Omnidirectional type inference for ML: principality any way

ALISTAIR O'BRIEN, University of Cambridge, United Kingdom
DIDIER RÉMY, INRIA, France
GABRIEL SCHERER, INRIA & IRIF, Université Paris Cité, France

The Damas-Hindley-Milner (ML) type system owes its success to *principality*, the property that every well-typed expression has a unique most general type. This makes inference predictable and efficient. Yet, principality is *fragile*: many extensions of ML—GADTs, higher-rank polymorphism, and static overloading—break it by introducing *fragile* constructs that resist principal inference. Existing approaches recover principality through *directional* inference algorithms, which propagate *known* type information in a fixed (or *static*) order (*e.g.* as in bidirectional typing) to disambiguate such constructs. However, the rigidity of a *static* inference order often causes otherwise well-typed programs to be rejected.

We propose *omnidirectional* type inference, where type information flows in a *dynamic* order. Typing constraints may be solved in any order, suspending when progress requires known type information and resuming once it becomes available, using *suspended match constraints*. This approach is straightforward for simply typed systems, but extending it to ML is challenging due to *let-generalization*. Existing ML inference algorithms type let-bindings in a fixed order—type the let-bound expression first, generalize its type, and then type the let-body. To overcome this, we introduce *incremental instantiation*, allowing partially solved type schemes containing suspended constraints to be instantiated, with a mechanism to incrementally update instances as the scheme is refined. Omnidirectionality provides a *general framework* for restoring principality in the presence of fragile features. We demonstrate its versatility on two fundamentally different features of OCaml: static overloading of record labels and datatype constructors and semi-explicit first-class polymorphism. In both cases, we obtain a *principal* type inference algorithm that is more expressive than OCaml's current typechecker.

1 Introduction

The Damas-Hindley-Milner (ML) [Damas and Milner 1982; Hindley 1969] type system has long occupied a sweet spot in the design space of strongly typed programming languages, as it enjoys the *principal types property*: every well-typed expression e has a most general type σ from which all other valid types for e are instances of σ. For example, the identity function $\lambda x.\, x$ has the principal type $\forall \alpha.\, \alpha \to \alpha$, generalizing types like $\mathsf{int} \to \mathsf{int}$ and $\mathsf{bool} \to \mathsf{bool}$.

The existence of principal types in ML has important practical benefits. It makes inference predictable, compositional, and efficient: since every expression has a most general type, local typing decisions are always optimal, with no need for guessing or backtracking. Beyond inference, principality ensures that well-typedness is stable under common program transformations such as let-contraction (or inlining) and argument reordering.

Principality, however, is fragile. Many extensions of ML—such as extensible records with row-polymorphism [Garrigue 1998; Ohori 1995; Rémy 1989; Rémy and Vouillon 1997; Wand 1989] and higher-kinded types [Jones 1995b]—are *robust*: they preserve principality. Others, including GADTs [Garrigue and Rémy 2013; Schrijvers, Jones, Sulzmann and Vytiniotis 2009], higher-rank and first-class polymorphism [Garrigue and Rémy 1999; Odersky and Läufer 1996; Serrano, Hage, Jones and Vytiniotis 2020], and static overloading [Charguéraud, Bodin, Dunfield and Riboulet 2025], are *fragile*: they break principality under their *natural* typing rules.

Authors' Contact Information: Alistair O'Brien, ajo41@cam.ac.uk, University of Cambridge, Cambridge, United Kingdom; Didier Rémy, didier.remy@inria.fr, Paris, INRIA, France; Gabriel Scherer, gabriel.scherer@inria.fr, INRIA & IRIF, Université Paris Cité, Paris, France.

This fragility can already be observed in OCaml through impredicative higher-rank (*i.e.*, first-class) polymorphism, exposed by *polymorphic methods* [Garrigue and Rémy 1999] (green (✓) indicates typechecking success and red (✗) indicates failure):

```
let self x = x#f x                                                    OCaml ✗
```

In OCaml, objects are defined as a collection of methods within **object** ... **end**, and accessed using $e\,\#\,m$. Unlike Java or C++, OCaml uses *structural typing* for objects: object types are a list of method types between two chevrons *e.g.* $\langle \texttt{f} : \alpha.\ \alpha \to \alpha \rangle$, where the method f has the polymorphic identity function type $\forall \alpha.\, \alpha \to \alpha$ (the $\forall$ being omitted in OCaml syntax). When typing self in the example above, one could *guess* the type of x to be either $\langle \texttt{f} : \alpha.\, \alpha \to \alpha \rangle$ or $\langle \texttt{f} : \alpha.\, \alpha \to \alpha \to \alpha \rangle$—neither of which is strictly more general than the other, violating principality.

Principality can be recovered through explicit type annotations. The return type of overloaded datatype constructors may be annotated; polymorphic expressions can be annotated with a type scheme; and for GADTs, both the type of the match scrutinee and return type can be annotated with a rigid type, which is refined by type equalities introduced in each branch. In this example, the binding of x should be annotated with the higher-rank type $\langle \texttt{f} : \alpha.\, \alpha \to \alpha \rangle$.

Every fragile construct has a corresponding *robust* form where the type annotation is mandatory—for instance, $(e\,\#\,m : \sigma)$ is the robust form of $e\,\#\,m$ for polymorphic method invocation. Robust forms preserve principality, but at the cost of being significantly more cumbersome to use. Fragile forms relieve this burden, but can only be elaborated into their robust counterpart if sufficient type information is already available from the context. Thus, typing fragile constructs ultimately amounts to identifying when type information is *known*.

Intuitively, by known information, we mean typing constraints that must hold—either from typing rules (*e.g.* application requires the function to have an arrow type) or programmer supplied type annotations. Yet, formulating a declarative specification for *when* type information is known is difficult: most specifications are often twisted with some direct or indirect algorithmic flavor in order to preserve principality and completeness.

The two dominant approaches thus far are *bidirectional* typechecking [Pierce and Turner 1998] and π*-directional* type inference [Garrigue and Rémy 1999]. Each impose some *static* ordering of inference, using it to propagate inferred types and user-provided annotations as *known* information.

While effective in many settings, the rigidity of a static ordering causes even simple examples whose type could easily be inferred to be rejected. For instance, OCaml accepts or rejects the following expression, depending on the position of the annotation:

```
let self₁₁ (x : ⟨f : α. α → α⟩) = if true then x#f x else x          OCaml ✓
let self₁₂ x = if true then x#f x else (x : ⟨f : α. α → α⟩)          OCaml ✗
```

We propose *omnidirectional* type inference, a global inference approach that relies on a *dynamic* order of inference. The solving of inference constraints may proceed in any order, suspending whenever progress requires *known* type information. Other constraints may continue to be solved; once the missing information becomes available (typically via unification), the suspended typing constraints are resumed.

We present omnidirectionality as a general framework for inference in the presence of fragile features. While this paper instantiates our framework for two concrete features in OCaml—static overloading of constructors and record fields, and polymorphic object methods—its scope is broader: we expect it to extend to richer features such as GADTs, polymorphic parameters [White 2023], and more general forms of static overloading *à la* Swift.

Despite the name, omnidirectionality is not an extension of bidirectional typechecking. Fragile features are often handled by *local* inference methods, most notably bidirectional typechecking, because they rely on the propagation of known type information; omnidirectionality addresses this same need using *global* inference.

Because our long-term goal is to integrate omnidirectional inference into OCaml's typechecker, it must scale to ML-style polymorphism. While the idea of suspending constraints is not new (see §8), we show how suspended constraints can coexist with ML *local let-generalization*—an indispensable feature of OCaml[1]—but one that makes suspended constraints uniquely difficult to implement and specify declaratively.

Contributions

Section §2 introduces our setting: OCaml's static overloading of datatype constructors and record labels, and polymorphic methods. We review directional inference, its limitations, and motivate omnidirectional inference. To this end, we introduce our three key ideas: a new characterization of *known* type information, *suspended match constraints*, and *incremental instantiation*, and demonstrate how together they enable principal type inference for these fragile features. Before turning to technical developments, we also discuss the *limitations and trade-offs* of omnidirectionality.

The subsequent sections present our main contributions:

- (§3) The OmniML calculus, an extension of ML featuring OCaml's static overloading of record labels and semi-explicit first-class polymorphism (§2.2). We give typing rules with a new declarative characterization of *known* type information.
- (§4) A novel constraint language for omnidirectional inference, equipped with a semantics for suspended constraints. We describe the translation of OmniML programs to constraints representing typing problems, and establish the expected metatheoretic properties: soundness, completeness, and principality of inference.
- (§5) A formal definition of our constraint solver as a series of non-deterministic rewriting rules, proved correct with respect to the constraint semantics. The rewriting rules detail our treatment for the interaction of let-generalization with suspended constraints via *incremental instantiation*, and the formal description can be directly related to an efficient implementation.
- (§6) A description of an efficient implementation of our solver, including our treatment of suspended constraints and partial type schemes. Validating that omnidirectional inference for ML is practical.

Section §7 considers additional practical and peripheral aspects of our work. Finally, §8 compares related work, and §9 concludes with future work. Appendix §A contains a complete technical reference, collecting key definitions and figures for convenient lookup. All proofs are deferred to the appendices.

2 Overview

We ground our work in two fragile features of OCaml: *static overloading* of record labels and constructors, and *polymorphic object methods*. Both are useful in practice: static overloading is widely relied upon in large programs, and polymorphic methods make first-class polymorphism available within OCaml.

[1] In contrast, Haskell only supports top-level implicit let-generalization.

2.1 Static overloading of constructors and record labels

Static overloading denotes a form of overloading in which resolution is performed entirely at compile time, enabling the compiler to select a unique implementation without relying on runtime information—in contrast to *dynamic overloading*, which defers resolution to runtime via mechanisms such as dictionary-passing or dynamic dispatch. Many mainstream languages, such as C++, Rust, and Java, use static overloading; its appeal is that it provides a *zero-cost* abstraction.

OCaml supports a limited yet useful form of static overloading for record labels and datatype constructors. Ambiguity is resolved using *known type information* under its directional inference algorithm (discussed in §2.3). To illustrate static overloading in OCaml, consider two nominal record types with overlapping field names:

```
type point      = { x : int; y : int }
type gray_point = { x : int; y : int; color : int }
```

With both definitions in scope, OCaml must statically disambiguate each field usage:

```
let one = { x = 42; y = 1337 }            OCaml ✓  OmniML ✓
let ex₁ r = r.x                            OCaml ✗  OmniML ✗
let ex₂ (r : point) = r.x + r.y            OCaml ✓  OmniML ✓
let ex₃ r = (r.x, (r : point).y)           OCaml ✗  OmniML ✓
```

The type of expression one has the unambiguous type point, even though both point and gray_point define the fields x and y. This is because OCaml performs *closed-world reasoning*: the typechecker is able to unambiguously infer the type of one as point, since it is the only record type whose domain is $\{x, y\}$. Similarly, r.color necessarily infers gray_point for the type of r.

By contrast, r.x is ambiguous unless the type of r is *known*. In ex_1, the type of r is unconstrained, so disambiguation fails.[2][3] In ex_2, the annotation fixes the type of r, thus r's type is *known* and resolves r.x and r.y unambiguously. In ex_3, the type of r can only be point: considering the second projection first, we learn that r must have the type point, and since it is λ-bound, this should make the first projection unambiguous. However, OCaml still rejects this example due to its *static order* of inference (§2.3).

Default rules. If local type information and closed-world reasoning are insufficient, OCaml falls back to a syntactic default: it selects the most recently defined compatible type. For example, OCaml accepts the following expression, when warnings (⚠) are not turned into errors (✗), *i.e.*, without the -warn-error flag[3]:

```
let ex₁ r = r.x                            OCaml ⚠  OmniML ✗
```

The expression is compatible with both point and gray_point, since each defines a field x. But gray_point is chosen simply because it appears later in the source.

Such fallback behavior is inherently *non-principal*: it reflects the typechecker's decision to abandon principal inference and arbitrarily select a syntactic default when no unique type can be inferred. We therefore give no formal account of such "default rules".

Defaulting also interacts poorly with OCaml's directional inference. Once the compiler selects a type, it commits to it—even if that choice causes errors downstream. Consider ex_3: when typing

[2] In fact, OCaml does not fail on ambiguous types, but instead applies a default resolution strategy: it emits a warning and selects the last matching type definition in scope. Here, this will amount to choosing the type gray_point for r.

[3] To check all our examples, use the options -principal -w +41+18 -warn-error +41+18, which enables principal type inference and escalates the associated warnings to errors.

$$e ::= [e : \sigma] \mid \langle e \rangle \mid (e : \tau) \mid \ldots \qquad \text{Terms}$$
$$\tau ::= [\sigma] \mid \ldots \qquad \text{Types}$$

Fig. 1. Selected syntax for polytypes [Garrigue and Rémy 1999].

`r.x`, OCaml defaults `r` to `gray_point`, and subsequently fails on `(r : point).y`. OmniML succeeds by suspending the resolution of `r.x` until it learns from `(r : point).y` that `r` has type `point`.

Since overloaded datatype constructors are analogous to record fields, we focus only on record fields in this work. Our prototype implementation (§6), however, supports both.

2.2 Polymorphic methods

Polymorphic methods [Garrigue and Rémy 1999] bring a form of System-F-like expressiveness to OCaml by supporting first-class polymorphism (impredicative higher-rank polymorphism) while preserving principal type inference.

From polymorphic methods to polytypes. Polymorphic methods can be translated into ordinary methods that carry a *polytype*. A polytype is a *boxed* type scheme $[\sigma]$ that must be explicitly unboxed at use sites. The syntax of polytypes is given in Figure 1. We write $[e : \sigma]$ to box a term e with the scheme σ, yielding an expression whose type is the polytype $[\sigma]$. Since polytypes are boxed, they cannot be be freely instantiated, unlike ordinary type schemes. An expression e of a polytype $[\sigma]$ must first be unboxed using the construct $\langle e \rangle$ before it can be typed at any instance τ of σ. Because they are boxed, polytypes can be treated as regular (mono)types, thereby enabling impredicativity.

Concretely, the polymorphic method of **object method** `id : α. α → α = ` **fun** `x → x` **end** is translated to **object method** `id = [` **fun** `x → x : α. α → α ]` **end**. Method invocation implicitly unboxes the polytype *e.g.* `x # id` becomes $\langle$`x # id`$\rangle$.

This reduction is useful for two reasons: (1) Inference for OCaml's object layer is largely governed by row-polymorphism, which is *robust* and does not threaten principality; it is therefore orthogonal to our concerns. In contrast, polytypes are *fragile*. (2) Polytypes underpin other features in OCaml, notably the recent addition of polymorphic function parameters [White 2023].

Default rules. As with static overloading in §2.1, OCaml employs a default rule for polytypes: when a polytype cannot be inferred as *known*, its polymorphic scheme is defaulted to a monomorphic type. This is particularly useful, as it allows method calls such as **fun** `bag x → bag#mem x` to typecheck without annotating `bag`. Here, the method invocation `bag # mem` introduces an unboxing operation, becoming $\langle$`bag # mem`$\rangle$. Since the type of this expression is not yet known, the default rule applies, assigning `bag`'s `mem` field the *monomorphic* polytype $[\alpha \to \beta]$.

Defaulting is not considered in our work. In practice, many users restrict their use of objects to method calls on objects of known types. In such cases, defaulting never arises. By contrast, defaulting appears essential for polymorphic parameters [White 2023]: they introduce polytype boxing and unboxing at every function application, with the default corresponding to the common monomorphic case. We therefore defer them to future work.

Semi-explicit first-class polymorphism. Polytypes, introduced by Garrigue and Rémy [1999], allow so-called *semi-explicit first-class polymorphism*, requiring explicit boxing an unboxing, but permit instantiations to be implicit. They therefore stand in contrast to *implicit* higher-rank or first-class polymorphism, provided by systems such as DK [Dunfield and Krishnaswami 2013], QuickLook [Serrano, Hage, Jones and Vytiniotis 2020], and Frost [Tang, Jiang, Oliveira and Lindley 2025], where polymorphic values are unboxed. At the same time, they are less explicit than *fully* *explicit* polymorphism (*e.g.* System-F), where both quantification and instantiation must be explicitly and fully specified.

We focus on polytypes in the remainder of this work for two reasons. First, our work is grounded in the features of OCaml, where polytypes are used for polymorphic methods and polymorphic function parameters. Second, polytypes expose the subtle interaction with principality that is of interest. We discuss approaches for implicit first-class polymorphism in §8.

2.3 Directional type inference

We now discuss the two main directional inference approaches: π-directional and bidirectional, illustrated using polytypes as a running example. We then discuss limitations of both approaches, providing us with the motivation for omnidirectional type inference.

π-directional type inference. Most ML type inference algorithms proceed in a fixed order when typechecking let-bindings $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$: first typecheck the definition e_1, generalize its type, and then typecheck the body e_2 under the extended environment. π-directionality leverages this ordering to resolve ambiguous constructs in a *principal* way. The key idea is that this ordering reveals which types are *known*, namely types that are fixed by generalization—and therefore stable enough to guide disambiguation. We call this approach π-directional (read as "**pi**-directional") type inference, reflecting that **p**olymorphic expressions must be typed before their **i**nstances.

To make this notion precise, we annotate types with *annotation variables* ε. We write τ^ε for a type τ annotated with the variable ε. Annotation variables record the origins of types and may themselves be generalized, yielding type schemes such as $\forall\varepsilon.\,\tau$, where ε appears free in τ; in particular, we may have a scheme $\forall\varepsilon.\,\tau^\varepsilon$, where ε annotates the topmost structure of τ. A type τ^ε is considered *known* when its (topmost) annotation variable ε is eligible for generalization (*e.g.* $\forall\varepsilon.\,\tau^\varepsilon$). Conversely, a type whose topmost annotation variable is monomorphic—that is, it cannot be generalized in the current context—is considered *not-yet-known* and cannot be relied on for disambiguation.

$\tau ::= \tau^\varepsilon \mid [\sigma] \mid \ldots$	Types
$\sigma ::= \tau \mid \forall\alpha.\,\sigma \mid \forall\varepsilon.\,\sigma$	Type schemes
ε	Annotation variables

$$\textsc{PolyML-Poly}\quad \frac{\Gamma \vdash e : \sigma_1 \qquad \sigma \Rightarrow_{\mathsf{ann}} \sigma_1 \qquad \sigma \Rightarrow_{\mathsf{ann}} \sigma_2}{\Gamma \vdash [e : \sigma] : [\sigma_2]^\varepsilon}$$

$$\textsc{PolyML-Inst}\quad \frac{\Gamma \vdash e : \forall\varepsilon.\,[\sigma]^\varepsilon}{\Gamma \vdash \langle e \rangle : \sigma}$$

$$\textsc{PolyML-Annot}\quad \frac{\Gamma \vdash e : \tau_1 \qquad \tau \Rightarrow_{\mathsf{ann}} \tau_1 \qquad \tau \Rightarrow_{\mathsf{ann}} \tau_2}{\Gamma \vdash (e : \tau) : \tau_2}$$

$$\textsc{Annot-Poly}\quad \frac{\sigma \Rightarrow_{\mathsf{ann}} \sigma'}{[\sigma] \Rightarrow_{\mathsf{ann}} [\sigma']^\varepsilon}$$

Fig. 2. π-directional syntax and typing rules for polytypes from PolyML [Garrigue and Rémy 1999].

Using this approach, we can give typing rules for polytypes; this yields precisely the formulation introduced in PolyML [Garrigue and Rémy 1999], given in Figure 2. The introduction form (PolyML-Poly) for polytypes is a boxing operator $[e : \sigma]$ with an explicit *closed* polytype annotation[4] σ (*i.e.*, σ contains no free type or annotation variables). The body e is checked against σ_1, a *freshened*

[4]For simplicity, we here restrict PolyML annotations to closed types. Both Garrigue and Rémy [1999] and our work support a more general form with existentially quantified type variables, as commonly used in OCaml.

copy of σ, *i.e.*, a variant of σ with fresh annotation variables placed at polytypes (Annot-Poly). The resulting expression has type $[\sigma_2]^\varepsilon$ where ε is an arbitrary (typically fresh) annotation variable and σ_2 is another copy of σ. Because σ is supplied by the programmer, the polytype is treated as known: $[e : \sigma]$ also has the generalized type scheme $\forall \varepsilon.\ [\sigma_2]^\varepsilon$. This is by design—the explicit annotation in $[e : \sigma]$ records that the polytype is known.

Conversely, to instantiate a polytype expression (PolyML-Inst), one must use an explicit unboxing operator $\langle e \rangle$, which requires no accompanying type annotation. However, the operator requires e to have a *known* polytype scheme of the form $\forall \varepsilon.\ [\sigma]^\varepsilon$ and then assigns $\langle e \rangle$ the type σ. If, by contrast, e has the type $[\sigma]^\varepsilon$ for some non-generalizable annotation variable ε, then e is considered as a *not-yet-known* polytype, and therefore $\langle e \rangle$ is ill-typed. This restriction enforces principality, preventing instantiation on *guessed* polytypes.

PolyML-Annot can be used to introduce fresh annotation variables. Intuitively, τ_1 and τ_2 are two *fresh copies* of the same closed annotation[4] τ, preventing unwanted sharing of annotation variables that could otherwise block generalization. For example, $\lambda x : [\sigma].\ \langle x \rangle$, which is syntactic sugar for $\lambda x.\ \text{let}\ x = (x : [\sigma])\ \text{in}\ \langle x \rangle$, is well-typed because the explicit annotation introduces a fresh annotation variable ε for $(x : [\sigma])$, which can then be generalized, yielding $\forall \varepsilon.\ [\sigma]^\varepsilon$ for the type of the let-bound variable x.

Building on its introduction in PolyML, π-directionality has since been adopted in other systems, notably MLF [Le Botlan and Rémy 2009], and in OCaml for features such as polymorphic object methods and the overloading of record fields and variant constructors.

We illustrate π-directionality using the following OCaml examples:[5]

```
let pid = [ fun x → x : α. α → α ]          OCaml ✓  OmniML ✓
let ex₅ = let p = pid in ⟨p⟩                 OCaml ✓  OmniML ✓
let ex₆ = (fun p → ⟨p⟩) pid                  OCaml ✗  OmniML ✓
```

At first glance, ex_5 and ex_6 appear equivalent: both simply instantiate the polytype bound to p. Yet, OCaml accepts ex_5 and rejects ex_6. This is because the let-binding in ex_5 assigns p the type scheme $\forall \varepsilon.\ [\forall \alpha.\ \alpha \to \alpha]^\varepsilon$, and thus its type is considered *known*—permitting unboxing (PolyML-Inst). In ex_6, by contrast, p is monomorphic at the point of instantiation as it is λ-bound, and unboxing is therefore forbidden.

