrowright \check{C} \textbf{ at } \Uparrow\tau \triangleright \Delta';\,\Gamma'\,[m] \qquad \tau \sqcup \square \Rightarrow \square \equiv \tau_1 \Rightarrow \tau_2 \qquad \Delta;\,\Gamma \vdash e \looparrowright \check{e} \Downarrow \tau_1}{\Delta;\,\Gamma \vdash C \circ_1 e \looparrowright \check{C} \circ_1 \check{e} \textbf{ at } \Uparrow\tau_2 \triangleright \Delta';\,\Gamma'\,[m]}$$

$$ms\circ_1{\Uparrow}\frac{\Delta;\,\Gamma \vdash C \looparrowright \check{C} \textbf{ at } \Uparrow\tau \triangleright \Delta';\,\Gamma'\,[m] \qquad \forall\tau_1,\tau_2.\,\tau \sqcup \square \Rightarrow \square \not\equiv \tau_1 \Rightarrow \tau_2 \qquad \Delta;\,\Gamma \vdash e \looparrowright \check{e} \Downarrow \square}{\Delta;\,\Gamma \vdash C \circ_1 e \looparrowright (\!|\check{C}|\!)_{\blacktriangleright\Rightarrow} \circ_1 \check{e} \textbf{ at } \Uparrow\square \triangleright \Delta';\,\Gamma'\,[m]}$$

$$ms\circ_2\frac{\Delta;\,\Gamma \vdash e \looparrowright \check{e} \Uparrow \tau \qquad \tau \sqcup \square \Rightarrow \square \equiv \tau_1 \Rightarrow \tau_2 \qquad \Delta;\,\Gamma \vdash C \looparrowright \check{C} \textbf{ at } \Downarrow\tau_1 \triangleright \Delta';\,\Gamma'\,[m]}{\Delta;\,\Gamma \vdash e \circ_2 C \looparrowright \check{e} \circ_2 \check{C} \textbf{ at } \Uparrow\tau_2 \triangleright \Delta';\,\Gamma'\,[m]}$$

$$ms\circ_2{\Uparrow}\frac{\Delta;\,\Gamma \vdash e \looparrowright \check{e} \Uparrow \tau \qquad \forall\tau_1,\tau_2.\,\tau \sqcup \square \Rightarrow \square \not\equiv \tau_1 \Rightarrow \tau_2 \qquad \Delta;\,\Gamma \vdash C \looparrowright \check{C} \textbf{ at } \Downarrow\square \triangleright \Delta';\,\Gamma'\,[m]}{\Delta;\,\Gamma \vdash e \circ_2 C \looparrowright (\!|\check{e}|\!)_{\blacktriangleright\Rightarrow} \circ_2 \check{C} \textbf{ at } \Uparrow\square \triangleright \Delta';\,\Gamma'\,[m]}$$

Figure 33: Marked context classification: unannotated lambda and application.

$$ms{<>}_1\ \frac{\Delta;\Gamma \vdash C \leadsto \check{C}\ \mathbf{at}\ \Uparrow\tau\ \triangleright\ \Delta';\Gamma'\ [m] \qquad \tau \sqcup \forall\cdot\ \square \equiv \forall\cdot\ \tau' \qquad \Delta \vdash \sigma\ \mathsf{wf}}{\Delta;\Gamma \vdash C\langle\sigma\rangle_1 \leadsto \check{C}\langle\sigma\rangle_1\ \mathbf{at}\ \Uparrow[0 \mapsto \sigma]\,\tau'\ \triangleright\ \Delta';\Gamma'\ [m]}$$

$$ms{<>}_1{\Uparrow}\ \frac{\Delta;\Gamma \vdash C \leadsto \check{C}\ \mathbf{at}\ \Uparrow\tau\ \triangleright\ \Delta';\Gamma'\ [m] \qquad \forall\tau'.\ \tau \sqcup \forall\cdot\ \square \not\equiv \forall\cdot\ \tau' \qquad \Delta \vdash \sigma\ \mathsf{wf}}{\Delta;\Gamma \vdash C\langle\sigma\rangle_1 \leadsto (\!|\check{C}|\!)_{\blacktriangleright\forall}\langle\sigma\rangle_1\ \mathbf{at}\ \Uparrow\square\ \triangleright\ \Delta';\Gamma'\ [m]}$$