To emphasize that this behavior is specification-driven and not an artifact of OCaml's inference algorithm, consider two equivalent versions of ex_6:[6]

```
let ex₆₂ = app (fun p → ⟨p⟩) pid             OCaml ✗  OmniML ✓
let ex₆₃ = rev_app pid (fun p → ⟨p⟩)         OCaml ✗  OmniML ✓
```

While these terms are semantically equivalent, they highlight a potential hazard: their typability may vary under a directionally biased inference algorithm, depending on whether the function or argument is typed first. To limit such implementation-dependent behavior, OCaml infers all subexpressions in an *order-independent* manner until they are let-bound. Consequently, OCaml does not make any distinction between ex_6, ex_{62}, and ex_{63}.

Treating both examples uniformly is in one sense a strength of π-directionality, but it also reveals a limitation: annotability is fragile, in that well-typedness depends on the *precise* placement of annotations, often forcing the programmer to introduce annotations that would otherwise be

[5]In OCaml, these examples can be typechecked by translating $[e : \sigma]$ to **object method** f : $\sigma = e$ **end** and $\langle e \rangle$ to e # f. This translation is the inverse of that discussed in §2.2.

[6]app and rev_app are the application function **fun** f x → f x and the reverse application function **fun** x f → f x, respectively.

unnecessary. For instance, the following two terms differ only in the position of the annotation, yet only $\texttt{self}_{21}$ is well-typed in OCaml—while they are both well-typed in OmniML:

```
let self₂₁ x = ⟨(x : [ α. α → α ])⟩ x        OCaml ✓  OmniML ✓
let self₂₂ x = ⟨x⟩ (x : [ α. α → α ])        OCaml ✗  OmniML ✓
```

Bidirectional typechecking. Bidirectional typechecking is a standard alternative to unification for propagating type information. It is typically formulated by splitting typing rules into two modes: *checking mode* ($\Gamma \vdash e \Leftarrow \tau$), which typechecks a term e against a type τ in a given context, and *inference mode* which infers e's type from the context alone ($\Gamma \vdash e \Rightarrow \tau$).

The type system designer assigns modes—checking or inference—to each language construct. For instance, one can decide to typecheck function applications $e_1\ e_2$ by first *inferring* that e_1 has some function type $\tau_1 \to \tau_2$, and then *checking* e_2 against τ_1 (Syn-App); but the opposite, mode-correct choice (Chk-App) is also possible:

$$\textsc{Syn-App}\ \frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 \Leftarrow \tau_1}{\Gamma \vdash e_1\ e_2 \Rightarrow \tau_2} \qquad \textsc{Chk-App}\ \frac{\Gamma \vdash e_1 \Leftarrow \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 \Rightarrow \tau_1}{\Gamma \vdash e_1\ e_2 \Leftarrow \tau_2}$$

Following the standard bidirectional recipe of checking introduction forms and inferring elimination forms—often referred to as the *Pfenning recipe* [Dunfield and Pfenning 2004]—the former rule (Syn-App) is typically preferred.

$$\textsc{Syn-Var}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x \Rightarrow \tau} \qquad \textsc{Chk-Fun}\ \frac{\Gamma, x : \tau_1 \vdash e \Leftarrow \tau_2}{\Gamma \vdash \lambda x.\, e \Leftarrow \tau_1 \to \tau_2} \qquad \textsc{Syn-App}\ \frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 \Leftarrow \tau_1}{\Gamma \vdash e_1\ e_2 \Rightarrow \tau_2} \qquad \textsc{Syn-Annot}\ \frac{\Gamma \vdash e \Leftarrow \tau}{\Gamma \vdash (e : \tau) \Rightarrow \tau}$$

$$\textsc{Chk-Forall}\ \frac{\Gamma \vdash e \Leftarrow \sigma \qquad \alpha \notin \mathrm{fv}(\Gamma)}{\Gamma \vdash e \Leftarrow \forall \alpha.\, \sigma} \qquad \textsc{Chk-Inst}\ \frac{\Gamma \vdash e \Rightarrow \sigma \qquad \sigma \leq \tau}{\Gamma \vdash e \Leftarrow \tau} \qquad \textsc{Chk-Poly}\ \frac{\Gamma \vdash e \Leftarrow \sigma}{\Gamma \vdash [e] \Leftarrow [\sigma]} \qquad \textsc{Syn-Inst}\ \frac{\Gamma \vdash e \Rightarrow [\sigma]}{\Gamma \vdash \langle e \rangle \Rightarrow \sigma}$$

Fig. 3. Bidirectional typing rules for polytypes.

Crucially, the assignment of modes determines how type information flows through the program. In particular, types become available either as inputs to checking judgments or as outputs of synthesizing judgments. We call such types *known*. Formally, a type τ is *known* when it is either: (1) part of an annotation, (2) supplied as input to a checking judgment in the conclusion ($\Gamma \vdash e \Leftarrow \tau$), or (3) produced by a synthesizing premise ($\Gamma \vdash e \Rightarrow \tau$).

Using this notion of known types, we can give a *principal* type system for polytypes, shown in Figure 3. This system eliminates two artifacts needed in the π-directional presentation: explicit annotations on boxing and annotation variables. Since $(e : \tau)$ already propagates known information, $[e]$ requires no attached annotation; and because "known-ness" now follows from inference modes rather than polymorphism, annotation variables are unnecessary.

While this approach yields a remarkably clean account of polytypes, it suffers from the fundamental limitation of bidirectional typing: there is generally no optimal assignment of modes. For any choice of modes, some programs will typecheck successfully, while others will fail unnecessarily. Yet, the typing rules must irrevocably commit to a fixed set of modes, after which, principal types exist, but only with respect to a specification that made non-principal choices to begin with. For instance, $\texttt{ex}_6$ would be ill-typed under the rules of Figure 3.

Limitations of directional type inference. Bidirectional typechecking is lightweight, practical, and well-suited for complex language features such as higher-rank polymorphism, dependent types, or subtyping. It supports the propagation of type information with minimal annotations. Its main downside lies in the need to fix an often arbitrary flow of type information—as in the case of function applications discussed above.

On the other hand, π*-directional* type inference appears better suited for ML: (1) thanks to its use of polymorphism—the essence of ML; and (2) its ability to be retrofitted easily onto existing typecheckers. But it remains surprisingly weak in some cases: it does not even allow the propagation of user-provided type annotations from a function to its argument! This weakness is sometimes counter-intuitive to the user. For example, the following would be rejected as ambiguous using π-directional type inference alone:

```
let ex₇ =                                              OCaml ✓  OmniML ✓
  let g (f : [ α. α → α ] → int) = f pid in
  g (fun p → ⟨p⟩ 42)
```

Here, p is λ-bound and therefore monomorphic. Without further propagation, the term ⟨p⟩ would be ambiguous, as no polymorphic (and thus *known*) type can be ascribed to p. OCaml resolves this by supplementing π-directional inference with a form of bidirectional propagation: the expected type of g's parameter (`[ α. α → α ] → int`) is bidirectionally propagated to the application of g, assigning p to have the *known* type `[ α. α → α ]` and thereby disambiguating ⟨p⟩.

2.4 Omnidirectional type inference

Omnidirectional inference infers typing constraints in any order. Constraints advance *dynamically*; those that require *known* type information suspend, and resume when other constraints supply it. This stands in contrast to the fixed *static* order of bidirectional and π-directional inference.

Consider again $\texttt{ex}_{62}$ and $\texttt{ex}_{63}$ from §2.3:

```
let ex₆₂ = app (fun p → ⟨p⟩) pid                        OCaml ✗  OmniML ✓
let ex₆₃ = rev_app pid (fun p → ⟨p⟩)                    OCaml ✓  OmniML ✓
```

Under π-directionality, both terms are ill-typed; under OCaml's bidirectional approach,[7] only $\texttt{ex}_{62}$ is rejected. Yet both terms have a principal type—it is merely a question of propagating type information in the right order. Omnidirectional type inference typechecks both: since it allows the typing of either side to proceed first, suspending the other until the relevant type information is *known*.

To be or not to be known. That is the question, indeed. To specify omnidirectionality declaratively, we must say when a type is considered *known* without relying on any fixed directional order. Our key idea is that a type is *known* when it is the *unique* type that can be inferred within some surrounding term context $\mathscr{E}$.

Take $\texttt{ex}_{62}$ as an example, using the following term context where □ denotes the hole:

```
let pid = [ fun x → x : α. α → α ]
let ex₆₂ = app (fun p → □) pid
```

Here, p has a uniquely inferrable type $[\forall\alpha.\,\alpha \to \alpha]$, no other type can be inferred for p. As a result, we can consider p's type *known*.

[7] In this example, we use the option `-no-principal` to enable bidirectional propagation in addition to π-directionality

$$\frac{\textsc{Use-I} \quad \mathscr{E}[e \triangleright [\sigma]] \qquad \Gamma \vdash \mathscr{E}[\langle e : \sigma \rangle] : \tau}{\Gamma \vdash \mathscr{E}[\langle e \rangle] : \tau}$$

Once the type is known, the fragile implicit term can be elaborated to a robust explicit counterpart. For example, the unboxing $\langle \mathsf{p} \rangle$ can be elaborated into the explicitly annotated form $\langle \mathsf{p} : \alpha.\ \alpha \to \alpha \rangle$. Consequently, to typecheck $\langle e \rangle$ under the context $\mathscr{E}$, it suffices to assert that e has the known polytype $[\sigma]$ and elaborate $\langle e \rangle$ into an annotated unboxing $\langle e : \sigma \rangle$, as captured by Use-I. The predicate $\mathscr{E}[e \triangleright [\sigma]]$—our *unicity condition*—formalizes precisely what it means for a type to be *known*. We defer its technical definition, which is rather subtle, to §3.4.

Suspension in action. Suspension is the mechanism that allows inference to proceed in any order, in spite of constructs that require *known* type information. In our framework, we realize this through our novel primitive: *suspended match constraints*.

A match constraint $(\mathsf{match}\ \tau\ \mathsf{with}\ \overline{\rho \Rightarrow C})$ pairs a (typically unknown) matchee type τ with a finite series of shape-pattern branches $\overline{\rho \Rightarrow C}$. Such constraints remain *suspended* until the *shape* of τ (*i.e.*, its top-level constructor) is known. Then, they are *discharged*: a unique branch is selected and its associated constraint has to be solved. A match constraint that is never discharged is considered unsatisfiable.

For now, it suffices to think of shapes as parts of types (*e.g.* the record type constructor T in $(\mathsf{rcd}\ T\ \bar{\tau})$[8]) while shape patterns ρ act as type 'destructors', binding parts of the type (*e.g.* the constructor name T) to meta-variables (*e.g.* the pattern variable t) used in $\bar{C}$. This will be made precise in §3.3.

We now illustrate the role of suspended constraints on our running *fragile* features: static overloading of records (and variants) and semi-explicit first-class polymorphism. Each feature translates the typability of the term into constraints, formalized using a constraint generation function of the form $[\![e : \tau]\!]$, which, given a term e and expected type τ, produces a constraint C which is satisfiable if and only if e has the type τ. As we will see, once we adopt the suspended constraint machinery developed in this paper, much of the complexity of these typing fragile constructs vanishes—suspended constraints do most of the heavy lifting.

For an ambiguous record projection $e.\ell$, we generate the typing constraint:

$$[\![e.\ell : \tau]\!] \triangleq \exists \alpha.\, [\![e : \alpha]\!] \wedge \mathsf{match}\ \alpha\ \mathsf{with}\ \mathsf{rcd}\ t\ \_ \Rightarrow t.\ell \leq (\alpha \to \tau)$$

This constraint introduces the unification variable α, unifying it with the type of e (via $[\![e : \alpha]\!]$), and suspends resolution of the return type τ until the type α of e becomes *known* to be some non-variable type τ_0. The branch then matches τ_0 against the record type pattern $(\mathsf{rcd}\ t\ \_)$. If τ_0 is a record type $(\mathsf{rcd}\ T\ \bar{\tau}_1)$, then the pattern binds the record name variable t to the record name T, otherwise the whole constraint fails. When t is bound to T, the right-hand-side constraint becomes $T.\ell \leq (\alpha \to \tau)$, which requires that the type of the projection of the label ℓ at type T in the global record-declaration environment can be instantiated into $(\alpha \to \tau)$.

When typechecking the polytype unboxing operator $\langle e \rangle$, if e is already known to have the type $[\sigma]$, then we can simply instantiate σ. However, if the type of e is not yet known—*i.e.*, it is a (possibly constrained) type variable α—then we must defer until more information is available. We capture this behavior with a suspended match constraint:

$$[\![\langle e \rangle : \tau]\!] \triangleq \exists \alpha.\, [\![e : \alpha]\!] \wedge \mathsf{match}\ \alpha\ \mathsf{with}\ [s] \Rightarrow s \leq \tau$$

The match remains suspended until α resolves to some type τ_0. If, upon resolution, τ_0 is $[\sigma]$, the pattern $[s]$ matches successfully, binding the polytype variable s to σ and performing the instantiation $\sigma \leq \tau$. Otherwise, the pattern does not match and the constraint fails.

[8]For readability in shape patterns, type constructors are written in prefix form and record type constructors are prefixed with rcd. This prefix is omitted in OCaml code, where type constructors appear in postfix position.

Scaling to ML. In the absence of (implicit) polymorphism, type inference is solely based on unification constraints which can be solved in any order; omnidirectional inference with suspended match constraints is then natural and easy to implement.

The difficulty originates from ML *implicit* `let`-polymorphism for which all known implementations follow the π-order: first typing the binding, generalizing it into a type scheme, and finally typing the body under the extended typing environment that binds the generalized scheme. The Hindley-Milner algorithm $\mathcal{J}$, its variants $\mathcal{W}$ or $\mathcal{M}$ [Lee and Yi 1998], or more flexible constraint-based type inference implementations [Odersky, Sulzmann and Wehr 1999; Pottier and Rémy 2005; Rémy 1990, 1992] all follow this strategy, to the best of our knowledge.

Consider the following program:

```
type α gpoint = { x : α; y : α }
let diag (n : α) : α gpoint = { x = n; y = n }

let ex₈ gp =                                          OCaml ✗  OmniML ✓
  let getx p = p.x in getx (diag 42), (getx gp : float)
```

We introduce a new parameterized record type `α gpoint`, whose fields `x` and `y` are overloaded—recall that both `point` and `gray_point` already define these fields—but here they have a *polymorphic* projection type $\forall\alpha.\ \alpha$ `gpoint` $\rightarrow \alpha$. The function `diag` constructs a diagonal point, a point lying on the diagonal $x = y$, and has the type $\forall\alpha.\ \alpha \rightarrow \alpha$ `gpoint`.

When typechecking $\texttt{ex}_8$, we cannot infer the type of `getx` first, since the type of `p.x` is still not yet known. Instead, we must typecheck the body first, where the call `getx (diag 42)` reveals that `p` has type `α gpoint`. Failing to do so would make inference incomplete, since the program is clearly well-typed. Nor can we treat the let-binding as monomorphic, since both calls to `getx` use different instantiations of α: indeed, α is int in `getx (diag 42)` while α is float in `(getx gp : float)`.

We solve this by introducing *incremental instantiation*, *i.e.*, the ability to instantiate type schemes that are not yet fully determined (so-called *partial type schemes*) and consequently revisit their instances when they are being refined, *incrementally*. This allows inferring parts of a let-body to disambiguate its definition, without duplicating constraint-solving work.

The forest, not the trees. Suspended match constraints offer a *general framework* to typing the features we have considered so far, and more. Some of those features can be handled using more specialized approaches: for example, SML employs row variables to support overloaded fields for structural records, while GHC uses qualified types to allow overloading of nominal record fields with a simple-enough type.

In contrast to these specialized mechanisms, suspended constraints are more expressive. They can handle cases where the typing rule to use on the subterms depends on the outcome of disambiguation, such as overloaded polymorphic record fields or overloaded GADT constructors in patterns. Moreover, these simpler approaches typically lack a declarative semantics that justify rejecting programs with unresolved disambiguation choices.

2.5 Limitations

Omnidirectional inference is powerful, but not omnipotent: (1) some programs are still rejected as ambiguous, even though a unique elaboration could in principle be chosen—a deliberate "Goldilocks" compromise; (2) our current formalization cannot disambiguate based on the *return type* of overloaded projections; (3) it presently omits a formalization of *default rules* (§2.1); and (4) our framework entails a higher conceptual and implementation complexity than static directional approaches (*e.g.* bidirectional typechecking).

Not too hot, not too cold. Some expressions must be rejected even though their elaboration would be unambiguous. Consider:

```
type cie_color = { x : int; y : int; z : int }
type cie_point = { x : int; y : int; color : cie_color }
let ex₁₀ r = r.color.x                                    OCaml ✗  OmniML ✗
```

Neither field projections in $\texttt{ex}_{10}$ can individually be disambiguated. However, if one were allowed to combine the constraints, they would jointly determine that `r` must have type `cie_point`: in `gray_point`, the field `color` has type `int`, and hence cannot itself be projected, leaving a unique consistent elaboration: `r.cie_point.color.cie_color.x`.[9]

Our framework nonetheless rejects $\texttt{ex}_{10}$. This is intentional: overloaded projections must be elaborated *sequentially*, each in isolation, rather than jointly with others. We view this restriction as a "Goldilocks" compromise: it rules out examples like the above, but avoids the intractability of full general overloading which is NP-hard, even without let-polymorphism, as shown by a reduction from 3-SAT [Charguéraud, Bodin, Dunfield and Riboulet 2025].

No returns accepted. At present, our system also resolves overloaded projections without taking their *return type* into account.

```
let ex₁₁ r = (r.color : cie_color)                        OCaml ✗  OmniML ✗
```

Under the rules defined in this work, programs such as $\texttt{ex}_{11}$ are rejected. This limitation is shared with existing languages such as OCaml, Haskell, and SML, none of which use the return type of a projection to guide disambiguation. Our framework already goes beyond these systems, but we believe that omnidirectionality provides the right foundation to incorporate return type disambiguation. We leave this refinement to future work.

No defaults, by default. We do not yet provide a formal account of *default rules* mentioned in §2.1 and §2.2. However, our prototype implementation, discussed in §6, does support (optionally) attaching a default strategy to each suspended constraint. In practice, defaulting proves useful and appears essential for richer features such as implicit first-class polymorphism. Developing a formal treatment of defaulting within our framework is an important direction for future work.

On complexity budgets. Omnidirectional type inference is conceptually straightforward but technically challenging. It follows a simple key idea: solving constraints in any order, suspending when known type information is required. However, realizing this idea precisely and efficiently comes at a higher complexity cost than bidirectional or π-directional type inference.

- (§3) Giving a declarative characterization of *known* type information without statically relying on directionality is hard. Our contextual rules nicely solve this problem, but their meta-theory is unsurprisingly more complex than local rules.
- (§6) Implementing efficient incremental instantiation is harder than ML instantiation, as refinements of partial type schemes must trigger re-instantiations while avoiding solving the same constraints repeatedly across successive instantiations.

We hope to pay off some of this complexity in future work; in particular, we believe omnidirectionality is the missing piece to unlock modular implicits [White, Bour and Yallop 2014]—an approach to generalized overloading and a long-anticipated feature within the OCaml community.

$$
\begin{array}{rcll}
\alpha, \beta, \gamma & \in & \mathcal{V} & \text{Type variables} \\
\tau & ::= & \alpha \mid 1 \mid \tau_1 \to \tau_2 \mid \mathsf{rcd}\ T\ \bar{\tau} \mid [\sigma] & \text{Types} \\
\sigma & ::= & \tau \mid \forall \alpha.\, \sigma & \text{Type schemes} \\
T & & & \text{Record name} \\
\mathfrak{g} & \in & \mathcal{G} & \text{Ground types} \\
e & ::= & x \mid () \mid \lambda x.\, e \mid e_1\ e_2 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid (e : \exists \bar{\alpha}.\, \tau) & \text{Terms} \\
 & \mid & [e] \mid [e : \exists \bar{\alpha}.\, \sigma] \mid \langle e \rangle \mid \langle e : \exists \bar{\alpha}.\, \sigma \rangle & \\
 & \mid & \{\overline{\ell = e}\} \mid e.\ell \mid T.\{\overline{\ell = e}\} \mid e.T.\ell & \\
\Gamma & ::= & \emptyset \mid \Gamma, x : \sigma & \text{Contexts} \\
\Omega & ::= & \emptyset \mid \Omega, T.\ell : \forall \bar{\alpha}.\, \mathsf{rcd}\ T\ \bar{\alpha} \to \tau & \text{Label contexts} \\
\zeta & ::= & \nu \bar{\gamma}.\, \tau \qquad (\zeta \in \mathcal{S}) & \text{Shapes} \\
\varsigma & \in & \mathcal{S}_{\mathsf{canon}} \qquad (\mathcal{S}_{\mathsf{canon}} \subset \mathcal{S}) & \text{Canonical principal shapes}
\end{array}
$$

$$
\textsc{Var}\ \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}
\qquad
\textsc{Fun}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\, e : \tau_1 \to \tau_2}
\qquad
\textsc{App}\ \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}
\qquad
\textsc{Unit}\ \frac{}{\Gamma \vdash () : 1}
$$

$$
\textsc{Gen}\ \frac{\Gamma \vdash e : \sigma \qquad \alpha \# \mathsf{fv}(\Gamma)}{\Gamma \vdash e : \forall \alpha.\, \sigma}
\qquad
\textsc{Inst}\ \frac{\Gamma \vdash e : \forall \alpha.\, \sigma}{\Gamma \vdash e : \sigma[\alpha := \tau]}
\qquad
\textsc{Let}\ \frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau}
$$

$$
\textsc{Annot}\ \frac{\Gamma \vdash e : \tau[\bar{\alpha} := \bar{\tau}]}{\Gamma \vdash (e : \exists \bar{\alpha}.\, \tau) : \tau[\bar{\alpha} := \bar{\tau}]}
\qquad
\textsc{Poly-X}\ \frac{\Gamma \vdash e : \sigma[\bar{\alpha} := \bar{\tau}]}{\Gamma \vdash [e : \exists \bar{\alpha}.\, \sigma] : [\sigma[\bar{\alpha} := \bar{\tau}]]}
\qquad
\textsc{Use-X}\ \frac{\Gamma \vdash e : [\sigma][\bar{\alpha} := \bar{\tau}]}{\Gamma \vdash \langle e : \exists \bar{\alpha}.\, \sigma \rangle : \sigma[\bar{\alpha} := \bar{\tau}]}
$$

$$
\textsc{Rcd-X}\ \frac{\mathsf{dom}\ (\Omega(T)) = \bar{\ell} \qquad (T.\ell_i \le \tau \to \tau_i)_{i=1}^{n} \qquad (\Gamma \vdash e_i : \tau_i)_{i=1}^{n}}{\Gamma \vdash T.\{\ell_1 = e_1\ ;\ \ldots\ ;\ \ell_n = e_n\} : \tau}
\qquad
\textsc{Rcd-Closed}\ \frac{\bar{\ell} \blacktriangleright T \qquad \Gamma \vdash T.\{\overline{\ell = e}\} : \tau}{\Gamma \vdash \{\overline{\ell = e}\} : \tau}
$$

$$
\textsc{Rcd-Proj-X}\ \frac{T.\ell \le \tau_1 \to \tau_2 \qquad \Gamma \vdash e : \tau_1}{\Gamma \vdash e.T.\ell : \tau_2}
\qquad
\textsc{Rcd-Proj-Closed}\ \frac{\ell \triangleright T \qquad \Gamma \vdash e.T.\ell : \tau}{\Gamma \vdash e.\ell : \tau}
\qquad
\textsc{Lab-Inst}\ \frac{\Omega(T.\ell) = \forall \bar{\alpha}.\, \mathsf{rcd}\ T\ \bar{\alpha} \to \tau}{T.\ell \le (\mathsf{rcd}\ T\ \bar{\tau} \to \tau[\bar{\alpha} := \bar{\tau}])}
$$

Fig. 4. Syntax and explicit, robust typing rules of OmniML.

3 The OmniML calculus

To prove correctness of type inference, we must define a language and its type system. Identifying an appropriate declarative type system to use as a specification is itself a challenging problem. In particular, natural specifications for fragile features often fail to preserve principality.

Consider records, for instance. We can ask the user to provide a type annotation by using an explicit record projection $e.T.\ell$, which has a simple typing rule (Rcd-Proj-X in Figure 4). By contrast, the natural typing rule for the overloaded projections $e.\ell$ breaks principality (Rcd-Proj-I-Nat). For example, term $\lambda r.\, r.\mathsf{x}$ admits two incompatible types, `point` $\to$ `int` and `gray_point` $\to$ `int`, as explained in §2.

$$
\textsc{Rcd-Proj-I-Nat}\ \frac{T.\ell \le \tau_1 \to \tau_2 \qquad \Gamma \vdash e : \tau_1}{\Gamma \vdash e.\ell : \tau_2}
$$

[9]The syntax $e.T.\ell$ qualifies the label ℓ to unambiguously belong to the type T in the projection.

To restore principality for overloaded projections, we require that the type of e be *known*—that is, determined to be a specific record type rcd $T\ \bar{\tau}$, rather than merely *guessed*. As discussed earlier (§2.4), this requirement is expressed via a *unicity condition*, giving rise to the contextual rule Rcd-Proj-I (Figure 5), which elaborates the fragile overloaded projection $e.\ell$ into its robust explicit counterpart $e.T.\ell$.

3.1 Syntax

OmniML (Figure 4) extends ML with two fragile constructs: polytypes and nominal records. Variants are not treated formally in OmniML, but behave analogously to records.

Notation for collections. We write $\overline{X}$ for a (possibly empty) set of elements $\{X_1, \ldots, X_n\}$ and a (possibly empty) sequence $X_1, \ldots, X_n$. The interpretation of whether $\overline{X}$ is a set or a sequence is often implicit. We write $\overline{X} \# \overline{X'}$ as a shorthand for when $\overline{X} \cap \overline{X'} = \emptyset$. We write $\overline{X}, \overline{X'}$ as the union or concatenation (depending on the interpretation) of $\overline{X}$ and $\overline{X}'$. We often write X for the singleton set (or sequence).

Types. Monotypes (or just types) include, as usual, type variables α, the unit type 1, arrow types, but also nominal record types rcd $T\ \bar{\tau}$, and polytypes $[\sigma]$. We use a non-standard syntax rcd $T\ \bar{\tau}$ for record types, where T is a *record name* and $\bar{\tau}$ a list of type parameters.[10] Type schemes σ are of the form $\forall\bar{\alpha}.\,\tau$, they are equal up to the reordering of binders and removal of useless variables. Standard notions such as the set of free variables $\mathsf{fv}(\sigma)$ and capture-avoiding substitutions $\sigma[\alpha := \tau]$ are defined in the usual way. We write $\mathcal{V}$ for the set of type variables, and use $\bar{\alpha} \# \sigma$ as a short-hand for $\bar{\alpha} \# \mathsf{fv}(\sigma)$. Finally, $\mathfrak{g} \in \mathcal{G}$ denotes a *ground type*—a type with no free variables.[11]

Terms. Terms of OmniML are variables x, the unit literal $()$, lambda-abstractions $\lambda x.\,e$, applications $e_1\ e_2$, annotations $(e : \exists\bar{\alpha}.\,\tau)$, and let-bindings let $x = e_1$ in e_2, extended with the following expressions:

- For polytypes, we introduce implicit and explicit boxing and unboxing forms: $[e]$, $[e : \exists\bar{\alpha}.\,\sigma]$, and $\langle e\rangle$, $\langle e : \exists\bar{\alpha}.\,\sigma\rangle$ respectively.
- Overloaded record labels include record literals $\{\ell_1 = e_1\ ;\ \ldots\ ;\ \ell_n = e_n\}$ and field projections $e.\ell$. Both constructs have explicit counterparts: $T.\{\ell_1 = e_1\ ;\ \ldots\ ;\ \ell_n = e_n\}$ and $e.T.\ell$, where the explicit record annotation $T.\_$ indicates that the labels unambiguously belong to the record type T.

All annotations in OmniML are closed *i.e.*, their existentially quantified variables $\bar{\alpha}$ are exactly the free variables of the type τ or type scheme σ. We use e^i to range over fragile, implicit terms (*e.g.* $e.\ell$) and e^x for their explicit counterparts (*e.g.* $e.T.\ell$).

Typing rules for explicit terms are mostly standard; nominal records require a more intricate (yet largely folklore) treatment of *closed world* reasoning. The crux of our work is the novel typing of the fragile constructs, presented in §3.4.

3.2 Typing rules for robust, explicit constructs

As usual, the main typing judgment $\Gamma \vdash e : \sigma$ (Figure 4) states that in context Γ, the expression e has the type (scheme) σ. Rules Var through Let are standard. Annotations $(e : \exists\bar{\alpha}.\,\tau)$ (Annot) ensures that the type of e is (an instance of) the type τ. The type variables $\bar{\alpha}$ are *flexibly* (or existentially) bound in τ, meaning they may be instantiated to some types $\bar{\tau}$ so that the resulting annotation

[10] Using T as an argument of (rcd $T\ \bar{\tau}$) rather a head constructor ($T\ \bar{\tau}$) will make record shape patterns easier to understand.
[11] $\mathfrak{g}$ can be pronounced "Fraktur g" or just "ground g".

matches the type of e. For instance, the term $(\lambda x.\, x + 1 : \exists \alpha.\, \alpha \to \alpha)$ is well-typed under Annot with the substitution $[\alpha := \text{int}]$.

Explicit polytypes. Poly-X serves as the introduction rule: given the (closed) type scheme σ, it forms a first-class polytype $[\sigma]$, requiring the expression e to be at least as polymorphic as σ. Use-X is the corresponding elimination rule, unpacking an expression of polytype $[\sigma]$ into one of polymorphic type σ, which may be freely instantiated (via Inst). Both rules also allow polytype annotations to be partial, *i.e.*, σ may have free type variables $\bar{\alpha}$, which are existentially quantified to close the annotation, as in Annot.

Explicit nominal records. We assume a global label context Ω mapping labels to their projection type, *i.e.*, type schemes of the form $\forall \bar{\alpha}.\, \text{rcd}\ T\ \bar{\alpha} \to \tau$. A label ℓ may belong to multiple record types, but is unique within each record type T. We write $\Omega(T)$ for the set of labels belonging to the record type T: $\Omega(T) \triangleq \{\ell : T.\ell : \forall \bar{\alpha}.\, \text{rcd}\ T\ \bar{\alpha} \to \tau \in \Omega\}$. We write $\Omega(T.\ell)$ for the unique scheme $\forall \bar{\alpha}.\, \text{rcd}\ T\ \bar{\alpha} \to \tau$ associated with ℓ in T (if defined).

Label instantiations are typed by an auxiliary judgment $T.\ell \le \tau_1 \to \tau_2$ defined by Lab-Inst and meaning that $\tau_1 \to \tau_2$ is an instance of the projection scheme $\Omega(T.\ell)$. Explicit field projections (Rcd-Proj-X) require that $e.T.\ell$ projects from a record e of type τ_1 to τ_2, provided $T.\ell \le \tau_1 \to \tau_2$ holds. Explicit records (Rcd-X) are typed similarly, checking that each field has the appropriate type. In addition, the premise asserts that the fields $\bar{\ell}$ appearing in the record expression exactly match the labels of T (*i.e.*, $\text{dom}\ (\Omega(T))$).

Following OCaml, our explicit system also supports *closed-world* reasoning, which exploits the absence of ambiguity in the label context Ω to infer record annotations. In particular, in a record expression $\{\ell_1 = e_1\ ; \dots ;\ \ell_n = e_n\}$, if the set of labels $\ell_1, \dots, \ell_n$ uniquely identifies a record type T in the context Ω, then the record has the type of $T.\{\ell_1 = e_1\ ; \dots ;\ \ell_n = e_n\}$ (via Rcd-Closed). Similarly, if the label ℓ is associated with exactly one record type T in Ω, then the projection $e.\ell$ has the type of $e.T.\ell$ (by Rcd-Proj-Closed).

These two forms of label uniqueness differ. A *closed set* of labels may uniquely identify a record type even if no individual label is unique. Conversely, a unique label implies uniqueness of every closed set containing it. For instance, recall `one` from §2.1:

```
type point  = { x : int; y : int }
type gray_point = { x : int; y : int; color : int }
let one = { x = 42; y = 1337 }                    OCaml ✓  OmniML ✓
```

Here, the closed set $\{\mathsf{x}, \mathsf{y}\}$ in `one` uniquely identifies `point`, even though the individual labels `x` and `y` also appear in `gray_point`.

We formalize this *closed world uniqueness* using the predicates $\ell \triangleright T$ (a label uniquely identifies T) and $\bar{\ell} \blacktriangleright T$ (a closed label set uniquely identifies T):

$$\begin{aligned} \ell \triangleright T &\triangleq \ell \in \text{dom}\ (\Omega(T)) \ \wedge\ \forall T',\ (\ell \in \text{dom}\ (\Omega(T')) \implies T = T') \\ \bar{\ell} \blacktriangleright T &\triangleq \text{dom}\ (\Omega(T)) = \bar{\ell} \ \wedge\ \forall T',\ (\text{dom}\ (\Omega(T')) = \bar{\ell} \implies T = T') \end{aligned}$$

These predicates depend only on the global label environment Ω: they ignore field types and require no contextual type information. The associated typing rules (Rcd-Closed, Rcd-Proj-Closed) are therefore *robust*, since disambiguation relies solely on globally known label information rather than type-directed disambiguation.

3.3 Shapes

We introduce *shapes* as a generalization of type constructors. They provide a uniform treatment of both constructors and polytypes, and are useful in defining polytype unification (§6).

A shape ζ is a type with holes, written $\nu\bar{\gamma}.\,\tau$, where $\bar{\gamma}$ denotes the set of type variables representing the holes. By construction, we require $\bar{\gamma}$ to be *exactly* the free variables of τ. Hence, shapes are closed and do not contain useless binders. We consider shapes equal up to α-conversion. When τ is a ground type, we omit the binder and simply write τ for the shape. We use $\bot$ to denote the shape $\nu\gamma.\,\gamma$, which we call the *trivial* shape. Let $\mathcal{S}$ denote the set of all shapes and $\mathcal{S}^* \subset \mathcal{S}$ the set of non-trivial shapes.

Shapes are equipped with the standard instantiation ordering, defined by Inst-Shape. When writing $\zeta \preceq \zeta'$, we say that ζ is more general than ζ'. When ζ and ζ' are more general than one another, they are actually equal. The trivial shape $\bot$ is the most general shape. If ζ is $\nu\bar{\gamma}.\,\tau$, the shape application $\zeta\;\bar{\tau}$ is defined as $\tau[\bar{\gamma} := \bar{\tau}]$. We say that ζ is a shape of τ when there exists $\bar{\tau}$ such that $\tau = \zeta\;\bar{\tau}$; in this case, we call the pair $(\zeta, \bar{\tau})$ a decomposition of τ.

Inst-Shape
$$\frac{}{\nu\bar{\gamma}_1.\,\tau \preceq \nu\bar{\gamma}_2.\,\tau[\bar{\gamma}_1 := \bar{\tau}_1]}$$

Definition 3.1. A non-trivial shape $\zeta \in \mathcal{S}^*$ is the principal shape of the type τ iff:

(1) $\exists \bar{\tau}',\ \tau = \zeta\;\bar{\tau}'$
(2) $\forall \zeta' \in \mathcal{S}^*, \forall \bar{\tau}',\ \tau = \zeta'\;\bar{\tau}' \implies \zeta \preceq \zeta'$

Theorem 3.2 (Principal shapes). *Any non-variable type τ has a principal shape ζ.*

A principal shape $\nu\bar{\gamma}.\,\tau$ is *canonical* if its free variables appear in the sequence $\bar{\gamma}$ in the order in which they occur in τ. Canonical principal shapes are written ς, and we write $\mathcal{S}_{\mathsf{canon}}$ for the set of all such shapes. Each non-variable type τ has a unique canonical principal shape, written $\mathsf{shape}(\tau)$. For example, $\mathsf{shape}(\mathsf{rcd}\;T\;\bar{\tau})$ is $\nu\bar{\gamma}.\,\mathsf{rcd}\;T\;\bar{\gamma}$.

Shapes are particularly interesting in the context of polytypes, which can be decomposed into shapes and thus treated analogously to type constructors. For instance, consider the polytype $[\forall\alpha.\,([\forall\beta.\,(\alpha \to \mathsf{int\ list}) * \beta]) \to \alpha \to \alpha]$. Since its principal shape ς is $\nu\gamma.\,[\forall\alpha.\,([\forall\beta.\,(\alpha \to \gamma) * \beta]) \to \alpha \to \alpha]$, the original polytype can therefore be represented as the shape application $\varsigma\,(\mathsf{int\ list})$.

3.4 Typing rules for fragile, implicit constructs

$$e ::= \ldots \mid \square \text{ with } \bar{e} \qquad \text{Terms}$$
$$\mathscr{E} ::= \square \mid \mathscr{E}\;e \mid e\;\mathscr{E} \mid \ldots \qquad \text{Term contexts}$$

Hole
$$\frac{(\Gamma \vdash e_i : \tau_i)_{i=1}^{n}}{\Gamma \vdash \square \text{ with } \bar{e} : \tau'}$$

Use-I
$$\frac{\mathscr{E}[e \triangleright \nu\bar{\gamma}.\,[\sigma]] \qquad \Gamma \vdash \mathscr{E}[\langle e : \exists\bar{\gamma}.\,\sigma\rangle] : \tau}{\Gamma \vdash \mathscr{E}[\langle e\rangle] : \tau}$$

Poly-I
$$\frac{\mathscr{E}[\square \triangleleft \nu\bar{\gamma}.\,[\sigma] \mid e] \qquad \Gamma \vdash \mathscr{E}[[e : \exists\bar{\gamma}.\,\sigma]] : \tau}{\Gamma \vdash \mathscr{E}[[e]] : \tau}$$

Rcd-I
$$\frac{\mathscr{E}[\square \triangleleft \nu\bar{\gamma}.\,\mathsf{rcd}\;T\;\bar{\gamma} \mid \bar{e}] \qquad \Gamma \vdash \mathscr{E}[T.\{\overline{\ell = e}\}] : \tau}{\Gamma \vdash \mathscr{E}[\{\overline{\ell = e}\}] : \tau}$$

Rcd-Proj-I
$$\frac{\mathscr{E}[e \triangleright \nu\bar{\gamma}.\,\mathsf{rcd}\;T\;\bar{\gamma}] \qquad \Gamma \vdash \mathscr{E}[e.T.\ell] : \tau}{\Gamma \vdash \mathscr{E}[e.\ell] : \tau}$$

$$\mathscr{E}[e \triangleright \varsigma \mid \bar{e}] \triangleq \forall \Gamma, \tau, \mathfrak{g},\ \Gamma \vdash \lfloor \mathscr{E}[\square \text{ with } \bar{e}, (e : \mathfrak{g})] \rfloor : \tau \implies \mathsf{shape}(\mathfrak{g}) = \varsigma$$

$$\mathscr{E}[\square \triangleleft \varsigma \mid \bar{e}] \triangleq \forall \Gamma, \tau, \mathfrak{g},\ \Gamma \vdash \lfloor \mathscr{E}[((\square \text{ with } \bar{e}) : \mathfrak{g})] \rfloor : \tau \implies \mathsf{shape}(\mathfrak{g}) = \varsigma$$

Fig. 5. Typing rules for fragile, implicitly typed extensions.

We now turn to the typing of fragile implicit constructs (Figure 5). To prevent the kind of uncontrolled guessing permitted by *natural* typing rules (*e.g.* Rcd-Proj-I-Nat), we adopt *contextual* typing rules: an implicit term e^i is typed *within* a surrounding one-hole term context $\mathscr{E}$ (Figure 5). The context $\mathscr{E}$ is used to ensure that the relevant type information (*e.g.* a shape ς) is *known*—that is, it is the unique shape ς that can be inferred from the context, rather than an arbitrary one that could be *guessed*. Such rules are therefore inherently non-compositional.

Unicity. The key question for our contextual typing rules is whether the shape ς of a term's type is *uniquely determined* by its surrounding context $\mathscr{E}$. We capture this with our *unicity conditions* $\mathscr{E}[e \triangleright \varsigma \mid \bar{e}]$ and $\mathscr{E}[\square \triangleleft \varsigma \mid \bar{e}]$, which state that all valid typings of the (erased) context $\mathscr{E}$ assign the same canonical shape ς to the subterm e and the context's hole $\square$, respectively.

In other words, $\mathscr{E}[e \triangleright \varsigma \mid \bar{e}]$ holds when the type of the subterm e has a unique shape ς fixed by its context $\mathscr{E}$ and sibling terms $\bar{e}$, while $\mathscr{E}[\square \triangleleft \varsigma \mid \bar{e}]$ holds when the expected type of the hole in $\mathscr{E}$ (with subterms $\bar{e}$) has the unique shape ς.

$$\begin{aligned} \mathscr{E}[e \triangleright \varsigma \mid \bar{e}] &\triangleq \forall \Gamma, \tau, \mathfrak{g},\ \Gamma \vdash \lfloor \mathscr{E}[\square \text{ with } \bar{e}, (e : \mathfrak{g})] \rfloor : \tau \implies \mathsf{shape}(\mathfrak{g}) = \varsigma \\ \mathscr{E}[\square \triangleleft \varsigma \mid \bar{e}] &\triangleq \forall \Gamma, \tau, \mathfrak{g},\ \Gamma \vdash \lfloor \mathscr{E}[((\square \text{ with } \bar{e}) : \mathfrak{g})] \rfloor : \tau \implies \mathsf{shape}(\mathfrak{g}) = \varsigma \end{aligned}$$

To make sense of these definitions, we rely on a few auxiliary notions:

(1) We write $(\square \text{ with } \bar{e})$ for a *typed hole* carrying subterms $\bar{e}$. The subterms $\bar{e}$ are required to be well-typed in the current environment (Hole), but their types are independent of the type of the hole: the hole itself may be assigned an arbitrary type.
(2) We introduce an *erasure* function $\lfloor e \rfloor$ that erases all not-yet-elaborated implicit constructs e^i in e with a typed hole around their subterms. For example, $\lfloor e.\ell \rfloor$ is $(\square \text{ with } \lfloor e \rfloor)$. The full definition is given in Figure 6.

$$\begin{aligned} \lfloor [e] \rfloor &\triangleq \square \text{ with } \lfloor e \rfloor \\ \lfloor [e : \exists \bar{\alpha}.\, \sigma] \rfloor &\triangleq [\lfloor e \rfloor : \exists \bar{\alpha}.\, \sigma] \\ \lfloor \langle e \rangle \rfloor &\triangleq \square \text{ with } \lfloor e \rfloor \\ \lfloor \langle e : \exists \bar{\alpha}.\, \sigma \rangle \rfloor &\triangleq \langle \lfloor e \rfloor : \exists \bar{\alpha}.\, \sigma \rangle \\ \lfloor \{\ell_1 = e_1 ; \ldots ; \ell_n = e_n\} \rfloor &\triangleq \begin{cases} \{\ell_1 = \lfloor e_1 \rfloor ; \ldots ; \ell_n = \lfloor e_n \rfloor\} & \text{if } \bar{\ell} \blacktriangleright T \\ \square \text{ with } \lfloor e_1 \rfloor, \ldots, \lfloor e_n \rfloor & \text{otherwise} \end{cases} \\ \lfloor T.\{\ell_1 = e_1 ; \ldots ; \ell_n = e_n\} \rfloor &\triangleq T.\{\ell_1 = \lfloor e_1 \rfloor ; \ldots ; \ell_n = \lfloor e_n \rfloor\} \\ \lfloor e.\ell \rfloor &\triangleq \begin{cases} \lfloor e \rfloor.\ell & \text{if } \ell \triangleright T \\ \square \text{ with } \lfloor e \rfloor & \text{otherwise} \end{cases} \\ \lfloor e.T.\ell \rfloor &\triangleq \lfloor e \rfloor.T.\ell \\ \lfloor \square \text{ with } \bar{e} \rfloor &\triangleq \square \text{ with } (\lfloor e_i \rfloor)_{i=1}^{n} \end{aligned}$$

Fig. 6. Selected cases of the erasure of e. All other cases are homomorphic.

Typed holes ensure that subterms—such as type annotations—remain present even when the implicit construct containing them is erased. This is because they may introduce constraints that can still contribute to unicity. For instance, $\lambda r.\ (r : \texttt{point}).\texttt{x}$ would be ill-typed if we erased $(r : \texttt{point})$.

The erasure of $\lfloor \mathscr{E}[\square \text{ with } (e : \mathfrak{g}), \bar{e}] \rfloor$ and $\lfloor \mathscr{E}[((\square \text{ with } \bar{e}) : \mathfrak{g})] \rfloor$ replaces all not-yet-elaborated implicit constructs with typed holes, ensuring that the unicity of ς is determined *only* by implicit terms that have been elaborated thus far. This induces a causal, or partial, order among implicit constructs: an implicit term can only be elaborated once all of its dependencies—those providing the relevant *known* type information—have themselves been elaborated. This ordering prevents self-justifying cycles in unicity and rules out "out-of-thin-air" guesses.

The attentive reader may notice that our unicity conditions contain a negative occurrences of the typing judgment. At first glance, this seems problematic for well-foundedness. However, the issue is easily resolved by noting that these occurrences arise only as typing assumptions on erased terms, which do not themselves involve any implicit rules. Formally one could make this explicit by introducing a restricted typing judgment $\Gamma \vdash_{\text{robust}} e : \tau$ that excludes the implicit rules (*-I), and by using this strictly simpler judgment in the antecedent of unicity conditions. We omit this distinction for simplicity.

The omnidirectional recipe. All typing rules instantiate a common framework—the *omnidirectional recipe*—which ensures that certain omitted type annotations are uniquely determined from the context. Each construct, however, requires a specific instantiation of the framework. We first describe the framework, then present each feature separately.

Step 1: *Contextualize.* Each implicit fragile term e^i is typed relative to a surrounding one-hole term context $\mathscr{E}$: its rule asserts the typability of $\Gamma \vdash \mathscr{E}[e^i] : \tau$ as the conclusion.