$$ms\&_1\ \frac{\Delta;\Gamma \vdash C \leadsto \check{C}\ \mathbf{at}\ \Uparrow\tau_1\ \triangleright\ \Delta';\Gamma'\ [m] \qquad \Delta;\Gamma \vdash e \leadsto \check{e}\ \Uparrow\tau_2}{\Delta;\Gamma \vdash C\ \&_1\ e \leadsto \check{C}\ \&_1\ \check{e}\ \mathbf{at}\ \Uparrow\tau_1 \times \tau_2\ \triangleright\ \Delta';\Gamma'\ [m]}$$

$$ms\&_2\ \frac{\Delta;\Gamma \vdash e \leadsto \check{e}\ \Uparrow\tau_1 \qquad \Delta;\Gamma \vdash C \leadsto \check{C}\ \mathbf{at}\ \Uparrow\tau_2\ \triangleright\ \Delta';\Gamma'\ [m]}{\Delta;\Gamma \vdash e\ \&_2\ C \leadsto \check{e}\ \&_2\ \check{C}\ \mathbf{at}\ \Uparrow\tau_1 \times \tau_2\ \triangleright\ \Delta';\Gamma'\ [m]}$$

$$ma\&_1\ \frac{\tau \sqcup \square \times \square \equiv \tau_1 \times \tau_2 \qquad \Delta;\Gamma \vdash C \leadsto \check{C}\ \mathbf{at}\ \Downarrow\tau_1\ \triangleright\ \Delta';\Gamma'\ [m] \qquad \Delta;\Gamma \vdash e \leadsto \check{e}\ \Downarrow\tau_2}{\Delta;\Gamma \vdash C\ \&_1\ e \leadsto \check{C}\ \&_1\ \check{e}\ \mathbf{at}\ \Downarrow\tau\ \triangleright\ \Delta';\Gamma'\ [m]}$$

$$ma\&_2\ \frac{\tau \sqcup \square \times \square \equiv \tau_1 \times \tau_2 \qquad \Delta;\Gamma \vdash e \leadsto \check{e}\ \Downarrow\tau_1 \qquad \Delta;\Gamma \vdash C \leadsto \check{C}\ \mathbf{at}\ \Downarrow\tau_2\ \triangleright\ \Delta';\Gamma'\ [m]}{\Delta;\Gamma \vdash e\ \&_2\ C \leadsto \check{e}\ \&_2\ \check{C}\ \mathbf{at}\ \Downarrow\tau\ \triangleright\ \Delta';\Gamma'\ [m]}$$

Figure 34: Marked context classification: type application and pairs.

$$ms\iota_1\ \frac{\Delta;\Gamma \vdash C \leadsto \check{C}\ \mathbf{at}\ \Uparrow\tau\ \triangleright\ \Delta';\Gamma'\ [m]}{\Delta;\Gamma \vdash \iota_1\,C \leadsto \iota_1\,\check{C}\ \mathbf{at}\ \Uparrow\tau + \square\ \triangleright\ \Delta';\Gamma'\ [m]}$$

$$ms\iota_2\ \frac{\Delta;\Gamma \vdash C \leadsto \check{C}\ \mathbf{at}\ \Uparrow\tau\ \triangleright\ \Delta';\Gamma'\ [m]}{\Delta;\Gamma \vdash \iota_2\,C \leadsto \iota_2\,\check{C}\ \mathbf{at}\ \Uparrow\square + \tau\ \triangleright\ \Delta';\Gamma'\ [m]}$$

$$ma\iota_1\ \frac{\tau \sqcup \square + \square \equiv \tau_1 + \tau_2 \qquad \Delta;\Gamma \vdash C \leadsto \check{C}\ \mathbf{at}\ \Downarrow\tau_1\ \triangleright\ \Delta';\Gamma'\ [m]}{\Delta;\Gamma \vdash \iota_1\,C \leadsto \iota_1\,\check{C}\ \mathbf{at}\ \Downarrow\tau\ \triangleright\ \Delta';\Gamma'\ [m]}$$