Step 2: *Select a unicity condition.* This is the secret ingredient! The unicity condition ensures that the shape ς is fully determined by the surrounding context $\mathscr{E}$ and subexpressions $\bar{e}$ (*e.g.* the subexpressions $\bar{e}$ in $\{\overline{\ell = e}\}$).
If the e^i is an introduction form, we infer the shape from the context's hole $\mathscr{E}[\square \triangleleft \varsigma \mid \bar{e}]$. If e^i is an elimination form, we infer the shape from the *principal term e* (the term whose type contains the connective we're eliminating): $\mathscr{E}[e \triangleright \varsigma \mid \bar{e}]$.
This division of terms is reminiscent of the *Pfenning recipe* for bidirectional typechecking [Dunfield and Pfenning 2004], where $\mathscr{E}[\square \triangleleft \varsigma \mid \bar{e}]$ can be viewed as a checking mode, and $\mathscr{E}[e \triangleright \varsigma \mid \bar{e}]$ as an inference mode for shapes.

Step 3: *Elaborate.* The uniquely inferred shape ς is used to elaborate e^i into its explicit counterpart e^x, and the rule asserts $\Gamma \vdash \mathscr{E}[e^x] : \tau$ as a premise.

Implicit polytypes. Unboxing a polytype $\langle e \rangle$ is an *elimination form*. Following the omnidirectional recipe, Use-I is a contextual rule (*Step 1*) requiring that the principal term e have the unique shape $\nu\bar{\gamma}.\,[\sigma]$ in the context $\mathscr{E}$ (*Step 2*). In *Step 3*, we then elaborate $\langle e \rangle$ into $\langle e : \exists\bar{\gamma}.\,\sigma \rangle$. Conversely, boxing with $[e]$ is an *introduction form*. In Poly-I, we require that the expected type of the context's hole $\mathscr{E}$ has the shape $\nu\bar{\gamma}.\,[\sigma]$ (*Step 2*). We then type $[e]$ as $[e : \exists\bar{\gamma}.\,\sigma]$ (*Step 3*).

Implicit nominal records. Overloaded record labels are handled analogously. Typing record projections in Rcd-Proj-I is an *elimination form* for the record type rcd $T\ \bar{\tau}$: the projection $e.\ell$ is typed as $e.T.\ell$ (*Step 3*) provided the type of expression e in context $\mathscr{E}$ has record shape $\nu\bar{\gamma}.\,\text{rcd}\ T\ \bar{\gamma}$ (*Step 2*). For record construction, $\{\overline{\ell = e}\}$ is an *introduction form*. In Rcd-I, we type an overloaded record $\{\overline{\ell = e}\}$ as $T.\{\overline{\ell = e}\}$ (*Step 3*), provided the context $\mathscr{E}$ with subterms $\bar{e}$ expects a record type of shape $\nu\bar{\gamma}.\,\text{rcd}\ T\ \bar{\gamma}$ (*Step 2*).

We now illustrate the typing of implicit constructs with a few examples.

Example 3.3. Consider the term $\texttt{ex}_3 \triangleq \lambda r.\,(r.\texttt{x}, (r : \texttt{point}).\texttt{y})$ from §2.1.[12] In $\texttt{ex}_3$, r can only be of type `point`. Indeed, considering the second projection first, we should learn that r is of type `point` (using Annot) and since it is λ-bound, this makes the first projection unambiguous. (For record types without parameters, we use T as a shorthand for rcd $T\ \emptyset$.)

[12]The typing rules for tuples are standard and present in Appendix §A.

Formally, we derive:

$$\text{RCD-PROJ-I}\ \frac{\mathscr{E}'[r \triangleright \mathtt{point}] \qquad \emptyset \vdash \mathscr{E}'[r.\mathtt{point}.\mathtt{x}] : \mathtt{point} \to \mathtt{int} * \mathtt{int}}{\emptyset \vdash \underbrace{\mathscr{E}'[r.\mathtt{x}]} : \mathtt{point} \to \mathtt{int} * \mathtt{int}}$$

$$\frac{\mathscr{E}[(r : \mathtt{point}) \triangleright \mathtt{point}] \qquad \emptyset \vdash \overbrace{\mathscr{E}[r.\mathtt{point}.\mathtt{y}]} : \mathtt{point} \to \mathtt{int} * \mathtt{int}}{\emptyset \vdash \mathscr{E}[(r : \mathtt{point}).\mathtt{y}] : \mathtt{point} \to \mathtt{int} * \mathtt{int}}\ \text{RCD-PROJ-I}$$

where the contexts $\mathscr{E}$ and $\mathscr{E}'$ are:

$$\begin{aligned} \mathscr{E} &\triangleq \lambda r.\,(r.\mathtt{x}, \square) \\ \mathscr{E}' &\triangleq \lambda r.\,(\square, r.\mathtt{point}.\mathtt{y}) \end{aligned}$$

We have $\emptyset \vdash \mathscr{E}'[r.\mathtt{point}.\mathtt{x}] : \mathtt{point} \to \mathtt{int} * \mathtt{int}$, indeed. It therefore remains to prove $\mathscr{E}[(r : \mathtt{point}) \triangleright \mathtt{point}]$ (**1**) and $\mathscr{E}'[r \triangleright \mathtt{point}]$ (**2**).

For (1), we use the fact that unicity is *composable*: if $\mathscr{E}_1[e \triangleright \varsigma]$, then $\mathscr{E}_2[\mathscr{E}_1][e \triangleright \varsigma]$. Since $\square[(r : \mathtt{point}) \triangleright \mathtt{point}]$ holds trivially, we have $\mathscr{E}[(r : \mathtt{point}) \triangleright \mathtt{point}]$.

For (2), assume $\emptyset \vdash \mathscr{E}'[\square \text{ with } (r : \mathfrak{g})] : \tau$. Since $r.\mathtt{point}.\mathtt{y}$ requires r to have the type point (due to RCD-PROJ-X), it follows that there is no other choice but to take point for $\mathfrak{g}$, which proves (2).

Example 3.4. To illustrate a simple case of non-typability, we reconsider the example $\mathtt{ex}_1 \triangleq \lambda r.\, r.\mathtt{x}$ from §2.1. If there is a derivation of $\mathtt{ex}_1$, then there must be one of the form:

$$\frac{\mathscr{E}[r \triangleright \nu\bar{\gamma}.\,\mathtt{rcd}\ T\ \bar{\gamma}] \qquad \emptyset \vdash \mathscr{E}[r.T.\mathtt{x}] : \tau}{\emptyset \vdash \mathscr{E}[r.\mathtt{x}] : \tau}\ \text{RCD-PROJ-I}$$

where $\mathscr{E}$ is the term $\lambda r.\,\square$, which is the largest possible context. Unfortunately, $\mathscr{E}[r \triangleright \nu\bar{\gamma}.\,\mathtt{rcd}\ T\ \bar{\gamma}]$ does not hold for any T. Indeed, we have $\emptyset \vdash \mathscr{E}[\square \text{ with } (r : \mathfrak{g})] : \mathfrak{g} \to \tau$. for any $\mathfrak{g}$ and τ. Hence, point and gray_point are both possible shapes for the type of r.

Example 3.5. Considering the example from §2.5.

```
type gray_point = { x : int; y : int; color : int }
type cie_color  = { x : int; y : int; z : int }
type cie_point  = { x : int; y : int; color : cie_color }
let ex₁₀ r = r.color.x                                    OCaml ✗  OmniML ✗
```

As explained in §2.5, $\mathtt{ex}_{10}$ is ill-typed because our unicity conditions enforce that each implicit field projection must be resolved individually and in a sequential fashion. We view this as a "Goldilocks" solution: we deliberately trade a small loss in expressivity for the benefit of a tractable, backtracking-free inference algorithm.

To type $\mathtt{ex}_{10}$ one must eliminate the final implicit projection in a context of the form $\mathscr{E}[e.\ell]$. Neither projection can be resolved by applying RCD-PROJ-CLOSED since the labels (color and x) are both overloaded. Thus, the implicit projections must be typed using RCD-PROJ-I. Two cases arise, neither of which are possible:

Case $\mathscr{E}$ *is* $\lambda r.\,\square$. We should have a derivation that ends with

$$\frac{\mathscr{E}[r.\mathtt{color} \triangleright \mathtt{cie\_color}] \qquad \emptyset \vdash \mathscr{E}[r.\mathtt{color}.\mathtt{cie\_color}.\mathtt{x}] : \tau}{\emptyset \vdash \mathscr{E}[r.\mathtt{color}.\mathtt{x}] : \tau}\ \text{RCD-PROJ-I}$$

However, $\mathscr{E}[r.\mathtt{color} \triangleright \mathtt{cie\_color}]$ does not hold. Indeed, the judgment $\emptyset \vdash \lfloor \mathscr{E}[(\square \text{ with } r.\mathtt{color} : \mathfrak{g})] \rfloor : \mathfrak{g} \to \tau'$ (*i.e.*, $\emptyset \vdash \lambda r.\,((\square \text{ with } r) : \mathfrak{g}) : \mathfrak{g} \to \tau'$) holds for any $\mathfrak{g}$. Hence, the shape of the type of $r.\mathtt{color}.\mathtt{x}$ is not uniquely determined and this case cannot occur.

$$
\begin{array}{rcll}
C & ::= & \mathsf{true} \mid \mathsf{false} \mid C_1 \wedge C_2 \mid \tau_1 = \tau_2 \mid \exists \alpha.\, C \mid \forall \alpha.\, C & \text{Constraints} \\
& \mid & \mathsf{let}\ x = \lambda \alpha.\, C_1\ \mathsf{in}\ C_2 \mid x\ \tau \mid \mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi} & \\
\chi & ::= & \rho \Rightarrow C & \text{Branches} \\
\rho & ::= & \_ \mid \ldots & \text{Shape patterns} \\
\mathscr{C} & ::= & \square \mid \mathscr{C} \wedge C \mid C \wedge \mathscr{C} \mid \exists \alpha.\, \mathscr{C} \mid \forall \alpha.\, \mathscr{C} & \text{Constraint contexts} \\
& \mid & \mathsf{let}\ x = \lambda \alpha.\, \mathscr{C}\ \mathsf{in}\ C \mid \mathsf{let}\ x = \lambda \alpha.\, C\ \mathsf{in}\ \mathscr{C} & \\
\phi & ::= & \emptyset \mid \phi[\alpha := \mathfrak{g}] \mid \phi[x := \mathfrak{S}] & \text{Semantic environments} \\
\mathfrak{g} & \in & \mathcal{G} & \text{Ground types} \\
\mathfrak{S} & \subseteq & \mathcal{G} & \text{Sets of ground types}
\end{array}
$$

$$
\textsc{True}\ \frac{}{\phi \vdash \mathsf{true}}
\qquad
\textsc{Conj}\ \frac{\phi \vdash C_1 \qquad \phi \vdash C_2}{\phi \vdash C_1 \wedge C_2}
\qquad
\textsc{Unif}\ \frac{\phi(\tau_1) = \phi(\tau_2)}{\phi \vdash \tau_1 = \tau_2}
\qquad
\textsc{Exists}\ \frac{\phi[\alpha := \mathfrak{g}] \vdash C}{\phi \vdash \exists \alpha.\, C}
\qquad
\textsc{Forall}\ \frac{\forall \mathfrak{g},\ \phi[\alpha := \mathfrak{g}] \vdash C}{\phi \vdash \forall \alpha.\, C}
$$

$$
\textsc{Let}\ \frac{\mathfrak{S} = \phi(\lambda \alpha.\, C_1) \qquad \mathfrak{S} \neq \emptyset \qquad \phi[x := \mathfrak{S}] \vdash C_2}{\phi \vdash \mathsf{let}\ x = \lambda \alpha.\, C_1\ \mathsf{in}\ C_2}
\qquad
\textsc{App}\ \frac{\phi(\tau) \in \phi(x)}{\phi \vdash x\ \tau}
$$

$$
\begin{array}{rcl}
\phi(\lambda \alpha.\, C) & \triangleq & \{\mathfrak{g} \in \mathcal{G} : \phi[\alpha := \mathfrak{g}] \vdash C\} \\
C_1 \vDash C_2 & \triangleq & \forall \phi,\ \phi \vdash C_1 \implies \phi \vdash C_2 \\
C_1 \equiv C_2 & \triangleq & (C_1 \vDash C_2)\ \wedge\ (C_1 \vDash C_2)
\end{array}
$$

$$
\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\rho} \Rightarrow \bar{C} \triangleq
\begin{cases}
\exists \bar{\alpha}.\, \tau = \varsigma\ \bar{\alpha} \wedge \theta(C_i) & \text{if } \rho_i \text{ matches } \varsigma\ \bar{\alpha} \hookrightarrow \theta \\
\mathsf{false} & \text{otherwise}
\end{cases}
$$

$$
\boxed{\rho \text{ matches } \varsigma\ \bar{\gamma} \hookrightarrow \theta}
\qquad
\_ \text{ matches } \varsigma\ \bar{\gamma} \hookrightarrow \emptyset
\qquad \vdots
$$

$$
\textsc{Match-Ctx}\ \frac{\mathscr{C}[\tau\,!\,\varsigma] \qquad \phi \vdash \mathscr{C}[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}]}{\phi \vdash \mathscr{C}[\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}]}
\qquad
\begin{array}{l}
\mathscr{C}[\tau\,!\,\varsigma] \triangleq \\
\quad \forall \phi, \mathfrak{g}.\ \phi \vdash \lfloor \mathscr{C}[\tau = \mathfrak{g}] \rfloor \implies \mathsf{shape}(\mathfrak{g}) = \varsigma
\end{array}
$$

Fig. 7. Selected syntax and semantics of constraints.

Case $\mathscr{E}$ *is* $\lambda r.\, \square.\mathsf{x}$. The derivation must end with:

$$
\frac{\mathscr{E}[r \triangleright \mathtt{cie\_point}] \qquad \emptyset \vdash \mathscr{E}[r.\mathtt{cie\_point.y}] : \tau}{\emptyset \vdash \mathscr{E}[r.\mathtt{color}] : \tau}\ \textsc{Rcd-Proj-I}
$$

However, $\mathscr{E}[r \triangleright \mathtt{cie\_point}]$ does not hold either. Again, the judgment $\emptyset \vdash \lfloor \mathscr{E}[((\square\ \mathsf{with}\ r) : \mathfrak{g})] \rfloor : \tau' \to \mathfrak{g}$, (*i.e.*, $\emptyset \vdash \lambda r.\, ((\square\ \mathsf{with}\ r) : \mathfrak{g}) : \tau' \to \mathfrak{g}$) holds for any $\mathfrak{g}$.

4 Constraints

To reason about constraint-based inference, we need more than a procedure for generating and solving constraints: we require a *formal logic* of constraints, with a syntax and a declarative semantics that characterizes satisfiability. This semantics is essential: it validates the design of our constraint language and provides the foundation for proving the soundness, completeness, and principality of inference. Without it, the meta-theory of our approach cannot be stated precisely.

We now introduce the syntax and semantics of our constraint language. Building atop the constraint-based inference framework of Pottier and Rémy [2005], we adopt a constraint language (Figure 7) that includes both term and type variables. Its semantics is given by a satisfiability judgment $\phi \vdash C$ (Figure 7). The semantic environment ϕ assigns to each free type variable α a ground type $\mathfrak{g} \in \mathcal{G}$ (a type with no free variables) and to each term variable x a set of ground types[13] $\mathfrak{S} \subseteq \mathcal{G}$ (the instances of a type scheme bound to x). We write $\phi[\alpha := \mathfrak{g}]$ and $\phi[x := \mathfrak{S}]$ for extensions of ϕ, and $\phi(\tau)$ for the ground type obtained by substitution.

Constraints include basic logical forms: tautological true (True), unsatisfiable false, and conjunctive $C_1 \wedge C_2$ (Conj) constraints. The unification constraint $\tau_1 = \tau_2$ is satisfied when τ_1 and τ_2 are equal (Unif). An existential constraint $\exists\alpha. C$ holds if there exists a witness $\mathfrak{g}$ for α satisfying C (Exists), while a universal constraint $\forall\alpha. C$ holds if C is satisfied for every binding of α (Forall). When σ is a polymorphic type scheme $\forall\bar{\alpha}. \tau'$, we use the notation $\sigma \leq \tau$ as shorthand for the instantiation constraint $\exists\bar{\alpha}. \tau' = \tau$. OmniML-specific constraints, such as record label instantiation $t.\ell \leq \tau_1 \to \tau_2$ (§2.4), are introduced later (§4.2).

Polymorphism in the constraint language is expressed through *generalization* and *instantiation* constraints. In a generalization constraint (or *let*-constraint) $\mathsf{let}\ x = \lambda\alpha. C_1\ \mathsf{in}\ C_2$, the definition of x is a constraint abstraction $\lambda\alpha. C_1$, a function that, when applied to a type τ, returns $C_1[\alpha := \tau]$. The binding of x is then available in C_2 and refers to this abstraction. Semantically, the abstraction $\lambda\alpha. C_1$ is interpreted as a set of ground types that satisfies C_1, and the generalization constraint requires this set to be non-empty, *i.e.*, there is at least one instantiation of α that satisfies C_1. Instantiations (or applications) $x\ \tau$ eliminate abstractions by applying a type τ to the abstraction bound to x. This holds precisely when $\phi(\tau) \in \phi(x)$, *i.e.*, τ is one of the satisfiable instances of x. The λ-abstraction and application syntax follows Pottier [2014]. In other presentations, constraint abstractions $\lambda\alpha. \exists\bar{\beta}. C$ are written as constrained type schemes $\forall\alpha, \bar{\beta}. C \Rightarrow \alpha$, and instantiation constraints $x\ \tau$ are written $x \leq \tau$.

Finally, we introduce *suspended match constraints* ($\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}$), which consist of:

(1) A matchee τ. The constraint remains suspended until the *shape* of τ is determined, *i.e.*, while τ is a type variable.
(2) A list of branches $\bar{\chi}$ of the form $\rho \Rightarrow C$, where ρ is a shape pattern.[14] The constraint C is solved in the extended context produced by the matching pattern.
For example, the wildcard pattern _ matches any shape, and binds nothing. To ensure determinism, the set of patterns $\bar{\rho}$ must be *disjoint*—that is, no shape may be matched by more than one pattern in the list.

The formal semantics of suspended match constraints are somewhat involved; we return to them in the next subsection (§4.1).

Closed constraints are either satisfiable in any semantic environment (*i.e.*, they are tautologies) or unsatisfiable. For example, the satisfiability of the constraint $\exists\alpha. \alpha = \mathsf{int}$ is established by the derivation on the right-hand side.

$$\dfrac{\dfrac{\mathsf{int} = \mathsf{int}}{\phi[\alpha := \mathsf{int}] \vdash \alpha = \mathsf{int}}\ \textsc{Unif}}{\phi \vdash \exists\alpha. \alpha = \mathsf{int}}\ \textsc{Exists}$$

We write $C_1 \vDash C_2$ to express that C_1 *entails* C_2, meaning every solution ϕ to C_1 is also a solution to C_2. We write $C_1 \equiv C_2$ to indicate that C_1 and C_2 are equivalent, that is, they have exactly the same set of solutions.

Throughout this paper, we will find it convenient to work with *constraint contexts*. A constraint context $\mathscr{C}$ is simply a constraint with a *hole*, analogous to term contexts $\mathscr{E}$ introduced in §3. We write $\mathscr{C}[C]$ to denote filling the hole of the context $\mathscr{C}$ with the constraint C. Hole filling may capture variables, so we frequently require explicit side conditions when variable capture must be avoided. We write $\mathsf{bv}(\mathscr{C})$ for the set of variables bound at the hole in $\mathscr{C}$.

[13] $\mathfrak{S}$ can be pronounced "Fraktur S" or "ground S".
[14] The match constraints considered in this paper only use a single branch. In a richer language, however, multiple branches would be useful; for instance, record projection could be overloaded to operate on nominal records as well as tuples (or objects, modules, *etc.*), requiring several branches in the generated constraint. We therefore kept the general syntax.

4.1 Suspended constraints

A central difficulty in our work on suspended constraints was defining a satisfying semantics. The challenge lies in formalizing what it means for type information to be *known* without presupposing a *static* solving order. This is the same issue encountered in our typing rules (§3.4), and we address it in the same way: by introducing a contextual rule together with a unicity condition.

To define the semantics for suspended constraints, we first introduce *discharged match constraints.*

Definition 4.1 (Discharged match constraint). Given a suspended constraint (match τ with $\bar{\chi}$) and a canonical shape ς, we introduce the syntactic sugar (match $\tau := \varsigma$ with $\bar{\chi}$) for the *discharged match constraint* that selects the branch in $\bar{\chi}$ that matches ς:

$$\text{match } \tau := \varsigma \text{ with } \overline{\rho \Rightarrow C} \;\triangleq\; \begin{cases} \exists \bar{\alpha}.\, \tau = \varsigma\, \bar{\alpha} \wedge \theta(C_i) & \text{if } \rho_i \text{ matches } \varsigma\ \bar{\alpha} \hookrightarrow \theta \\ \text{false} & \text{otherwise} \end{cases}$$

The first conjunct ($\tau = \varsigma\, \bar{\alpha}$) ensures that ς is indeed the canonical shape of τ, and the second conjunct is the selected branch constraint C_i under the appropriate substitution. Since the syntax of suspended match constraints requires that branch patterns are non-overlapping, the matching branch $\rho_i \Rightarrow C_i$ is uniquely determined. It may, however, be undefined if the branches are not exhaustive, in which case the discharged constraint is false.

The partial function (ρ matches $\varsigma\ \bar{\alpha}$), introduced in Figure 7, describes how a pattern ρ is matched against a canonical principal shape ς. Before matching, ς is applied with fresh shape variables $\bar{\alpha}$ (of the same arity as ς), so that the result of matching may refer to them. The match either fails or returns a substitution θ mapping pattern variables (*e.g.* record name variables t from §2.4) to shape components (*e.g.* record names T). These components may themselves mention the freshly introduced variables $\bar{\alpha}$. In Figure 7, we only introduce the wildcard pattern and its matching rule; in §4.2 we extend both the syntax of shape patterns and the matching function itself to handle the patterns that arise in OmniML.

A natural attempt. As with typing rules for fragile constructs (*e.g.* Rcd-Proj-I-Nat), suspended match constraints have a *natural rule*:

$$\textsc{Match-Nat}\quad \frac{\varsigma = \text{shape}(\phi(\tau)) \qquad \phi \vdash \text{match } \tau := \varsigma \text{ with } \bar{\chi}}{\phi \vdash \text{match } \tau \text{ with } \bar{\chi}}$$

This rule states that a suspended constraint holds whenever the corresponding discharged constraint holds for the canonical shape of $\phi(\tau)$.

Although simple and declarative, this semantics is too permissive. It allows *guessing* the shape of τ rather than requiring it to be *known*. For example, $\exists \alpha.$ match α with $\_ \Rightarrow \alpha = \text{int}$ is satisfiable under the natural semantics:

$$\dfrac{\dfrac{\_ \text{ matches int } \emptyset \hookrightarrow \emptyset \qquad \dfrac{\text{int} = \text{int}}{\phi[\alpha := \text{int}] \vdash \alpha = \text{int}}\,\textsc{Unif}}{\phi[\alpha := \text{int}] \vdash \text{match } \alpha \text{ with } \_ \Rightarrow \alpha = \text{int}}\,\textsc{Match-Nat}}{\phi \vdash \exists \alpha.\, \text{match } \alpha \text{ with } \_ \Rightarrow \alpha = \text{int}}\,\textsc{Exists}$$

This "out-of-thin-air" behavior does not match the intended meaning of suspended match constraints and raises several problems: (1) a reasonable solver—one that avoids backtracking—cannot be complete with respect to this semantics; and (2) it breaks the existence of principal solutions, just as the *natural* typing rules do (*e.g.* Rcd-Proj-I-Nat).

Contextual semantics. To rule out guessing, we instead adopt a *contextual* semantics: a match constraint is satisfiable only if the shape of the type is determined by the surrounding context. The corresponding rule for suspended constraints, Match-Ctx (Figure 7), is the only non-syntax-directed rule in our semantics.

$$\textsc{Match-Ctx}\quad \frac{\mathscr{C}[\tau \mathbin{!} \varsigma] \qquad \phi \vdash \mathscr{C}[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}]}{\phi \vdash \mathscr{C}[\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}]}$$

In this rule, a suspended constraint ($\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}$) in the context $\mathscr{C}$ can be discharged, provided the shape ς is not guessed from ϕ, but recovered from the constraint context $\mathscr{C}$. This *unicity* condition $\mathscr{C}[\tau \mathbin{!} \varsigma]$ (defined below) ensures that ς is uniquely determined by the context $\mathscr{C}$, capturing precisely what it means for the shape of a type to be *known*.

Definition 4.2 (Erasure). The erasure $\lfloor C \rfloor$ of a constraint C is defined as the constraint obtained by replacing suspended match constraints in C with true.

Definition 4.3 (Simple constraints). We say that C is *simple* if it contains no suspended match constraints.

Definition 4.4 (Unicity). We define the unicity condition $\mathscr{C}[\tau \mathbin{!} \varsigma]$, which states that τ has a unique canonical shape ς within the context $\mathscr{C}$ as: $\forall \phi, \mathfrak{g}.\ \phi \vdash \lfloor \mathscr{C}[\tau = \mathfrak{g}] \rfloor \implies \mathsf{shape}(\mathfrak{g}) = \varsigma$.

Similarly to the unicity conditions introduced for typing rules in §3.4, unicity for constraints $\mathscr{C}[\tau \mathbin{!} \varsigma]$ relies on *erasure* $\lfloor \mathscr{C}[\tau = \mathfrak{g}] \rfloor$ to restrict attention to previously discharged match constraints, inducing a causal order between match constraints, enforced by Match-Ctx.

When τ is not a variable, $\square[\tau \mathbin{!} \varsigma]$ holds trivially when ς is the shape of τ. Likewise, when (the erasure of) $\mathscr{C}$ is unsatisfiable, then $\mathscr{C}[\alpha \mathbin{!} \varsigma]$ holds vacuously for any ς. The interesting and nontrivial cases—the ones we illustrate next—arise when τ is a type variable and $\mathscr{C}$ is satisfiable.

Example 4.5. Recall the problematic example that was satisfiable under the natural semantics:

$$\exists \alpha.\ \mathsf{match}\ \alpha\ \mathsf{with}\ \_ \Rightarrow \alpha = \mathsf{int}$$

Under the contextual semantics, the suspended constraint appears in a context $\mathscr{C}$ with no contextual information. *i.e.*, $\mathscr{C}$ is $\exists \alpha. \square$. So for any ground type $\mathfrak{g}$, $\lfloor \mathscr{C}[\alpha = \mathfrak{g}] \rfloor$ (*i.e.*, $\exists \alpha.\ \alpha = \mathfrak{g}$) is satisfiable, allowing $\mathfrak{g}$ to have an arbitrary shape (*e.g.* int, bool, *etc.*). As a result, the uniqueness condition $\mathscr{C}[\alpha \mathbin{!} \varsigma]$ never holds making Match-Ctx inapplicable. The constraint is unsatisfiable as intended.

Example 4.6. Consider the satisfiable constraint:

$$\exists \alpha.\ \alpha = \mathsf{int} \land \mathsf{match}\ \alpha\ \mathsf{with}\ \_ \Rightarrow \mathsf{true}$$

Here, we apply the contextual rule with the context $\mathscr{C}$ equal to $\exists \alpha.\ \alpha = \mathsf{int} \land \square$. Any solution ϕ of this context necessarily satisfies $\alpha = \mathsf{int}$, so we have $\mathscr{C}[\alpha \mathbin{!} \mathsf{int}]$ and the suspended constraint can be discharged.

Example 4.7. Consider the more intricate example:

$$\exists \alpha, \beta.\ (\mathsf{match}\ \alpha\ \mathsf{with}\ \_ \Rightarrow \beta = \mathsf{bool}) \ \land\ (\mathsf{match}\ \beta\ \mathsf{with}\ \_ \Rightarrow \mathsf{true}) \ \land\ (\alpha = \mathsf{int})$$

Suppose we attempt to apply Match-Ctx to the match on β first. We want to show $\mathscr{C}[\beta \mathbin{!} \mathsf{bool}]$ for the context $\mathscr{C}$ equal to ($\mathsf{match}\ \alpha\ \mathsf{with}\ \_ \Rightarrow \beta = \mathsf{bool}$) $\land \square \land \alpha = \mathsf{int}$. Its erasure $\lfloor \mathscr{C} \rfloor$ is $\mathsf{true} \land \square \land \alpha = \mathsf{int}$, which imposes no constraints on β. Thus both $\lfloor \mathscr{C}[\beta = \mathsf{int}] \rfloor$ and $\lfloor \mathscr{C}[\beta = \mathsf{bool}] \rfloor$ are both satisfiable: unicity does not hold and Match-Ctx cannot be applied.

By contrast, if we first discharge the match on α, we consider the context $\mathscr{C}$ equal to $\square \land$ ($\mathsf{match}\ \beta\ \mathsf{with}\ \_ \Rightarrow \mathsf{true}$) $\land\ \alpha = \mathsf{int}$. Its erasure $\lfloor \mathscr{C} \rfloor$ equal to $\square \land \mathsf{true} \land \alpha = \mathsf{int}$ does constraint α, giving

$$
\begin{array}{lll}
t & & \text{Record variables} \\
s & & \text{Scheme variables} \\
\rho & ::= \ \ldots \mid \mathsf{rcd}\ t\ \_ \mid [s] & \text{Shape patterns} \\
C & ::= \ \ldots \mid t.\ell \le \tau_1 \to \tau_2 \mid \mathsf{dom}\ t = \bar{\ell} \mid s \le \tau \mid x \le s & \text{Constraints}
\end{array}
$$

$$
\begin{array}{rcl}
(\mathsf{rcd}\ t\ \_)\ \mathsf{matches}\ (\nu\bar{\gamma}.\,\mathsf{rcd}\ T\ \bar{\gamma})\ \bar{\gamma}' & \triangleq & [t := T] \\
{[s]}\ \mathsf{matches}\ (\nu\bar{\gamma}.\,[\sigma])\ \bar{\gamma}' & \triangleq & [s := \sigma[\bar{\gamma} := \bar{\gamma}']]
\end{array}
$$

$$
\begin{array}{rcl}
T.\ell \le \tau_1 \to \tau_2 & \triangleq & \begin{cases} \exists\bar{\alpha}.\,\tau_1 = \mathsf{rcd}\ T\ \bar{\alpha} \wedge \tau_2 = \tau & \text{if } \Omega(T.\ell) = \forall\bar{\alpha}.\,\mathsf{rcd}\ T\ \bar{\alpha} \to \tau \\ \mathsf{false} & \text{otherwise} \end{cases} \\
\mathsf{dom}\ T = \bar{\ell} & \triangleq & \begin{cases} \mathsf{true} & \text{if dom } (\Omega(T)) = \bar{\ell} \\ \mathsf{false} & \text{otherwise} \end{cases} \\
(\forall\bar{\alpha}.\,\tau') \le \tau & \triangleq & \exists\bar{\alpha}.\,\tau' = \tau \\
x \le (\forall\bar{\alpha}.\,\tau) & \triangleq & \forall\bar{\alpha}.\,x\ \tau
\end{array}
$$

Fig. 8. Patterns for OmniML.

$\mathscr{C}[\alpha\,!\,\mathsf{int}]$. We may therefore discharge the match on α, rewriting it as (match α := int with $\_ \Rightarrow \beta = \mathsf{bool}$) *i.e.*, $\alpha = \mathsf{int} \wedge \beta = \mathsf{bool}$. Substituting back, we are left to satisfy the constraint $\mathscr{C}[\alpha = \mathsf{int} \wedge \beta = \mathsf{bool}]$ *i.e.*, $\alpha = \mathsf{int} \wedge \beta = \mathsf{bool} \wedge (\mathsf{match}\ \beta\ \mathsf{with}\ \_ \Rightarrow \mathsf{true}) \wedge \alpha = \mathsf{int}$. At this point, unicity for β holds, since the context now includes $\beta = \mathsf{bool}$. We can therefore apply Match-Ctx to eliminate the final match constraint.

This example demonstrates that suspended constraints must be resolved in a dependency-respecting order: attempting to resolve a match constraint too early may result in unsatisfiability.

Example 4.8. Let us consider a constraint with a cyclic dependency between match constraints:

$$\exists\alpha,\beta.\,(\mathsf{match}\ \alpha\ \mathsf{with}\ \_ \Rightarrow \beta = \mathsf{bool}) \ \wedge\ (\mathsf{match}\ \beta\ \mathsf{with}\ \_ \Rightarrow \alpha = \mathsf{int})$$

Under the natural semantics this constraint is satisfiable, since one may *guess* the assignment $\alpha := \mathsf{int}, \beta := \mathsf{bool}$, making both match constraints succeed. However, our solver and contextual semantics reject it.

Without loss of generality, suppose we attempt to apply Match-Ctx on α first. We must establish $\mathscr{C}[\alpha\,!\,\mathsf{int}]$ for the context $\mathscr{C} := \square \wedge \mathsf{match}\ \beta\ \mathsf{with}\ \_ \Rightarrow \alpha = \mathsf{int}$. But the erasure $\lfloor\mathscr{C}\rfloor$ is $\square \wedge \mathsf{true}$, which imposes no constraint on α. Hence unicity fails and Match-Ctx is inapplicable.

4.2 Constraint generation

We now present the formal translation from terms e to constraints C, such that the resulting constraint is satisfiable if and only if the term is well typed. The translation is defined as a function $[\![e : \tau]\!]$, where e is the term to be translated and τ is the expected type of e. The expected type τ is permitted to contain type variables, which can be existentially bound in order to perform type inference. The models of constraint $[\![e : \tau]\!]$ interpret the free variables of τ such that τ becomes a valid type of e. For example, to infer the entire type of e we may pick a fresh type variable α for τ.

Shape patterns. Thus far, our formal presentation of shape patterns has remained abstract, deliberately leaving the syntax and semantics of match constraints partially unspecified to accommodate a range of language features. We now concretize this by specifying the shape patterns used in OmniML (see Figure 8), and introducing the corresponding constraints for the variables they bind. Shape patterns include:

$$
\begin{array}{lll}
[\![x : \tau]\!] & \triangleq & x\ \tau \\
[\![() : \tau]\!] & \triangleq & \tau = 1 \\
[\![\lambda x.\, e : \tau]\!] & \triangleq & \exists \alpha, \beta.\ \mathsf{let}\ x = \lambda \alpha'.\, \alpha' = \alpha\ \mathsf{in}\ [\![e : \beta]\!] \wedge \tau = \alpha \to \beta \\
[\![e_1\ e_2 : \tau]\!] & \triangleq & \exists \alpha, \beta.\ [\![e_1 : \beta]\!] \wedge [\![e_2 : \alpha]\!] \wedge \beta = \alpha \to \tau \\
[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau]\!] & \triangleq & \mathsf{let}\ x = \lambda \alpha.\, [\![e_1 : \alpha]\!]\ \mathsf{in}\ [\![e_2 : \tau]\!] \\
[\![(e : \exists \bar{\alpha}.\, \tau') : \tau]\!] & \triangleq & \exists \bar{\alpha}.\ [\![e : \tau']\!] \wedge \tau = \tau' \\
[\![[e : \exists \bar{\alpha}.\, \sigma] : \tau]\!] & \triangleq & \exists \bar{\alpha}.\ [\![e : \sigma]\!] \wedge \tau = [\sigma] \\
[\![\langle e : \exists \bar{\alpha}.\, \sigma \rangle : \tau]\!] & \triangleq & \exists \bar{\alpha}, \beta.\ [\![e : \beta]\!] \wedge \beta = [\sigma] \wedge \sigma \leq \tau \\
[\![\langle e \rangle : \tau]\!] & \triangleq & \exists \alpha.\ [\![e : \alpha]\!] \wedge \mathsf{match}\ \alpha\ \mathsf{with}\ [s] \Rightarrow s \leq \tau \\
[\![[e] : \tau]\!] & \triangleq & \mathsf{let}\ x = \lambda \alpha.\, [\![e : \alpha]\!]\ \mathsf{in}\ \mathsf{match}\ \tau\ \mathsf{with}\ [s] \Rightarrow x \leq s \\
[\![e.\ell : \tau]\!] & \triangleq & \begin{cases} [\![e.T.\ell : \tau]\!] & \text{if } \ell \triangleright T \\ \exists \alpha.\ [\![e : \alpha]\!] \wedge \mathsf{match}\ \alpha\ \mathsf{with}\ \mathsf{rcd}\ t\ \_ \Rightarrow t.\ell \leq \alpha \to \tau & \text{otherwise} \end{cases} \\
[\![e.T.\ell : \tau]\!] & \triangleq & \exists \alpha.\ [\![e : \alpha]\!] \wedge T.\ell \leq \alpha \to \tau \\
[\![\{\overline{\ell = e}\} : \tau]\!] & \triangleq & \begin{cases} [\![T.\{\overline{\ell = e}\} : \tau]\!] & \text{if } \bar{\ell} \blacktriangleright T \\ \exists \bar{\alpha}.\ \bigwedge_{i=1}^{n} [\![e_i : \alpha_i]\!]\ \wedge & \text{otherwise} \\ \qquad \mathsf{match}\ \tau\ \mathsf{with}\ \mathsf{rcd}\ t\ \_ \Rightarrow (\mathsf{dom}\ t = \bar{\ell} \wedge \bigwedge_{i=1}^{n} t.\ell_i \leq \tau \to \alpha_i) & \end{cases} \\
[\![T.\{\overline{\ell = e}\} : \tau]\!] & \triangleq & \exists \bar{\alpha}.\ \bigwedge_{i=1}^{n} [\![e_i : \alpha_i]\!] \wedge \mathsf{dom}\ T = \bar{\ell} \wedge \bigwedge_{i=1}^{n} T.\ell_i \leq \tau \to \alpha_i \\
[\![\square\ \mathsf{with}\ \bar{e} : \tau]\!] & \triangleq & \exists \bar{\alpha}.\ \bigwedge_{i=1}^{n} [\![e_i : \alpha_i]\!] \\
\\
[\![e : \forall \bar{\alpha}.\, \tau]\!] & \triangleq & \forall \bar{\alpha}.\ [\![e : \tau]\!]
\end{array}
$$

Fig. 9. The constraint generation translation for OmniML.

(1) Record type patterns rcd t _ which match record types rcd $T\ \bar{\tau}$ and bind the record variable t to the name T. The parameters $\bar{\tau}$ are not bound, since record field overloading depends only on the name T.
(2) Polytype patterns $[s]$, which match polytypes $[\sigma]$ and bind the scheme variable s to σ.

Each new kind of pattern introduces corresponding constraint formers: $t.\ell \leq \tau_1 \to \tau_2$ asserts that $\tau_1 \to \tau_2$ is an instance of projection type associated with the explicit label $t.\ell$; dom $t = \bar{\ell}$ ensures that the domain of record type t is the set of labels $\bar{\ell}$; and $s \leq \tau$ checks that τ is an instance of s, while $x \leq s$ asserts that every instance of s is an instance of x.

By definition, each of these constraint forms is unsatisfiable, since we do not extend the satisfiability relation to include them. Their role is purely syntactic: they only appear within the branches of a suspended match constraint. Once such a constraint is discharged (match $\tau := \varsigma$ with $\bar{\rho} \Rightarrow \bar{C}$) (Definition 4.1), the substitution θ produced by the matching pattern ρ_i—which binds the pattern variables (*e.g.* t, s, *etc.*)—is applied to the corresponding branch C_i. Hence, it suffices to define the semantics of these constraint formers only for their substituted forms (*e.g.* $T.\ell \leq \tau_1 \to \tau_2$, dom $T = \bar{\ell}$, *etc.*), as shown in Figure 8. We define the semantics of these substituted forms by translation into existing constraints—that is, as syntactic sugar over the existing core constraint language.

Constraint generation. Constraint generation $[\![e : \tau]\!]$ is defined in Figure 9. All generated type variables are fresh with respect to the expected type τ, ensuring capture-avoidance. We now review the cases of the constraint generator, beginning with the cases corresponding to traditional ML constructs.

Unsurprisingly, variables x generate an instantiation constraint $x\ \tau$. The unit term () requires τ to be the unit type 1. A function generates a constraint that binds two fresh flexible type variables for the parameter and return types. We use a let-constraint to bind the parameter in the constraint generated for the body of the function. The let-constraint is monomorphic since α' is fully constrained by type variables defined outside the abstraction's scope and therefore cannot be generalized. Applications introduce two fresh flexible type variables, one for the argument type and one for the type of the function, typing each subterm with these, ensuring τ is the expected return type. Let-bindings generate a polymorphic let-constraint; $\lambda\alpha.\, [\![e : \alpha]\!]$ is a principal constraint abstraction for e: its intended interpretation is the set of all types that e admits. Annotations bind their flexible type variables and enforce the equality of the annotated type τ' and the expected type τ.

Explicit polytype boxing asserts that e has the polymorphic type σ (using universal quantification) and that the expected type is the polytype $[\sigma]$. Conversely, explicit unboxing requires that τ be an instance of σ. These cases introduce no suspended match constrains because the annotations contain the required type information: the polytype $[\sigma]$.

By contrast, implicit unboxing suspends until the inferred type of e is known to be some non-variable type τ_0. At that point, the suspended match constraint attempts to match τ_0 against the pattern $[s]$. If $\tau_0 = [\sigma]$, the match succeeds, binding s to σ and requiring τ to be an instance of s (*i.e.*, $\sigma \leq \tau$); otherwise, the match fails and the term is ill-typed.

Implicit boxing behaves dually: we infer the principal type for e using a let-constraint and suspend until the expected type of the entire term resolves to some type τ_0. If τ_0 matches the same pattern $[s]$ (*i.e.*, $\tau_0 = [\sigma]$), we assert that the principal type of e is at least as general as s, via the constraint $x \leq s$ (*i.e.*, $x \leq \sigma$); otherwise, the match fails.

Record constructs are handled in a similar way, but their matching patterns concern record shapes rather than polytypes. A record projection generate a fresh variable α for the record type and constrain e to this type, suspending until the type of e (α) resolves to some type τ_0. When $\tau_0 = \mathsf{rcd}\ T\ \bar{\tau}$, the shape pattern $(\mathsf{rcd}\ t\ \_)$ matches successfully, binding t to the record name T; the matching branch then retrieves the projected label's type from the global label context Ω and instantiates it to match $\alpha \to \tau$. If τ_0 is not a record type, the match fails and the term is ill-typed.

For record expressions, we generate a fresh variable α_i for each field assignment to capture the type of each e_i as α_i. The rest of the constraint is deferred until the context determines the type of the whole record type to be τ_0. Once known, it is matched against the same pattern $(\mathsf{rcd}\ t\ \_)$. If the match succeeds (*i.e.*, $\tau_0 = \mathsf{rcd}\ T\ \bar{\tau}$ and $t = T$) the labels are instantiated to match the projection types $\tau \to \alpha_i$, and we additionally check that the domain of t is exactly $\bar{\ell}$, ensuring that every label is defined. Otherwise, the constraint is unsatisfiable as intended.

Explicit records and projections, along with closed-world disambiguated terms, bypass suspension and directly instantiate the appropriate labels.

Example 4.9. Consider the following example:

```
let ex₄ r = let x = r.x in x + (r : point).y          OCaml ✗  OmniML ✓
```

The typing constraint generated for ex_4 contains the following, where α stands for the type of r:

$$\exists\alpha.\ \mathsf{let}\ x = \lambda\beta.\ (\mathsf{match}\ \alpha\ \mathsf{with}\ \ldots)\ \mathsf{in}\ x\ \mathsf{int} \wedge \alpha = \mathtt{point}$$

The suspended constraint can be discharged under our contextual semantics. We apply the Match-Ctx rule with context $\mathscr{C}$ equal to $\mathsf{let}\ x = \lambda\beta.\, \square\ \mathsf{in}\ x\ \mathsf{int} \wedge \alpha = \mathtt{point}$. Although the context includes a let-binding—which in practice involves let-generalization—we can still deduce $\mathscr{C}[\alpha\, !\, \mathtt{point}]$, since the erased context $\lfloor \mathscr{C} \rfloor$ contains the unification constraint $\alpha = \mathtt{point}$.

This example illustrates that our formulation of suspended constraints interacts nicely with let-polymorphism. Although the two features are specified in a modular fashion, they are carefully crafted to work together, as we further show in our next example.

Example 4.10. A subtle yet crucial feature of our semantics is its support for *backpropagation*:

```
let ex11 = let getx r = r.x in getx one                OCaml ✗  OmniML ✓
```

As in the previous example, the type of `r` cannot be disambiguated from the let-definition alone. There, the ambiguity was resolved when the type (α) was unified to a known type (`point`) in the let-body. Here, the situation is more subtle: an *instance* of `getx`'s type scheme is taken, which only satisfies the application (`getx one`) if `r` has either a variable type or the record type `point`. However, the projection `r.x` would be ill-typed if `r` had a variable type (unicity would fail), so `point` is the unique consistent solution. This information therefore flows back from the instances in the let-body to the let-definition, a phenomenon we call *backpropagation.*

The constraint generated when typing $\texttt{ex}_{11}$ is:

$$\exists\alpha.\,\mathsf{let}\ getx = \lambda\delta.\,\exists\beta,\gamma.\ \big(\delta = \beta \to \gamma \wedge \mathsf{match}\ \beta\ \mathsf{with}\ \ldots\big)\ \mathsf{in}\ getx\ (\texttt{point} \to \alpha)$$

With the context $\mathscr{C}$ equal to $\mathsf{let}\ getx = \lambda\delta.\,\exists\beta,\gamma.\,\delta = \beta \to \gamma \wedge \square\ \mathsf{in}\ getx\ (\texttt{point} \to \alpha)$, we can show that the unicity predicate $\mathscr{C}[\beta\,!\,\texttt{point}]$ holds. For any $\mathfrak{g}$, the erasure $\lfloor \mathscr{C}[\beta = \mathfrak{g}] \rfloor$ is $\mathsf{let}\ getx = \lambda\delta.\,\exists\beta,\gamma.\,\delta = \beta \to \gamma \wedge \beta = \mathfrak{g}\ \mathsf{in}\ getx\ (\texttt{point} \to \alpha)$. Since $getx$ is bound to the constraint abstraction $\lambda\delta.\,\exists\gamma.\ \delta = (\mathfrak{g} \to \gamma)$, the instantiation $getx\ (\texttt{point} \to \alpha)$ can only be satisfied when $\mathfrak{g}$ is equal to `point`. This proves unicity, hence the generated constraint for $\texttt{ex}_{11}$ is satisfiable.