$$ma\iota_2\ \frac{\tau \sqcup \square + \square \equiv \tau_1 + \tau_2 \qquad \Delta;\Gamma \vdash C \leadsto \check{C}\ \mathbf{at}\ \Downarrow\tau_2\ \triangleright\ \Delta';\Gamma'\ [m]}{\Delta;\Gamma \vdash \iota_2\,C \leadsto \iota_2\,\check{C}\ \mathbf{at}\ \Downarrow\tau\ \triangleright\ \Delta';\Gamma'\ [m]}$$

$$ms\pi_1\ \frac{\Delta;\Gamma \vdash C \leadsto \check{C}\ \mathbf{at}\ \Uparrow\tau\ \triangleright\ \Delta';\Gamma'\ [m] \qquad \tau \sqcup \square \times \square \equiv \tau_1 \times \tau_2}{\Delta;\Gamma \vdash \pi_1\,C \leadsto \pi_1\,\check{C}\ \mathbf{at}\ \Uparrow\tau_1\ \triangleright\ \Delta';\Gamma'\ [m]}$$

$$ms\pi_2\ \frac{\Delta;\Gamma \vdash C \leadsto \check{C}\ \mathbf{at}\ \Uparrow\tau\ \triangleright\ \Delta';\Gamma'\ [m] \qquad \tau \sqcup \square \times \square \equiv \tau_1 \times \tau_2}{\Delta;\Gamma \vdash \pi_2\,C \leadsto \pi_2\,\check{C}\ \mathbf{at}\ \Uparrow\tau_2\ \triangleright\ \Delta';\Gamma'\ [m]}$$

$$ms\pi_1{\Uparrow}\ \frac{\Delta;\Gamma \vdash C \leadsto \check{C}\ \mathbf{at}\ \Uparrow\tau\ \triangleright\ \Delta';\Gamma'\ [m] \qquad \forall\tau_1,\tau_2.\ \tau \sqcup \square \times \square \not\equiv \tau_1 \times \tau_2}{\Delta;\Gamma \vdash \pi_1\,C \leadsto \pi_1\,(\!|\check{C}|\!)_{\blacktriangleright\times}\ \mathbf{at}\ \Uparrow\square\ \triangleright\ \Delta';\Gamma'\ [m]}$$

$$ms\pi_2{\Uparrow}\ \frac{\Delta;\Gamma \vdash C \leadsto \check{C}\ \mathbf{at}\ \Uparrow\tau\ \triangleright\ \Delta';\Gamma'\ [m] \qquad \forall\tau_1,\tau_2.\ \tau \sqcup \square \times \square \not\equiv \tau_1 \times \tau_2}{\Delta;\Gamma \vdash \pi_2\,C \leadsto \pi_2\,(\!|\check{C}|\!)_{\blacktriangleright\times}\ \mathbf{at}\ \Uparrow\square\ \triangleright\ \Delta';\Gamma'\ [m]}$$

Figure 35: Marked context classification: injections and projections.

$$mscase_1\ \frac{\Delta;\Gamma \vdash e \rightsquigarrow \check{e} \Uparrow \tau \qquad \tau \sqcup \square + \square \equiv \tau_1 + \tau_2 \\ \Delta;\Gamma, x{:}\tau_1 \vdash C \rightsquigarrow \check{C}\ \mathbf{at}\ \Uparrow \tau_1' \triangleright \Delta';\Gamma'\,[m] \qquad \Delta;\Gamma, y{:}\tau_2 \vdash e' \rightsquigarrow \check{e}' \Uparrow \tau_2' \qquad \tau_1' \sim \tau_2'}{\Delta;\Gamma \vdash \mathbf{case}\, e\, \mathbf{of}\, x.\, C \mid y.\, e' \rightsquigarrow \mathbf{case}\, \check{e}\, \mathbf{of}\, x.\, \check{C} \mid y.\, \check{e}'\ \mathbf{at}\ \Uparrow \tau_1' \sqcup \tau_2' \triangleright \Delta';\Gamma'\,[m]}$$