4.3 Metatheory

Constraint generation is sound and complete with respect to the typing judgment:

Theorem 4.11 (Constraint generation is sound and complete). *A closed OmniML term e is typable if and only if the constraint $\exists\alpha.\,[\![e : \alpha]\!]$ is satisfiable.*

Theorem 4.12 (Principal types). *For any well-typed closed OmniML term e, there exists a type τ such that: (i) $\vdash e : \tau$. (ii) For any other typing $\vdash e : \tau'$, then $\tau' = \theta(\tau)$ for some substitution θ.*

5 Constraint solving

We now present a machine for solving constraints in our language. The solver operates as a rewriting system on constraints $C \longrightarrow C'$. Once no further transitions are applicable, *i.e.*, $C \not\longrightarrow$, the constraint C is either a solved form—from which we can read off a most general solution—or unsatisfiable (if the constraint C is closed).

Conjunction notation. Our rewriting rules frequently manipulate collections of constraints. We write $\bar{C}$ for a (possibly empty) sequence of constraints interpreted as their conjunction $\bigwedge_i C_i$. The empty sequence denotes true.

5.1 Unification

Our constraints ultimately reduce to equations between types, which we solve using first-order unification. Like our solver, we specify unification as a non-deterministic rewriting relation between *unification problems* $U_1 \longrightarrow U_2$, that eventually reduces to a solved form $\hat{U}$ or to false.

Unification problems U (Figure 10a) are a restricted subset of constraints, extended with *multi-equations* [Pottier and Rémy 2005]—a multi-set of types considered equal. These generalize binary equalities: ϕ satisfies a multi-equation ϵ if all of its members are mapped to a single ground type $\mathfrak{g}$ (Multi-Unif). Multi-equations are considered equal modulo permutation of their members.

$$\begin{array}{llll} U & ::= & \mathsf{true} \mid \mathsf{false} \mid U_1 \wedge U_2 \mid \exists \alpha.\, U \mid \epsilon & \text{Unification problems} \\ \epsilon & ::= & \emptyset \mid \tau = \epsilon & \text{Multi-equations} \\ C & ::= & \ldots \mid \epsilon & \text{Constraints} \\ \mathscr{U} & ::= & \square \mid \mathscr{U} \wedge U_2 \mid U_1 \wedge \mathscr{U} \mid \exists \alpha.\, \mathscr{U} & \text{Unification context} \end{array}$$

$$\textsc{Multi-Unif}\quad \frac{\forall \tau \in \epsilon,\ \phi(\tau) = \mathfrak{g}}{\phi \vdash \epsilon}$$

(a) Syntax and semantics of unification problems.

$$\textsc{U-Exists}\quad \frac{(\exists \alpha.\, U_1) \wedge U_2 \qquad \alpha \# U_2}{\exists \alpha.\, U_1 \wedge U_2} \qquad \textsc{U-Cycle}\quad \frac{U \qquad \mathsf{cyclic}(U)}{\mathsf{false}} \qquad \textsc{U-True}\quad \frac{U \wedge \mathsf{true}}{U} \qquad \textsc{U-False}\quad \frac{\mathscr{U}[\mathsf{false}] \qquad \mathscr{U} \neq \square}{\mathsf{false}}$$

$$\textsc{U-Merge}\quad \frac{\alpha = \epsilon_1 \wedge \alpha = \epsilon_2}{\alpha = \epsilon_1 = \epsilon_2} \qquad \textsc{U-Stutter}\quad \frac{\alpha = \alpha = \epsilon}{\alpha = \epsilon} \qquad \textsc{U-Name}\quad \frac{\varsigma\,(\bar{\tau}, \tau_i, \bar{\tau}') = \epsilon \qquad \alpha \# \bar{\tau}, \bar{\tau}', \epsilon \qquad \tau_i \notin \mathcal{V}}{\exists \alpha.\, \alpha = \tau_i \wedge \varsigma\,(\bar{\tau}, \alpha, \bar{\tau}') = \epsilon} \qquad \textsc{U-Decomp}\quad \frac{\varsigma\, \bar{\alpha} = \varsigma\, \bar{\beta} = \epsilon}{\varsigma\, \bar{\alpha} = \epsilon \wedge \bar{\alpha} = \bar{\beta}}$$

$$\textsc{U-Clash}\quad \frac{\varsigma\, \bar{\alpha} = \varsigma'\, \bar{\beta} = \epsilon \qquad \varsigma \neq \varsigma'}{\mathsf{false}} \qquad \textsc{U-Trivial}\quad \frac{\epsilon \qquad |\epsilon| \leq 1}{\mathsf{true}}$$

(b) Unification algorithm as a series of rewriting rules $U_1 \longrightarrow U_2$. All shapes are principal.

Fig. 10. Unification: syntax, semantics, and the rewriting-based solver $U_1 \longrightarrow U_2$.

The unification rules are listed in Figure 10b. Rewriting proceeds under an arbitrary context $\mathscr{U}$, modulo α-equivalence and associativity and commutativity of conjunctions. Our algorithm is largely standard [Pottier and Rémy 2005], with its main novelty being the use of *canonical principal shapes* in place of type constructors. This uniform treatment of monotypes and polytypes simplifies unification and improves on the previous treatment of polytype unification [Garrigue and Rémy 1999].

We briefly summarize the role of each rule. U-Exists lifts existential quantifiers, enabling applications of U-Merge and U-Cycle since all multi-equations eventually become part of a single conjunction. U-Merge combines multi-equations sharing a common variable and U-Stutter removes duplicate variables. U-Decomp decomposes equal types with matching shapes into equalities between their subcomponents, while U-Clash detects shape mismatches that result in failure. U-Name introduces fresh variables for subcomponents, ensuring unification operates on *shallow terms*, making sharing of type variables explicit and avoiding copying types in rules such as U-Decomp. U-True and U-Trivial eliminate trivial constraints, and U-False propagates failure. Finally, U-Cycle implements the *occurs check*, ensuring that a type variable does not occur in the type it is being unified with. This is a necessary condition for unification, as it would otherwise lead to infinite types. This is formalized by the relation $\alpha \prec_U \beta$, which indicates that α occurs in a non-variable type τ equated to β in U. Concretely, this holds when $U = \mathscr{U}[\beta = \tau = \epsilon]$, $\alpha \in \mathsf{fv}(\tau)$, $\tau \notin \mathcal{V}$ and $\alpha, \beta \# \mathsf{bv}(\mathscr{U})$. A unification problem U is said to be cyclic, written $\mathsf{cyclic}(U)$, if $\alpha \prec_U^+ \alpha$ for some α, where $\prec_U^+$ denotes the transitive closure of $\prec_U$.

Definition 5.1 (Solved form $\hat{U}$). We write $\hat{U}$ for constraints in *solved form*, that is, constraints of the form $\exists \bar{\alpha}.\, \bar{\epsilon}$, where: (1) each ϵ_i contains at most one non-variable type; (2) each type variable may occur as a member of at most one multi-equation ϵ_i; (3) the constraint is acyclic.

$$
\begin{array}{llll}
i & & & \text{Instantiation variables} \\
C & ::= & \ldots \mid \mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = C_1\ \mathsf{in}\ C_2 \mid \exists i^x . C \mid i(\alpha) \rightsquigarrow \tau & \text{Constraints} \\
\mathfrak{r} & ::= & (\mathfrak{g}, \phi) \qquad (\mathfrak{r} \in \mathcal{R}) & \text{Ground regions} \\
\mathfrak{R} & \subseteq & \mathcal{R} & \text{Sets of ground regions} \\
\phi & ::= & \ldots \mid \phi[x := \mathfrak{R}] \mid \phi[i := \phi'] & \text{Semantic environments}
\end{array}
$$

$$
\phi(\lambda\alpha[\bar{\alpha}].C) \triangleq \{(\mathfrak{g}, \phi[\alpha := \mathfrak{g}, \bar{\alpha} := \bar{\mathfrak{g}}]) \in \mathcal{R} : \phi[\alpha := \mathfrak{g}, \bar{\alpha} := \bar{\mathfrak{g}}] \vdash C\}
$$

$$
\textsc{LetR}\ \frac{\mathfrak{R} = \phi(\lambda\alpha[\bar{\alpha}].C_1) \quad \mathfrak{R} \neq \emptyset \quad \phi[x := \mathfrak{R}] \vdash C_2}{\phi \vdash \mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = C_1\ \mathsf{in}\ C_2}
\qquad
\textsc{AppR}\ \frac{(\phi(\tau), \_) \in \phi(x)}{\phi \vdash x\ \tau}
$$

$$
\textsc{Exists-Inst}\ \frac{(\_, \phi') \in \phi(x) \quad \phi[i := \phi'] \vdash C}{\phi \vdash \exists i^x . C}
\qquad
\textsc{Incr-Inst}\ \frac{\phi(i)(\alpha) = \phi(\tau)}{\phi \vdash i(\alpha) \rightsquigarrow \tau}
$$

Fig. 11. Syntax and semantics of region-based let and incremental instantiation constraints.

5.2 Administrative constraints

Section §4 introduced the constraint language used for constraint generation (§4.2). In this section we extend the language with a small set of *administrative constraints*: auxiliary constructs that never appear in generated constraints, but are introduced internally by the solver to manage generalization and incremental instantiation.

These administrative constraints enable the solver to represent and refine *partial type schemes* as solving progresses, capturing intermediate generalization states and tracking how instantiations evolve as suspended constraints are discharged.

Regional let-constraints. As is standard in constraint-based formulations of ML type inference [Pottier and Rémy 2005], generalization-related rules operate on constraints of the form $\mathsf{let}\ x = \lambda\alpha.\exists\bar{\alpha}.C_1\ \mathsf{in}\ C_2$. The outermost existentially quantified variables $\bar{\alpha}$ correspond to the *potentially generalizable* variables. We refer to this existential prefix $\exists\bar{\alpha}$ as a *region*. Intuitively, a region delimits the scope of variables that may be generalized once the abstraction has been fully solved.

To make regions explicit, we introduce regional let-constraints $\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = C_1\ \mathsf{in}\ C_2$ (Figure 11), where α is the *root* of the region and $\bar{\alpha}$ are auxiliary existential variables. The order of $\bar{\alpha}$ is immaterial; regions are considered equal up to permutation of these variables.

Satisfiability of regional let-constraints is defined in Figure 11. The semantics of an abstraction with a region, written $\phi(\lambda\alpha[\bar{\alpha}].C)$, is a set $\mathfrak{R}$ of *ground regions* that satisfy C. A ground region $\mathfrak{r}$ is a pair $(\mathfrak{g}, \phi)$; it satisfies the constraint abstraction $\lambda\alpha[\bar{\alpha}].C$ if ϕ satisfies C, and $\mathfrak{g}$ is $\phi(\alpha)$. Intuitively, the region environment ϕ carries a ground type for each inference variable β occurring in C; in particular, a ground instance for α and $\bar{\alpha}$.

The rules LetR and AppR are the *regional counterparts* of the Let and App rules introduced in §4. They are semantically equivalent to their non-regional forms, except that each operates over sets of ground regions ($\mathfrak{R}$) rather than sets of ground types ($\mathfrak{S}$). The additional environment ϕ carried by these ground regions $(\mathfrak{g}, \phi)$ is semantically inert for LetR and AppR themselves, but will become important later, in the semantics of *incremental instantiation* constraints (§5.2).

Regional let-constraints strictly generalize ordinary let-constraints, as captured by the equivalence:

$$
\mathsf{let}\ x = \lambda\alpha.\exists\bar{\alpha}.C_1\ \mathsf{in}\ C_2 \equiv \mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = C_1\ \mathsf{in}\ C_2
$$

The trouble with lets. Solving let-constraints is deceptively difficult. Naively, let-constraints (or *generalization* constraints) could be solved by copying constraints:

$$\frac{\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = C_1\ \mathsf{in}\ \mathscr{C}[x\ \tau] \qquad \alpha, \bar{\alpha} \# \tau \qquad x \# \mathrm{bv}(\mathscr{C})}{\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = C_1\ \mathsf{in}\ \mathscr{C}[\exists \alpha, \bar{\alpha}.\, \alpha = \tau \wedge C_1]} \qquad \textsc{S-Let-App-Beta}$$

This rule, due to Pottier and Rémy [2005], resembles β-reduction in some presentations of explicit substitutions. While this rule is sound when C_1 is a *simple* constraint, it becomes unsound for abstractions containing suspended constraints.

To see why, consider again the example $\mathtt{ex}_8$ from §2.4:

```
type α gpoint = { x : α; y : α }
let diag (n : α) : α gpoint = { x = n; y = n }

let ex₈ gp =                                                    OCaml ✗  OmniML ✓
  let getx p = p.x in getx (diag 42), (getx gp : float)
```

As explained earlier, this program is clearly well-typed: the type of `p` is unambiguously determined as β `gpoint` through *backpropagation* from the first application of `getx` (`getx (diag 42)`). The (simplified) generated constraint for $\mathtt{ex}_8$ contains:

$$\exists \alpha_{\mathsf{gp}}, \delta. \quad \mathsf{let}\ \mathtt{getx} = \lambda \alpha_{\mathtt{getx}}.\, \exists \alpha_{\mathsf{p}}, \gamma.\ \bigwedge \begin{pmatrix} \alpha_{\mathtt{getx}} = \alpha_{\mathsf{p}} \to \gamma \\ \mathsf{match}\ \alpha_{\mathsf{p}}\ \mathsf{with}\ (\mathsf{rcd}\ t\ \_) \Rightarrow t.\mathsf{x} \le \alpha_{\mathsf{p}} \to \gamma \end{pmatrix}$$
$$\mathsf{in}\ \mathtt{getx}\ (\mathsf{rcd}\ \mathsf{gpoint}\ \mathsf{int} \to \delta) \wedge \mathtt{getx}\ (\alpha_{\mathsf{gp}} \to \mathsf{float})$$

If we now apply S-Let-App-Beta to both applications of `getx` (removing the let-constraint for concision), we obtain:

$$\begin{aligned} \exists \alpha_{\mathsf{gp}}, \delta. \quad & \exists \alpha_{\mathsf{p}_1}, \gamma_1.\ \bigwedge \begin{pmatrix} \mathsf{rcd}\ \mathsf{gpoint}\ \mathsf{int} \to \delta = \alpha_{\mathsf{p}_1} \to \gamma_1 \\ \mathsf{match}\ \alpha_{\mathsf{p}_1}\ \mathsf{with}\ (\mathsf{rcd}\ t\ \_) \Rightarrow t.\mathsf{x} \le \alpha_{\mathsf{p}_1} \to \gamma_1 \end{pmatrix} \\ \wedge \quad & \exists \alpha_{\mathsf{p}_2}, \gamma_2.\ \bigwedge \begin{pmatrix} \alpha_{\mathsf{gp}} \to \mathsf{float} = \alpha_{\mathsf{p}_2} \to \gamma_2 \\ \mathsf{match}\ \alpha_{\mathsf{p}_2}\ \mathsf{with}\ (\mathsf{rcd}\ t\ \_) \Rightarrow t.\mathsf{x} \le \alpha_{\mathsf{p}_2} \to \gamma_2 \end{pmatrix} \end{aligned}$$

The first conjunct is satisfiable, since α_{p_1} is unified with rcd `gpoint` int, thereby discharging the match constraint on α_{p_1}. The second, however, is not: α_{p_2} remains underdetermined, leaving its match constraint unsatisfiable. Thus, although the original constraint was satisfiable, the application of S-Let-App-Beta makes it unsatisfiable. By copying the abstraction, we lose the essential *sharing* between both instantiations—namely, that both copies of α_{p} (α_{p_1} and α_{p_2}) must have the same shape $\nu\gamma.\,\mathsf{rcd}$ `gpoint` γ. This loss of sharing is precisely why S-Let-App-Beta is unsound in the presence of suspended constraints.

A tempting alternative, used in Vytiniotis, Jones, Schrijvers and Sulzmann [2011] and Beneš and Brachthäuser [2025], is to treat the let-bindings *monomorphically*, sharing α_{p} directly between both applications. However, this is incomplete with respect to OmniML's type system. It forbids local let-polymorphism (commonly used in OCaml) and fails to typecheck $\mathtt{ex}_8$: the two calls to `getx` require different instantiations of α_{p}—`int gpoint` and `float gpoint`, respectively.

Even setting soundness aside, S-Let-App-Beta is inefficient. Each application duplicates constraint solving work for the same abstraction. A more efficient approach is to *solve once and reuse*: first solve the abstraction once—*e.g.* reducing it to $\lambda\alpha[\bar{\alpha}].\,\bar{\epsilon}$, where $\bar{\alpha}$ are generalizable variables—and then reuse the result at each instantiation site by only copying the solved constraint ϵ. This mirrors the generalization and instantiation steps of ML inference algorithms such as $\mathcal{W}$: $\lambda\alpha[\bar{\alpha}].\,\bar{\epsilon}$ corresponds to the type scheme $\forall\bar{\alpha}.\,\vartheta(\alpha)$, where ϑ is the most general unifier of $\bar{\epsilon}$. Pottier and Rémy [2005] formalize this connection, and the optimized treatment is naturally expressed as a strategy on top of their S-Let-App-Beta rule.

We therefore face two related challenges: (1) handle instantiation soundly in the presence of suspended constraints, and (2) to avoid redundant work by reusing *partial* results across instantiations.

To address both, we introduce *partial type schemes*, our second novel mechanism for omnidirectional inference. Partial type schemes are type schemes that delay commitment to certain quantifications (*e.g.* α_{p} and γ). Such *partially generalized* variables are treated as generalized, but can be incrementally refined in future as suspended constraints are discharged.

Returning to our running example, we begin with the partial type scheme $\forall \alpha_{\mathsf{p}}, \gamma.\, \alpha_{\mathsf{p}} \to \gamma$ for `getx`, since the suspended match constraint in its abstraction cannot yet be solved. We continue solving the body of the let-constraint, tracking every instances of this partial scheme. As type information flows back—*e.g.* when the shape of α_{p} becomes known via backpropagation from the first application of `getx`—the scheme is refined to $\forall \beta.\, \mathsf{rcd}\ \mathtt{gpoint}\ \beta \to \beta$. The second application of `getx` then updates its instantiation accordingly, unifying α_{p_2} with $\mathsf{rcd}\ \mathtt{gpoint}\ \beta_2$ and γ_2 with β_2.

Incremental instantiation. To support partial type schemes, we also extend the constraint language with *incremental instantiation constraints* (Figure 11). We introduce two new constraint formers:

(1) $\exists i^x.\, C$, which binds a fresh instantiation i of x's region within C, and
(2) $i(\alpha) \rightsquigarrow \tau$, which asserts that the copy of α in i equals τ.

The instantiation variable i is required to ensure all incremental instantiations $i(\alpha) \rightsquigarrow \tau$ are solved uniformly.

Within the solver, we view incremental instantiations as markers indicating which parts of the abstraction still need to be copied, and abstractions themselves are treated as partial type schemes.

This mechanism enables efficient handling of constraint instantiations: solved parts are reused immediately, while suspended constraints can be solved later, further refining the abstraction and propagating new equations to all instantiation sites.

We now turn to the semantics of incremental instantiations (Figure 11). The existential constraint $\exists i^x.\, C$ is satisfiable (Exists-Inst) if C is satisfiable with i bound to one of the region environments ϕ' in the interpretation of x. An incremental instantiation $i(\alpha) \rightsquigarrow \tau$ is satisfiable (Incr-Inst) exactly when i's instance of α (*i.e.*, $\phi(i)(\alpha)$) is equal to τ.

When a regional abstraction $\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = C_1\ \mathsf{in}\ C_2$ contains an incremental instantiation constraint $i(\beta) \rightsquigarrow \tau$, and i is an instantiation of x, the variable β may range over any free variable of C_1—including α and the regional variables $\bar{\alpha}$. In other words, the regional let-constraint $\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = C_1\ \mathsf{in}\ C_2$ provides a non-obvious scoping rule: the variables α and $\bar{\alpha}$ are also bound in C_2, but may occur there only within incremental instantiations of x. This subtlety justifies introducing an explicit syntax for regional let-constraints, rather than encoding them as $\mathsf{let}\ x = \lambda\alpha.\, \exists\bar{\alpha}.\, C_1\ \mathsf{in}\ C_2$.

5.3 Solving rules

We now gradually introduce the rules of the constraint solver itself (Figures 12 to 14; for a complete view with all rules placed together, see Appendix §A). These rules define a non-deterministic rewriting system, operating modulo α-equivalence, and the associativity and commutativity of conjunction. Rewriting takes place under an arbitrary one-hole constraint context $\mathscr{C}$. A constraint C is satisfiable if it rewrites to a solved form $\hat{U}$ (Definition 5.1); otherwise it gets stuck—in particular, false is considered a stuck constraint. To characterize all stuck constraints, we introduce *normal forms*. Every solved form is a normal form, but not every normal form is solved.

S-Unif
$$\frac{U_1 \qquad U_1 \longrightarrow U_2}{U_2}$$

S-False
$$\frac{\mathscr{C}[\text{false}] \qquad \mathscr{C} \neq \square}{\text{false}}$$

S-Let
$$\frac{\text{let } x = \lambda\alpha.\, C_1 \text{ in } C_2}{\text{let } x\ \alpha\ [\emptyset] = C_1 \text{ in } C_2}$$

S-Exists-Conj
$$\frac{(\exists\alpha.\, C_1) \wedge C_2 \qquad \alpha \# C_2}{\exists\alpha.\, C_1 \wedge C_2}$$

S-Let-ExistsLeft
$$\frac{\text{let } x\ \alpha\ [\bar\alpha] = \exists\beta.\, C_1 \text{ in } C_2 \qquad \beta \# \alpha, \bar\alpha, C_2}{\text{let } x\ \alpha\ [\bar\alpha, \beta] = C_1 \text{ in } C_2}$$

S-Let-ExistsRight
$$\frac{\text{let } x\ \alpha\ [\bar\alpha] = C_1 \text{ in } \exists\beta.\, C_2 \qquad \beta \# \alpha, \bar\alpha, C_1}{\exists\beta.\, \text{let } x\ \alpha\ [\bar\alpha] = C_1 \text{ in } C_2}$$

S-Let-ConjLeft
$$\frac{\text{let } x\ \alpha\ [\bar\alpha] = C_1 \wedge C_2 \text{ in } C_3 \qquad C_1 \# \alpha, \bar\alpha}{C_1 \wedge \text{let } x\ \alpha\ [\bar\alpha] = C_2 \text{ in } C_3}$$

S-Let-ConjRight
$$\frac{\text{let } x\ \alpha\ [\bar\alpha] = C_1 \text{ in } (C_2 \wedge C_3) \qquad x \# C_2}{C_2 \wedge \text{let } x\ \alpha\ [\bar\alpha] = C_1 \text{ in } C_3}$$

Fig. 12. Basic rewriting rules $C_1 \longrightarrow C_2$.