$$mscase_1{\Uparrow}\ \frac{\Delta;\Gamma \vdash e \rightsquigarrow \check{e} \Uparrow \tau \qquad \forall \tau_1, \tau_2.\ \tau \sqcup \square + \square \not\equiv \tau_1 + \tau_2 \\ \Delta;\Gamma \vdash C \rightsquigarrow \check{C}\ \mathbf{at}\ \Uparrow \tau_1' \triangleright \Delta';\Gamma'\,[m] \qquad \Delta;\Gamma \vdash e' \rightsquigarrow \check{e}' \Uparrow \tau_2'}{\Delta;\Gamma \vdash \mathbf{case}\, e\, \mathbf{of}\, x.\, C \mid y.\, e' \rightsquigarrow \mathbf{case}\, (\!|\check{e}|\!)_{\blacktriangleright +}\, \mathbf{of}\, x.\, \check{C} \mid y.\, \check{e}'\ \mathbf{at}\ \Uparrow \square \triangleright \Delta';\Gamma'\,[m]}$$

$$mscase_1{\not\sim}\ \frac{\Delta;\Gamma \vdash e \rightsquigarrow \check{e} \Uparrow \tau \qquad \tau \sqcup \square + \square \equiv \tau_1 + \tau_2 \\ \Delta;\Gamma, x{:}\tau_1 \vdash C \rightsquigarrow \check{C}\ \mathbf{at}\ \Uparrow \tau_1' \triangleright \Delta';\Gamma'\,[m] \qquad \Delta;\Gamma, y{:}\tau_2 \vdash e' \rightsquigarrow \check{e}' \Uparrow \tau_2' \qquad \neg(\tau_1' \sim \tau_2')}{\Delta;\Gamma \vdash \mathbf{case}\, e\, \mathbf{of}\, x.\, C \mid y.\, e' \rightsquigarrow \mathbf{case}\, \check{e}\, \mathbf{of}\, x.\, \check{C} \mid y.\, \check{e}'\ \mathbf{at}\ \Uparrow \square \triangleright \Delta';\Gamma'\,[m]}$$

$$mscase_2\ \frac{\Delta;\Gamma \vdash e \rightsquigarrow \check{e} \Uparrow \tau \qquad \tau \sqcup \square + \square \equiv \tau_1 + \tau_2 \\ \Delta;\Gamma, x{:}\tau_1 \vdash e' \rightsquigarrow \check{e}' \Uparrow \tau_1' \qquad \Delta;\Gamma, y{:}\tau_2 \vdash C \rightsquigarrow \check{C}\ \mathbf{at}\ \Uparrow \tau_2' \triangleright \Delta';\Gamma'\,[m] \qquad \tau_1' \sim \tau_2'}{\Delta;\Gamma \vdash \mathbf{case}\, e\, \mathbf{of}\, x.\, e' \mid y.\, C \rightsquigarrow \mathbf{case}\, \check{e}\, \mathbf{of}\, x.\, \check{e}' \mid y.\, \check{C}\ \mathbf{at}\ \Uparrow \tau_1' \sqcup \tau_2' \triangleright \Delta';\Gamma'\,[m]}$$

$$mscase_2{\Uparrow}\ \frac{\Delta;\Gamma \vdash e \rightsquigarrow \check{e} \Uparrow \tau \qquad \forall \tau_1, \tau_2.\ \tau \sqcup \square + \square \not\equiv \tau_1 + \tau_2 \\ \Delta;\Gamma \vdash e' \rightsquigarrow \check{e}' \Uparrow \tau_1' \qquad \Delta;\Gamma \vdash C \rightsquigarrow \check{C}\ \mathbf{at}\ \Uparrow \tau_2' \triangleright \Delta';\Gamma'\,[m]}{\Delta;\Gamma \vdash \mathbf{case}\, e\, \mathbf{of}\, x.\, e' \mid y.\, C \rightsquigarrow \mathbf{case}\, (\!|\check{e}|\!)_{\blacktriangleright +}\, \mathbf{of}\, x.\, \check{e}' \mid y.\, \check{C}\ \mathbf{at}\ \Uparrow \square \triangleright \Delta';\Gamma'\,[m]}$$