Definition 5.2 (Normal forms). A constraint C is in *normal form*, written $\hat{C}$, if no rewriting rules apply, *i.e.*, $C \not\longrightarrow$. Similarly, a constraint context $\mathscr{C}$ is in *normal form*, written $\hat{\mathscr{C}}$, if $\mathscr{C}[\text{true}]$ is in normal form.

Basic rules. Figure 12 contains a selected set of basic solving rules, drawn from the standard repertoire of small-step constraint solvers for ML type inference. S-Unif invokes the unification algorithm on the current unification problem. The unification algorithm itself is treated as a black box by the solver, so the system could be extended with any equational theory of types implemented by the unification algorithm. S-Let rewrites let-constraints into regional form. S-Exists-Conj lifts existentials across conjunctions; S-Let-ExistsLeft and S-Let-ExistsRight lift existentials across let-binders; S-Let-ConjLeft, S-Let-ConjRight hoist constraints out of let-binders when they are independent of the local variables. Collectively, these lifting rules normalize the structure of each region into a block of existentially bound variables, whose body consists of a conjunction of solved multi-equations followed by a residual constraint—typically an instantiation, let-binding, or suspended constraint.

OmniML-specific constraints, such as the label and polytype instantiation constraints ($t.\ell \leq \tau_1 \rightarrow \tau_2$, $s \leq \tau$, *etc.*), require no special treatment in our solver. Once their pattern variables are substituted—after solving a match constraint—they are desugared into constraints already handled by the solver.

Incremental instantiations. Figure 13 describes the solving rules in charge of incremental instantiation. Incremental instantiation constraints are reduced using the following rules:

(1) S-Inst-Copy copies the shape of a type to the instantiation site, introducing fresh variables for each subcomponents and marking them with corresponding instantiation constraints. We write $i^x(\beta) \rightsquigarrow \tau$ as a shorthand for $i(\beta) \rightsquigarrow \tau$ when i is bound with $\exists i^x$ in the context. To ensure termination, the abstraction must contain acyclic types.
(2) S-Inst-Unif unifies two instantiations if they both refer to the same source variable β at the same instantiation site i.

There are three cases in which an instantiation constraint is eliminated:

(1) A base type (*i.e.*, a nullary shape) is copied and no further instantiations are needed (S-Inst-Copy).
(2) The copied variable α' is polymorphic, and thus the instantiation constraint imposes no restriction (S-Inst-Poly), provided no other instantiations of α' exist for the same instantiation i (if not, then apply S-Inst-Unif).

$$\textsc{S-Inst-Copy}\quad \dfrac{\begin{array}{c}\mathsf{let}\ x\ \alpha\ [\bar\alpha] = C\ \mathsf{in}\ \mathscr{C}[i^x(\beta) \rightsquigarrow \gamma] \\ \neg\mathsf{cyclic}(C) \qquad x \# \mathsf{bv}(\mathscr{C}) \\ C = C' \wedge \beta = \varsigma\,\bar\beta = \epsilon \qquad \beta \in \alpha, \bar\alpha \qquad \bar\beta' \# \beta, \gamma, \bar\beta\end{array}}{\mathsf{let}\ x\ \alpha\ [\bar\alpha] = C\ \mathsf{in}\ \mathscr{C}[\exists \bar\beta'.\, \gamma = \varsigma\,\bar\beta' \wedge \overline{i^x(\beta) \rightsquigarrow \beta'}]}$$

$$\textsc{S-Inst-Unif}\quad \dfrac{i(\beta) \rightsquigarrow \gamma_1 \wedge i(\beta) \rightsquigarrow \gamma_2}{i(\beta) \rightsquigarrow \gamma_1 \wedge \gamma_1 = \gamma_2}$$

$$\textsc{S-Inst-Poly}\quad \dfrac{\begin{array}{c}\mathsf{let}\ x\ \alpha\ [\bar\alpha] = \bar\epsilon \wedge C\ \mathsf{in}\ \mathscr{C}[i^x(\alpha') \rightsquigarrow \gamma] \qquad \forall \beta.\, \exists \bar\beta.\, \bar\epsilon \equiv \mathsf{true} \\ \beta, \bar\beta \subseteq \alpha, \bar\alpha \qquad \bar\beta \# C \qquad i(\beta) \# \mathsf{insts}(\mathscr{C}) \qquad x \# \mathsf{bv}(\mathscr{C})\end{array}}{\mathsf{let}\ x\ \alpha\ [\bar\alpha] = \bar\epsilon \wedge C\ \mathsf{in}\ \mathscr{C}[\mathsf{true}]}$$

$$\textsc{S-Inst-Mono}\quad \dfrac{\begin{array}{c}\mathsf{let}\ x\ \alpha\ [\bar\alpha] = C\ \mathsf{in}\ \mathscr{C}[i^x(\beta) \rightsquigarrow \gamma] \\ \beta \notin \alpha, \bar\alpha \qquad x, \beta \# \mathsf{bv}(\mathscr{C})\end{array}}{\mathsf{let}\ x\ \alpha\ [\bar\alpha] = C\ \mathsf{in}\ \mathscr{C}[\beta = \gamma]}$$

$$\textsc{S-Let-AppR}\quad \dfrac{\begin{array}{c}\mathsf{let}\ x\ \alpha\ [\bar\alpha] = C\ \mathsf{in}\ \mathscr{C}[x\ \tau] \\ \gamma \# \tau \qquad x \# \mathsf{bv}(\mathscr{C})\end{array}}{\mathsf{let}\ x\ \alpha\ [\bar\alpha] = C\ \mathsf{in}\ \mathscr{C}[\exists \gamma, i^x.\, i(\alpha) \rightsquigarrow \gamma \wedge \gamma = \tau]}$$

$$\textsc{S-Let-Solve}\quad \dfrac{\begin{array}{c}\mathsf{let}\ x\ \alpha\ [\bar\alpha] = \bar\epsilon\ \mathsf{in}\ C \\ \exists \alpha, \bar\alpha.\, \bar\epsilon \equiv \mathsf{true} \qquad x \# C\end{array}}{C}$$

$$\textsc{S-Compress}\quad \dfrac{\mathsf{let}\ x\ \alpha\ [\bar\alpha, \beta] = C_1 \wedge \beta = \gamma = \epsilon\ \mathsf{in}\ C_2 \qquad \beta \neq \gamma}{\mathsf{let}\ x\ \alpha\ [\bar\alpha] = C_1[\beta := \gamma] \wedge \gamma = \epsilon[\beta := \gamma]\ \mathsf{in}\ C_2[x(\beta) := \gamma]}$$

$$\textsc{S-Exists-Lower}\quad \dfrac{\begin{array}{c}\mathsf{let}\ x\ \alpha\ [\bar\alpha, \bar\beta] = C_1\ \mathsf{in}\ C_2 \\ \vdash \exists \alpha, \bar\alpha.\, C_1\ \mathsf{determines}\ \bar\beta\end{array}}{\exists \bar\beta.\, \mathsf{let}\ x\ \alpha\ [\bar\alpha] = C_1\ \mathsf{in}\ C_2}$$

$$\textsc{Det-Dom}\quad \dfrac{\gamma \# \bar\beta, \bar\alpha \qquad \bar\alpha \subseteq \mathsf{fv}(\epsilon)}{\vdash \exists \bar\beta.\, C \wedge \gamma = \epsilon\ \mathsf{determines}\ \bar\alpha}$$

$$\textsc{Det-Esc}\quad \dfrac{\mathsf{fv}(\tau) \# \bar\alpha, \bar\beta}{\vdash \exists \bar\beta.\, C \wedge \bar\alpha = \tau = \epsilon\ \mathsf{determines}\ \bar\alpha}$$

Fig. 13. Solving rules for let-constraints and instantiations.

(3) The copy is monomorphic and in scope, so we unify it directly (S-Inst-Mono).

S-Let-AppR rewrites an instantiation constraint $x\ \tau$ introducing an incremental instantiation constraint $i(\alpha) \rightsquigarrow \gamma$. Here, i is a fresh instantiation of x, α is the *root* of x's region, and γ is a fresh alias for τ. We introduce γ explicitly, since our rewriting rules for incremental instantiations generally assume that the copied type is a variable rather than an arbitrary type.

Let-constraints. Figure 13 also presents the rules governing generalization. S-Let-Solve removes a let-constraint when the bound term variable is unused and the abstraction is satisfiable. S-Compress determines that a regional variable β is an an alias for γ. We replace every free occurrence of β with γ—*including* the domains of any incremental instantiation constraints, written as the substitution $[x(\beta) := \gamma]$.

Conceptually, S-Compress acts a variable-level analogue of S-Inst-Copy: both rules copy solved constraints ϵ from an abstraction to its instantiation constraints. While S-Inst-Copy propagates the head shape of a multi-equation, S-Compress propagates equalities between variables within a multi-equation, thereby enabling subsequent applications of S-Inst-Unif.

S-Exists-Lower implements the non-trivial case of lowering existentials across let-binders. It identifies a subset of variables in the region of a let-constraint that are unified with variables from outside the region. Such variables are considered monomorphic and thus cannot be generalized; they can instead be safely lowered to the outer scope.

$$\textsc{S-Match-Ctx}\quad \mathscr{C}[\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}] \quad \vdash \mathscr{C}[\tau \,!\, \varsigma] \longrightarrow \mathscr{C}[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}]$$

$$\textsc{Uni-Var}\quad \frac{\alpha \# \mathsf{bv}(\mathscr{C}_2)}{\vdash \mathscr{C}_1[\alpha = \tau = \epsilon \wedge \mathscr{C}_2[\square]][\alpha \,!\, \mathsf{shape}(\tau)]}$$

$$\textsc{Uni-Type}\quad \frac{\tau \notin \mathcal{V}}{\vdash \mathscr{C}[\tau \,!\, \mathsf{shape}(\tau)]}$$

$$\textsc{Uni-BackProp}\quad \frac{\vdash (\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = \mathscr{C}_1[\mathsf{true}]\ \mathsf{in}\ \mathscr{C}_2[i^x(\alpha') \rightsquigarrow \gamma \wedge \square])[\gamma \,!\, \varsigma] \qquad \alpha' \in \alpha, \bar{\alpha} \qquad x \# \mathsf{bv}(\mathscr{C}_2) \qquad \alpha' \# \mathsf{bv}(\mathscr{C}_1)}{\vdash (\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = \mathscr{C}_1[\square]\ \mathsf{in}\ \mathscr{C}_2[i^x(\alpha') \rightsquigarrow \gamma])[\alpha' \,!\, \varsigma]}$$

Fig. 14. Rewriting rules for suspended match constraints.

This is the case when the types of $\bar{\beta}$ are *determined* in a unique way. In short, C determines $\bar{\beta}$ if and only if the solutions for $\bar{\beta}$ are uniquely fixed by the solutions to other variables in C.

Definition 5.3. C determines $\bar{\beta}$ if and only if every ground assignments ϕ and ϕ' that satisfy (the erasure of) C and coincide outside of $\bar{\beta}$ coincide on $\bar{\beta}$ as well.

$$C \text{ determines } \beta \;\triangleq\; \forall \phi, \phi'.\;\; \phi \vdash \lfloor C \rfloor \wedge \phi' \vdash \lfloor C \rfloor \wedge \phi =_{\backslash \bar{\beta}} \phi' \implies \phi = \phi'$$

Conceptually, this corresponds to the negation of the generalization condition in ML: a type variable *cannot* be generalized if it appears in the typing context. In the constraint setting, it *cannot* be generalized if it depends on variables from outside the region. For instance, $\exists \beta. \alpha = \beta \rightarrow \gamma$ determines γ, as α is free and therefore constrains the solution of γ.

To decide when C determines $\bar{\alpha}$, we introduce the judgment $\vdash C$ determines $\bar{\alpha}$, which syntactically proves that $\bar{\alpha}$ are determined in C. If C is of the form $\exists \bar{\beta}. C'$ where $\bar{\beta} \# \bar{\alpha}$, then we search for a multi-equation ϵ in C' of the form: (Det-Dom) $\gamma = \epsilon'$ where $\gamma \# \bar{\alpha}, \bar{\beta}$ and $\bar{\alpha} \subseteq \mathsf{fv}(\epsilon')$, or (Det-Esc) $\bar{\alpha} = \tau = \epsilon'$ where $\mathsf{fv}(\tau) \# \bar{\alpha}, \bar{\beta}$. This syntactic relation coincides with the semantic definition of determinacy whenever C is in solved form. Otherwise, it is a sound approximation of the semantic definition.

Lowering such variables improves solver efficiency. It avoids unnecessary duplication of work that would otherwise occur via S-Inst-Copy. Because these variables are determined by monomorphic ones, lowering allows the solver to apply S-Inst-Mono directly at instantiation sites, rather than duplicating equivalent constraints that ultimately express the same semantic fact—that the variable is monomorphic and shared across instances.

Suspended match constraints. Figure 14 describes the rules for discharging suspended constraints. S-Match-Ctx solves suspended match constraints. It would not be effective to allow rewriting whenever the unicity condition $\mathscr{C}[\tau \,!\, \varsigma]$ holds, because it is not a-priori feasible to check or decide this semantic condition which quantifies over all solutions. Instead we introduce a restricted, decidable approximation via three syntactic "unicity rules" that form the judgment $\vdash \mathscr{C}[\tau \,!\, \varsigma]$.

The unicity rule Uni-Type applies when τ is a non-variable type τ, in which case the shape is simply $\mathsf{shape}(\tau)$. Uni-Var applies when the scrutinee is a variable α and the context establishes that α is equal to some non-variable type τ by exhibiting an equality $\alpha = \tau = \epsilon$ and τ is a non-variable type. In this case, the shape of α is $\mathsf{shape}(\tau)$.

Finally, Uni-BackProp expresses *backpropagation*, previously illustrated in Example 4.10. In particular, the shape of a regional variable can sometimes be determined from its instantiations. If an abstraction contains a regional variable α', and the constraint context includes an incremental instantiation $i^x(\alpha') \rightsquigarrow \gamma$ such that the instance γ of α' has the unique shape ς, then α' must also have shape ς, as any other shape would render the instantiation unsatisfiable. This rule is well-founded because the regional depth of the hole strictly decreases in the premise.

For a normalized context $\hat{\mathscr{C}}$ (Definition 5.2), this syntactic condition exactly characterizes the semantic definition of unicity: $\mathscr{C}[\tau\,!\,\varsigma]$ holds if and only if $\vdash \mathscr{C}[\tau\,!\,\varsigma]$ is derivable (Appendix Lemma C.11).

Example 5.4. Let us consider the (simplified) constraint generated for $\mathtt{ex}_8$, discussed in §5.2. For readability, we begin by factoring out the following invariant two-hole constraint context $\mathscr{C}$:

$$\mathscr{C}\,[\square_1, \square_2] \quad \triangleq \quad \begin{array}{ll} \exists \alpha_{\mathrm{gp}}, \delta. & \mathsf{let}\ \mathtt{getx} = \lambda \alpha_{\mathtt{getx}}.\, \exists \alpha_{\mathrm{p}}, \gamma.\, \alpha_{\mathtt{getx}} = \alpha_{\mathrm{p}} \to \gamma \wedge \square_1 \\ & \mathsf{in}\ \square_2 \wedge \mathtt{getx}\ (\alpha_{\mathrm{gp}} \to \mathsf{float}) \end{array}$$

The initial constraint can then be written as follows:

$$\begin{array}{ll} \exists \alpha_{\mathrm{gp}}, \delta. & \mathsf{let}\ \mathtt{getx} = \lambda \alpha_{\mathtt{getx}}.\, \exists \alpha_{\mathrm{p}}, \gamma.\, \bigwedge \left(\begin{array}{l} \alpha_{\mathtt{getx}} = \alpha_{\mathrm{p}} \to \gamma \\ \mathsf{match}\ \alpha_{\mathrm{p}}\ \mathsf{with}\ (\mathsf{rcd}\ t\ \_) \Rightarrow t.\mathsf{x} \leq \alpha_{\mathrm{p}} \to \gamma \end{array} \right) \\ & \mathsf{in}\ \mathtt{getx}\ (\mathsf{rcd}\ \mathsf{gpoint}\ \mathsf{int} \to \delta) \wedge \mathtt{getx}\ (\alpha_{\mathrm{gp}} \to \mathsf{float}) \end{array}$$

$$\equiv \quad \mathscr{C}\left[\mathsf{match}\ \alpha_{\mathrm{p}}\ \mathsf{with}\ (\mathsf{rcd}\ t\ \_) \Rightarrow t.\mathsf{x} \leq \alpha_{\mathrm{p}} \to \gamma, \mathtt{getx}\ (\mathsf{rcd}\ \mathsf{gpoint}\ \mathsf{int} \to \delta)\right]$$

We now reduce the constraint step by step using the rules outlined above.

$$\begin{array}{lll}
& \mathscr{C}\left[\mathsf{match}\ \alpha_{\mathrm{p}}\ \mathsf{with}\ (\mathsf{rcd}\ t\ _) \Rightarrow t.\mathsf{x} \leq \alpha_{\mathrm{p}} \to \gamma \quad , \quad \mathtt{getx}\ (\mathsf{rcd}\ \mathsf{gpoint}\ \mathsf{int} \to \delta)\right] & \\
\longrightarrow & \mathscr{C}\left[\ldots, \quad \exists \gamma_{\mathtt{getx}}, i^{\mathtt{getx}}.\, i(\alpha_{\mathtt{getx}}) \rightsquigarrow \gamma_{\mathtt{getx}} \wedge \gamma_{\mathtt{getx}} = \mathsf{rcd}\ \mathsf{gpoint}\ \mathsf{int} \to \delta\right] & \textsc{S-Let-AppR} \\
\longrightarrow & \mathscr{C}\left[\ldots, \quad \exists \gamma_{\mathtt{getx}}, \gamma_{\mathrm{p}}, \gamma_1, i^{\mathtt{getx}}.\, \bigwedge \left(\begin{array}{l} \gamma_{\mathtt{getx}} = \gamma_{\mathrm{p}} \to \gamma_1 \wedge i(\alpha_{\mathrm{p}}) \rightsquigarrow \gamma_{\mathrm{p}} \wedge i(\gamma) \rightsquigarrow \gamma_1 \\ \gamma_{\mathtt{getx}} = \mathsf{rcd}\ \mathsf{gpoint}\ \mathsf{int} \to \delta \end{array} \right)\right] & \textsc{S-Inst-Copy} \\
\longrightarrow^{+} & \mathscr{C}\left[\ldots, \quad \exists \gamma_{\mathtt{getx}}, \gamma_{\mathrm{p}}, \gamma_1, i^{\mathtt{getx}}.\, \bigwedge \left(\begin{array}{l} i(\alpha_{\mathrm{p}}) \rightsquigarrow \gamma_{\mathrm{p}} \wedge i(\gamma) \rightsquigarrow \gamma_1 \wedge \gamma_{\mathtt{getx}} = \gamma_{\mathrm{p}} \to \gamma_1 \\ \gamma_1 = \delta \wedge \gamma_{\mathrm{p}} = \mathsf{rcd}\ \mathsf{gpoint}\ \mathsf{int} \end{array} \right)\right] & \\
\longrightarrow & \mathscr{C}\left[\mathsf{match}\ \alpha_{\mathrm{p}} := \nu\eta.\, \mathsf{rcd}\ \mathsf{gpoint}\ \eta\ \mathsf{with}\ (\mathsf{rcd}\ t\ _) \Rightarrow t.\mathsf{x} \leq \alpha_{\mathrm{p}} \to \gamma \quad , \quad \ldots\right] & \textsc{S-Match-Ctx} \\
\longrightarrow^{+} & \mathscr{C}\left[\mathsf{gpoint.x} \leq \alpha_{\mathrm{p}} \to \gamma \quad , \quad \ldots\right] & \\
\longrightarrow^{+} & \mathscr{C}\left[\alpha_{\mathrm{p}} = \mathsf{rcd}\ \mathsf{gpoint}\ \gamma \quad , \quad \ldots\right] & \\
\longrightarrow & \mathscr{C}\left[\ldots, \quad \exists \gamma_{\mathtt{getx}}, \gamma_{\mathrm{p}}, \gamma_1, \gamma_1', i^{\mathtt{getx}}.\, \bigwedge \left(\begin{array}{l} i(\gamma) \rightsquigarrow \gamma_1 \wedge i(\gamma) \rightsquigarrow \gamma_1' \wedge \gamma_{\mathtt{getx}} = \gamma_{\mathrm{p}} \to \gamma_1 \\ \gamma_1 = \delta \wedge \gamma_{\mathrm{p}} = \mathsf{rcd}\ \mathsf{gpoint}\ \mathsf{int} \\ \gamma_{\mathrm{p}} = \mathsf{rcd}\ \mathsf{gpoint}\ \gamma_1' \end{array} \right)\right] & \textsc{S-Inst-Copy} \\
\longrightarrow & \mathscr{C}\left[\ldots, \quad \exists \gamma_{\mathtt{getx}}, \gamma_{\mathrm{p}}, \gamma_1, \gamma_1', i^{\mathtt{getx}}.\, \bigwedge \left(\begin{array}{l} i(\gamma) \rightsquigarrow \gamma_1 \wedge \gamma_{\mathtt{getx}} = \gamma_{\mathrm{p}} \to \gamma_1 \\ \gamma_1 = \delta \wedge \gamma_{\mathrm{p}} = \mathsf{rcd}\ \mathsf{gpoint}\ \mathsf{int} \\ \gamma_{\mathrm{p}} = \mathsf{rcd}\ \mathsf{gpoint}\ \gamma_1' \wedge \gamma_1 = \gamma_1' \end{array} \right)\right] & \textsc{S-Inst-Unif} \\
\longrightarrow^{+} & \mathscr{C}\left[\ldots, \quad \exists \gamma_{\mathtt{getx}}, \gamma_{\mathrm{p}}, \gamma_1, \gamma_1', i^{\mathtt{getx}}.\, \bigwedge \left(\begin{array}{l} i(\gamma) \rightsquigarrow \gamma_1 \\ \gamma_{\mathtt{getx}} = \gamma_{\mathrm{p}} \to \gamma_1 \wedge \gamma_{\mathrm{p}} = \mathsf{rcd}\ \mathsf{gpoint}\ \gamma_1 \\ \gamma_1 = \delta = \gamma_1' = \mathsf{int} \end{array} \right)\right] & \\
\longrightarrow & \mathscr{C}\left[\ldots, \quad \exists \gamma_{\mathtt{getx}}, \gamma_{\mathrm{p}}, \gamma_1, \gamma_1', i^{\mathtt{getx}}.\, \bigwedge \left(\begin{array}{l} \gamma_{\mathtt{getx}} = \gamma_{\mathrm{p}} \to \gamma_1 \wedge \gamma_{\mathrm{p}} = \mathsf{rcd}\ \mathsf{gpoint}\ \gamma_1 \\ \gamma_1 = \delta = \gamma_1' = \mathsf{int} \end{array} \right)\right] & \textsc{S-Inst-Poly}
\end{array}$$

We begin by applying S-Let-AppR to the application of getx, introducing a fresh instantiation i. Applying S-Inst-Copy then copies the arrow shape of $\alpha_{\mathtt{getx}}$ and introduces two incremental instantiation constraints on the parameter type α_{p} and return type γ. Using U-Merge and U-Decomp, we combine the two equations on $\gamma_{\mathtt{getx}}$, from which the solver deduces that γ_{p} must be the record type rcd gpoint int. From this equality on γ_{p}, it follows that γ_{p} has the shape $\nu\eta.\,\mathsf{rcd}\ \mathtt{gpoint}\ \eta$. Together with the instantiation $i(\alpha_{\mathsf{p}}) \rightsquigarrow \gamma_{\mathsf{p}}$, we can apply S-Match-Ctx using Uni-BackProp. The match constraint is then discharged, yielding a label instantiation constraint forcing α_{p} to be rcd gpoint γ. We next apply S-Inst-Copy to this equality on α_{p}, introducing a duplicate instantiation constraint on γ, which is eliminated using S-Inst-Unif. Finally, applying U-Merge and U-Decomp simplifies the remaining equalities on γ_1 and γ_{p}, and S-Inst-Poly removes the last instantiation constraint on γ, since it is polymorphic.