$$mscase_2{\not\sim}\ \frac{\Delta;\Gamma \vdash e \rightsquigarrow \check{e} \Uparrow \tau \qquad \tau \sqcup \square + \square \equiv \tau_1 + \tau_2 \\ \Delta;\Gamma, x{:}\tau_1 \vdash e' \rightsquigarrow \check{e}' \Uparrow \tau_1' \qquad \Delta;\Gamma, y{:}\tau_2 \vdash C \rightsquigarrow \check{C}\ \mathbf{at}\ \Uparrow \tau_2' \triangleright \Delta';\Gamma'\,[m] \qquad \neg(\tau_1' \sim \tau_2')}{\Delta;\Gamma \vdash \mathbf{case}\, e\, \mathbf{of}\, x.\, e' \mid y.\, C \rightsquigarrow \mathbf{case}\, \check{e}\, \mathbf{of}\, x.\, \check{e}' \mid y.\, \check{C}\ \mathbf{at}\ \Uparrow \square \triangleright \Delta';\Gamma'\,[m]}$$

Figure 36: Marked context classification: synthesis-mode case.

$$macase_1\ \frac{\Delta;\Gamma \vdash e \rightsquigarrow \check{e} \Uparrow \tau_0 \qquad \tau_0 \sqcup \square + \square \equiv \tau_1 + \tau_2 \\ \Delta;\Gamma, x{:}\tau_1 \vdash C \rightsquigarrow \check{C}\ \mathbf{at}\ \Downarrow \tau \triangleright \Delta';\Gamma'\,[m] \qquad \Delta;\Gamma, y{:}\tau_2 \vdash e' \rightsquigarrow \check{e}' \Downarrow \tau}{\Delta;\Gamma \vdash \mathbf{case}\, e\, \mathbf{of}\, x.\, C \mid y.\, e' \rightsquigarrow \mathbf{case}\, \check{e}\, \mathbf{of}\, x.\, \check{C} \mid y.\, \check{e}'\ \mathbf{at}\ \Downarrow \tau \triangleright \Delta';\Gamma'\,[m]}$$

$$macase_1{\Uparrow}\ \frac{\Delta;\Gamma \vdash e \rightsquigarrow \check{e} \Uparrow \tau_0 \qquad \forall \tau_1, \tau_2.\ \tau_0 \sqcup \square + \square \not\equiv \tau_1 + \tau_2 \\ \Delta;\Gamma, x{:}\square \vdash C \rightsquigarrow \check{C}\ \mathbf{at}\ \Downarrow \tau \triangleright \Delta';\Gamma'\,[m] \qquad \Delta;\Gamma, y{:}\square \vdash e' \rightsquigarrow \check{e}' \Downarrow \tau}{\Delta;\Gamma \vdash \mathbf{case}\, e\, \mathbf{of}\, x.\, C \mid y.\, e' \rightsquigarrow \mathbf{case}\, (\!|\check{e}|\!)_{\blacktriangleright +}\, \mathbf{of}\, x.\, \check{C} \mid y.\, \check{e}'\ \mathbf{at}\ \Downarrow \tau \triangleright \Delta';\Gamma'\,[m]}$$

$$macase_2\ \frac{\Delta;\Gamma \vdash e \rightsquigarrow \check{e} \Uparrow \tau_0 \qquad \tau_0 \sqcup \square + \square \equiv \tau_1 + \tau_2 \\ \Delta;\Gamma, x{:}\tau_1 \vdash e' \rightsquigarrow \check{e}' \Downarrow \tau \qquad \Delta;\Gamma, y{:}\tau_2 \vdash C \rightsquigarrow \check{C}\ \mathbf{at}\ \Downarrow \tau \triangleright \Delta';\Gamma'\,[m]}{\Delta;\Gamma \vdash \mathbf{case}\, e\, \mathbf{of}\, x.\, e' \mid y.\, C \rightsquigarrow \mathbf{case}\, \check{e}\, \mathbf{of}\, x.\, \check{e}' \mid y.\, \check{C}\ \mathbf{at}\ \Downarrow \tau \triangleright \Delta';\Gamma'\,[m]}$$

$$macase_2{\Uparrow}\ \frac{\Delta;\Gamma \vdash e \rightsquigarrow \check{e} \Uparrow \tau_0 \qquad \forall \tau_1, \tau_2.\ \tau_0 \sqcup \square + \square \not\equiv \tau_1 + \tau_2 \\ \Delta;\Gamma, x{:}\square \vdash e' \rightsquigarrow \check{e}' \Downarrow \tau \qquad \Delta;\Gamma, y{:}\square \vdash C \rightsquigarrow \check{C}\ \mathbf{at}\ \Downarrow \tau \triangleright \Delta';\Gamma'\,[m]}{\Delta;\Gamma \vdash \mathbf{case}\, e\, \mathbf{of}\, x.\, e' \mid y.\, C \rightsquigarrow \mathbf{case}\, (\!|\check{e}|\!)_{\blacktriangleright +}\, \mathbf{of}\, x.\, \check{e}' \mid y.\, \check{C}\ \mathbf{at}\ \Downarrow \tau \triangleright \Delta';\Gamma'\,[m]}$$

$$msdef_1\ \frac{\Delta;\Gamma \vdash C \rightsquigarrow \check{C}\ \mathbf{at}\ \Uparrow \tau' \triangleright \Delta';\Gamma'\,[m] \qquad \Delta;\Gamma, x{:}\tau' \vdash e \rightsquigarrow \check{e} \Uparrow \tau}{\Delta;\Gamma \vdash \mathbf{def}\, C \vdash_1 e \rightsquigarrow \mathbf{def}\, \check{C} \vdash_1 \check{e}\ \mathbf{at}\ \Uparrow \tau \triangleright \Delta';\Gamma'\,[m]}$$

$$msdef_2\ \frac{\Delta;\Gamma \vdash e \rightsquigarrow \check{e} \Uparrow \tau' \qquad \Delta;\Gamma, x{:}\tau' \vdash C \rightsquigarrow \check{C}\ \mathbf{at}\ \Uparrow \tau \triangleright \Delta';\Gamma'\,[m]}{\Delta;\Gamma \vdash \mathbf{def}\, e \vdash_2 C \rightsquigarrow \mathbf{def}\, \check{e} \vdash_2 \check{C}\ \mathbf{at}\ \Uparrow \tau \triangleright \Delta';\Gamma'\,[m]}$$

Figure 37: Marked context classification: analysis-mode case and let (synthesis).

$$madef_1 \frac{\Delta;\, \Gamma \vdash C \looparrowright \check{C} \;\mathbf{at}\; \Uparrow \tau' \;\triangleright\; \Delta';\, \Gamma'\,[m] \qquad \Delta;\, \Gamma, x{:}\tau' \vdash e \looparrowright \check{e} \Downarrow \tau}{\Delta;\, \Gamma \vdash \mathbf{def}\, C \vdash_1 e \looparrowright \mathbf{def}\, \check{C} \vdash_1 \check{e} \;\mathbf{at}\; \Downarrow \tau \;\triangleright\; \Delta';\, \Gamma'\,[m]}$$

$$madef_2 \frac{\Delta;\, \Gamma \vdash e \looparrowright \check{e} \Uparrow \tau' \qquad \Delta;\, \Gamma, x{:}\tau' \vdash C \looparrowright \check{C} \;\mathbf{at}\; \Downarrow \tau \;\triangleright\; \Delta';\, \Gamma'\,[m]}{\Delta;\, \Gamma \vdash \mathbf{def}\, e \vdash_2 C \looparrowright \mathbf{def}\, \check{e} \vdash_2 \check{C} \;\mathbf{at}\; \Downarrow \tau \;\triangleright\; \Delta';\, \Gamma'\,[m]}$$

$$ms\Lambda \frac{\Delta, \alpha;\, \Gamma \vdash C \looparrowright \check{C} \;\mathbf{at}\; \Uparrow \tau \;\triangleright\; \Delta';\, \Gamma'\,[m]}{\Delta;\, \Gamma \vdash \Lambda\alpha.\, C \looparrowright \Lambda\alpha.\, \check{C} \;\mathbf{at}\; \Uparrow \forall\alpha.\, \tau \;\triangleright\; \Delta';\, \Gamma'\,[m]}$$

Figure 38: Marked context classification: let (analysis) and type abstraction.