5.4 Metatheory

In this section, we establish the correctness of our solver. Correctness follows from three standard metatheoretic properties: *progress*, *preservation*, and *termination*. Together, they ensure that every satisfiable (term-variable-closed) constraint eventually reduces to an equivalent solved form.

Definition 5.5. A constraint C is term-variable-closed if all its term variables x are bound *i.e.*, $\mathsf{fv}(C) \subseteq \mathcal{V}$.

Lemma 5.6 (Scope preservation). *If $C_1 \longrightarrow C_2$, then $\mathsf{fv}(C_1) \supseteq \mathsf{fv}(C_2)$.*

Theorem 5.7 (Closed Progress). *If a term-variable-closed constraint C cannot take a step $C \longrightarrow C'$, then either:*

(1) C is solved.
(2) C is false.
(3) for every match constraint $C = \hat{\mathscr{C}}[\mathsf{match}\ \alpha\ \mathsf{with}\ \bar{\chi}]$ with $\hat{C}$ in normal form (Definition 5.2), $\hat{\mathscr{C}}[\alpha \,!\, \varsigma]$ does not hold for any ς.

Theorem 5.8 (Termination). *The constraint solver terminates on all inputs.*

Theorem 5.9 (Preservation). *If $C_1 \longrightarrow C_2$, then $C_1 \equiv C_2$.*

Corollary 5.10 (Correctness). *For the term-variable-closed constraint C, C is satisfiable if and only if $C \longrightarrow^* \hat{U}$ and $\hat{U}$ is a solved form equivalent to C.*

6 Implementation

We have a working prototype implementing the OmniML language with suspended match constraints and partial type schemes, in which we have reproduced the various type-system features and examples presented in this work. The implementation closely follows the constraint-based presentation described in the previous section. It is public and open-source, available at https://github.com/johnyob/omniml. Its implementation is inspired by previous work such as Inferno [Pottier 2014, 2018]. It uses state-of-the-art ML type inference implementation techniques for efficiency, such as a Tarjan's union-find data structure for unification [Tarjan 1975] and *ranks* (or *levels*) for efficient generalization [Rémy 1992]. Let us discuss a few salient points.

Unification and scheduling. Each unsolved unification variable maintains a *wait list* of suspended constraints that are blocked until the variable is unified with a concrete type. When such a unification occurs, the wait list is flushed: the suspended constraints are scheduled on the global constraint scheduler, which is responsible for eventually solving them.

From a stack to a tree. Some efficient implementations of ML type inference, starting with Rémy [1992], represent the solver state as a linear *stack* of inference regions, from the outermost variable scope to the current region. Each let-binding and, more generally, each generalization location, introduces a new scope, hence a new region. Since the solver visits the source terms, hence introduces local regions in a depth-first manner, these variable scopes may be represented by an integer *rank* or *level*[15] attached to each inference variable or type node. Unification maintains these levels to their minimum value when merging types, which amounts to compute the least common ancestor of the region types should belong to. The use of integer levels to represent regions, which is quite efficient and simple to implement, does not suffice for partial generalization. If generalization at some region contain a variable appearing in a suspended match constraint, the region must be kept alive while we continue inference in other regions. Hence, later parts of the constraint may introduce a new let-region at the same level that is unrelated to the suspended one—neither its ancestor nor its descendant—breaking the linear assumption that allowed the representation of variable scopes by integer levels.

We must instead use a *tree* of nested let-regions to represent scopes, similar to the binding tree of G-nodes in MLF [Rémy and Yakobowski 2008]. Under this scheme, levels no longer uniquely determine a variable's region. Instead, we interpret a level relative to a path in the region tree from the root. When two variables are unified, they must always lie on some shared path—by scoping invariants—so computing their minimum level (along this path) still suffices to determine the least common ancestor: we keep the efficient integer comparisons.

Partial generalization. Generalization is the process of determining which type variables are polymorphic and which are monomorphic (*i.e.*, implementing S-Exists-Lower). *Partial generalization* arises when a region cannot be fully generalized due to suspended constraints that may update its variables. To manage this, we classify type variables into four categories:

- (**I**) Variables are yet to be generalized.
  Introduced by instantiations or source types in constraints.
- (**G**) Variables that are generalized.
  Not accessible from any instance type. Definitely polymorphic.
- (**PG**) Variables that are partially generalizable.
  Generalizable variables mentioned by suspended match constraint or partial instantiations. Maybe polymorphic, maybe monomorphic.
- (**PI**) Variables that were previously partially generalized but have since been updated.
  Awaiting re-generalization. Introduced by the unification of partially generalized types.

During solving, we track a set of *guards* for each type variable. Each guard indicates that the variable is captured by a suspended constraint, and may therefore be updated when that constraint is discharged. At generalization time, these guards conservatively approximate whether a variable may be updated in the future. Variables with at least one guard are partially generalized (**PG**); those without guards are fully generalized (**G**).

When an instance is taken from a partially generalized variable α (**PG**), we retain a forward reference from α to the instance β. This enables α to notify β if it is later updated, at which point α becomes a partial instance (**PI**) and propagates its updates to β (and to its other instances). This mirrors, in reverse, the way our formalized solver uses incremental instantiation constraints to track copies. In addition, the instance β remains guarded by α until the latter is either lowered or fully generalized.

Once a suspended match constraint is solved, it removes the guards it introduced. This may enable previously partially generalized variables α (**PG**) to become fully generalized (**G**). Conversely, if a partially generalized variable α is lowered (*e.g.* by S-Exists-Lower), becoming an instance (**PI** or **I**), it is unified with all its instances $\bar{\beta}$ obtained via its forward references.

[15] Ranks or levels in fact correspond to De Bruijn levels.

Lazy generalization. Repeatedly generalizing a region after every update is expensive. Instead we generalize on demand. We mark regions as "stale" when they may require re-generalization. When an instance is taken, we re-generalize the stale descendants of the region in the region tree.

Although this technique prevents premature re-generalization before instantiation, it only optimizes for a common case. In some situations, re-generalization before instantiation can still be more costly than instantiating first, since the instantiation may discharge suspended constraints that further refine the region's partial type scheme.

7 Discussion

We now discuss practical considerations and an alternative semi-unification-based implementation of incremental instantiation.

Error reporting. Recent work on error reporting for constraint-based inference [Bhanuka, Parreaux, Binder and Brachthäuser 2023] shows that the structure of constraints can be leveraged to produce precise, localized diagnostics. Our setting fits well with this line of work: suspended constraints, in particular, do not fundamentally complicate error reporting. In fact, the causal ordering among suspended constraints gives a natural account of information flow, which can aid in tracing the source of type errors. At present, our prototype supports only basic error reporting, attaching source locations to constraints. That said, we see no obstacle to adopting the approach of Bhanuka, Parreaux, Binder and Brachthäuser [2023].

Integration into OCaml. Integrating suspended constraints into OCaml's type checker raises two questions: their compatibility with OCaml's existing type system, and the feasibility of implementing them within OCaml's current typechecker.

From a type-theoretic perspective, we are confident that suspended constraints and incremental instantiation are compatible with the various features of the OCaml type system. Moreover, several other features of OCaml could benefit from suspended constraints. In some cases, such as GADTs, this perspective could open up a new research topic of its own. Nevertheless, reformulating every fragile feature to use omnidirectionality is not required for initial adoption in OCaml: existing features could remain unchanged, with suspended constraints introduced selectively where it provides clear benefits.

For the implementation perspective, the situation is more delicate. The current OCaml has grown organically over more than two decades. The underlying inference algorithm uses a single-pass architecture implemented with global mutable state and few abstraction boundaries. As a result, the typechecker is large, intricate and increasingly difficult to maintain. Our current expert opinion is that extending the existing implementation to support suspended constraints is feasible.[16] However, such an extension would combine the existing mutable state and in-place updates with a more complex control flow, raising concerns about code complexity and long-term maintainability. These concerns could ultimately argue against integrating suspended constraints into the current typechecker.

At the same time, there are ongoing efforts towards a clean reimplementation of the OCaml typechecker. Such designs would provide a more natural setting for suspended constraints and could significantly simplify their integration.

Incremental instantiation via semi-unification. Semi-unification [Henglein 1989, 1993], introduced in the late 1980's, is a generalization of unification that solves a collection of *instantiation inequalities*

[16] Compared to the implementation described in §6, OCaml already maintains a tree of inference regions. One could introduce the additional data structures required for *partial and lazy generalization*, along with the scheduling mechanisms needed to implement suspended constraints.

$\bigwedge_{i\in I} \tau_i \leq \tau'_i$, meaning that there exist a unifier θ and a collection of substitutions $\bar{\rho}$ such that $\rho_i(\theta(\tau_i)) = \theta(\tau'_i)$. Unfortunately, semi-unification was soon proved undecidable [Kfoury, Tiuryn and Urzyczyn 1993] and almost abandoned for type inference purposes—even though Henglein observed that his semi-algorithm appeared to terminate on *most* ML type inference problems involving polymorphic recursion.[17] Moreover, in the absence of polymorphic recursion, semi-unification constraints arising from ML type inference are acyclic and therefore guaranteed to terminate.

For our purposes, semi-unification offers an alternative foundation for incremental instantiation. Because semi-unification constraints can be solved *incrementally*, in any order, making them a natural fit for scaling omnidirectional type inference to ML-style polymorphism. Although it offered little advantage for traditional ML inference, Henglein's algorithm was *omnidirectionally-ready*, merely waiting for fragile constructs to give it new life!

Motivated by this observation, we explored a second prototype that implements let-generalization and incremental instantiation via semi-unification. To enable a more direct encoding of type inference problems and improve solving efficiency, we enriched semi-unification constraints with a notion of *scope*, forming the same tree-like structure as regions in our *generalization tree* (§6). This extends the formalization of semi-unification with *unknowns* proposed by Lushman and Cormack [2008], and allows for more efficient handling of *monomorphization*—that is, instantiations that collapse into unification constraints (akin to S-Inst-Mono in our constraint solver).

Although this prototype began from a different perspective, the implementation problems and their solutions turned out to be strikingly similar to those in our first prototype (described in §6). This close correspondence naturally raises the question of whether partial type schemes could be enriched to cope with polymorphic recursion.

8 Related work

Suspended constraints. Suspending constraints that cannot be solved yet is not a novel idea: it is a standard approach for implementing unification in dependently typed systems. This goes back to Huet's algorithm for higher-order unification [Huet 1975] and pattern unification [Miller 1991] where flexible-flexible pairs are delayed until at least one side becomes rigid. Our contribution lies in combining constraint suspension with ML-style implicit polymorphism—largely absent from dependently typed systems—and in formulating a declarative constraint semantics.

Conditional constraints [Pottier 2000] also delay resolution, waiting until the top-level constructor of a type is known. They provide an **`if-then-else`**-like primitive, but differ crucially from our suspended constraints: in Pottier's system, an unresolved conditional constraint is considered satisfiable, whereas in ours, an unresolved suspended constraint is not. This difference forces our semantics to track what is *known* in a context. Consequently, unresolved conditional constraints may enter a generalized type scheme as a form of qualified types, while our suspended constraints cannot. These semantic differences lead the two approaches to address very different user-facing type system features.

OutsideIn [Schrijvers, Jones, Sulzmann and Vytiniotis 2009] is a type system for GADTs that introduces *delayed implications* of the form $[\bar{\alpha}](\forall \bar{\beta}.\, C_1 \Rightarrow C_2)$. Constraint solving for delayed implications proceeds in two steps; solving simple constraints first and then solving delayed implications. The deferral ensures that inference for GADT match branches occurs when more is known about the scrutinee and expected return type from the context. To ensure principality,

[17] Although the undecidability of semi-unification already implied the existence of such examples, the specific class of examples of non-terminating cases was only characterized later by Figueiredo and Camarao [2004]. These examples use extremely complex patterns, suggesting that they are unlikely to appear in practice, unless crafted deliberately.

OutsideIn enforces an algorithmic restriction: the variables $\bar{\alpha}$ must already be instantiated to concrete type constructors before they may be unified by the implication's conclusion C_2. This ensures information only flows from the outside into the implication's conclusion. Notably, they do give a declarative specification for this restriction, using an elegant but mysterious quantification on all possible ways to type the context outside the GADT clauses. Using our new perspective on *known* type information, we can say that their semantics enforces that only *known* information from outside GADT clauses can be used inside. Later work on OutsideIn, namely OutsideIn(X) [Vytiniotis, Jones, Schrijvers and Sulzmann 2011], argues that delayed implication constraints make local let-generalization all but unmanageable, both in theory and implementation. Their proposed fix is to abandon local let-generalization altogether. By contrast, our work shows that the difficult interactions between let-generalization and suspended constraints can be resolved. Furthermore, OutsideIn(X) forgoes a declarative specification complete with respect to its inference algorithm, on the grounds that such a specification would be "as complicated and hard to understand as the [inference] algorithm". We believe that our *omnidirectional recipe* could provide a declarative specification: one capable of being principal and complete for GADTs, and we would be interested in studying this application.

Directionality. The fixed directionality of bidirectional type systems is a known limitation. Prior work has sought to mitigate this within the bidirectional setting by altering the order in which information flows. For instance, Xie and d. S. Oliveira [2018]'s *let arguments go first* typechecks applications $e_1\ e_2$ by checking the argument e_2 before the function e_1, allowing information to flow from argument to function (*cf.*, Chk-App).

More recent work on *contextual typing* [Xue, Cui, Jiang and d. S. Oliveira 2026; Xue and d. S. Oliveira 2024] goes further by deferring the commitment between Syn-App and Chk-App by parameterizing typing $\Gamma \vdash_m e : \tau$ with contextual information m. This enables typing multiple arguments from right to left when sufficient contextual information is available, and thus successfully typechecks programs such as $\texttt{ex}_6$ from §2.3. Nevertheless, it still enforces a fixed order of propagation and, consequently, some well-typed programs are rejected as ill-typed (*e.g.* $\texttt{ex}_{62}$, $\texttt{ex}_{63}$).[18]

Omnidirectionality removes this restriction, allowing information to flow without committing to a fixed propagation order. The contextual information m (or *mask*) of contextual typing can be viewed as a *lossy* abstraction of the surrounding context $\mathscr{E}$ in our system: omnidirectionality preserves $\mathscr{E}$ faithfully and exploits the information that the mask m forgets, propagating more type information and accepting more well-typed programs.

However, contextual typing can continue to guide inference even when no contextual information is available (*i.e.*, $m = \blacksquare$). In particular, *contextual subtyping* may default to reflexivity:

$$\frac{\textsc{DS-Refl}}{\Gamma \vdash_{\blacksquare} A \leq A}$$

Within inference, this amounts to resolving subtyping with unification. This mechanism is analogous to the idea of *default rules* (§2.2) in our setting.

Because contextual typing still fixes a propagation order, this default can be forced too early. The following program, in the contextual System-F calculus of Xue, Cui, Jiang and d. S. Oliveira [2026] (F^c), illustrates the issue:

```
let choose : α. α → α → α = fun x y → x
let ex₁₂ = (choose id : ((α. α → α) → α. α → α) → _)                    Fᶜ ✗
```

[18] Readers wishing to verify that $\texttt{ex}_{62}$ and $\texttt{ex}_{63}$ are rejected under contextual typing must inline app and rev_app.

This program is rejected because id must be inferred ($m = \blacksquare$) before the outer annotation can guide the instantiation of choose. With no contextual information available, this instantiation of id must proceed via DS-Refl. As a result, choose is instantiated with $\alpha.\ \alpha \to \alpha$, yielding a type incomparable with the annotation.

Finally, comparisons between omnidirectionality and contextual typing are not quite apples-to-apples. Contextual typing targets intersection types and first-class polymorphism, whereas omnidirectionality has focused on static overloading and semi-explicit first-class polymorphism. Since important extensions of omnidirectionality, including default rules and first-class polymorphism, remain future work, direct expressiveness comparisons are premature.

Overloading. Qualified types [Jones 1995a], best known for their use in Haskell's type-classes, are related to our suspended match constraints: both represent constraints on types or type variables that are delayed. At generalization time, constraints on generalizable variables are retained in the type scheme, yielding a *constrained type scheme* $\forall\bar{\alpha}.\, C \Rightarrow \tau$. This is much simpler to implement than our partial type schemes, but it provides a different behavior: each instance may resolve C differently (as the constraint is copied on instantiation). Qualified types are excellent choice when this is the desired behavior, typically for *dynamic overloading* [Wadler and Blott 1989]. But they are insufficient when we require a unique resolution of the constraint across all instances—as in *static overloading*. Static overloading can be emulated using qualified types by applying implementation-defined tweaks to specific built-in type-classes, enforcing that resolution is global and failing when it is under-determined. However, such mechanisms lack a clear declarative semantics: to our knowledge, capturing failure due to missing information requires our unicity conditions.

Leijen and Ye [2025] recently proposed a bidirectional account of generalized static overloading within ML. However, their approach is limited by its reliance on fixed directionality (§2.3). Variational typechecking [Chen, Erwig and Walkingshaw 2014] was originally developed for reasoning about well-typed CPP #ifdef-style macros, introducing *choices* $a\langle e_1, e_2\rangle$, where a is a *dimension* with *alternatives* e_1 and e_2. Once dimensions are fixed, we are able to project a well-typed non-variational program. Beneš and Brachthäuser [2025] apply this machinery to recast static overloading as variational typing, with a resolution algorithm that uniquely selects the dimensions. However, their system removes *local let-generalization* and requires an exponential-time resolution procedure—an unavoidable consequence of the NP-hardness of *general* static overloading [Charguéraud, Bodin, Dunfield and Riboulet 2025].

Partial type schemes provide an alternative that preserves ML's local let-generalization while suspended constraints offer a tractable account of static overloading. By enforcing resolution using *known* type information (captured by our novel unicity condition) rather than *guessed* information, our approach remains tractable. Our experience suggests that this is a "goldilocks" solution: expressive enough for most applications, yet tractable, and (crucially) compatible with ML's *local let-generalization*. However, we have not yet extended our approach to capture *generalized static overloading*; developing a full omnidirectional account of that setting is left to future work.

First-class polymorphism. Polytypes provide a form of *semi-explicit* first-class polymorphism and are of particular interest due to their role in OCaml's inference of polymorphic methods and polymorphic function parameters. Fully implicit first-class polymorphism is more expressive than polytypes and other forms of boxed polymorphism. In the latter, the programmer must explicitly manage the boxing and unboxing of terms (and types) for generalization and instantiation. However, making these operations implicits shifts the burden to inference, which must determine when to introduce and eliminate these polytypes. It remains unclear whether such inference can be achieved using omnidirectionality *without* default rules (§2.2)—a mechanism also used in bidirectional systems (§8).

Many systems for implicit first-class polymorphism exist; here we highlight a few in the context of ML. $\mathsf{ML^F}$ [Le Botlan and Rémy 2009] is an extension of ML that supports first-class polymorphism that goes beyond the power of System F, while retaining type inference with principal types. It is a generalization of polytypes, relying on π-directionality, but it remains unclear how to effectively scale $\mathsf{ML^F}$ to the rest of OCaml's features. Nevertheless, it would be interesting to explore whether the directional sensitivity of $\mathsf{ML^F}$ could be removed by omnidirectionality.

FreezeML [Emrich, Lindley, Stolarek, Cheney and Coates 2020] is an impredicative type inference system in which polymorphic variables can be *frozen*, written $\lceil x \rceil$, and only allowing instantiation on ordinary (unfrozen) variables x. Unlike polytypes, FreezeML permits higher-rank types directly in the syntax of types, though these can be encoded back into a minor extension of polytypes.[19] The essential difference is that generalization of higher-rank types is implicit, inferring the most-general type, whereas polytypes require an explicit type annotation $[e : \exists \bar{\alpha}.\, \sigma]$.

QuickLook [Serrano, Hage, Jones and Vytiniotis 2020], Haskell's latest approach at first-class polymorphism, uses bidirectional propagation to take a "quick look" at the spine of an application to guide the instantiation of higher-rank functions. This permits *argument reordering*, allowing later arguments to influence the type of earlier ones within the same application spine. While this mitigates the order-dependence of fixed directionality (§2.3), it still enforces a global direction of type propagation between surrounding constructs (*e.g.* let-bindings). In this sense, QuickLook is *locally omnidirectional* but *globally directional*: they reduce order sensitivity within applications, but do not provide a declarative account of type flow between arbitrary constructs, as required for omnidirectionality. More recently, Frost [Tang, Jiang, Oliveira and Lindley 2025] follows a similar line but extends it to handle η-expansion and introduces a freezing operator, akin to that of FreezeML, that helps propagate type information when the default order of inference is insufficient.

Type-based disambiguation in SML. In SML, tuples are treated as structural records with numeric labels, and projections such as #1 or #2 are type-directed. This relies on row typing: if e has the type $\{j = \tau_j; \varrho\}$, where ϱ is a row describing the remaining tuple fields, then $e.j$ has type τ_j. The same mechanism also underlies record projections.

However, SML does not support row-polymorphic definitions: restricting to monomorphic rows allows a simple yet efficient compilation strategy for records and tuples. To enforce this, SML adds a prose side-condition to the typing rules requiring that each row variable be fully determined by the "program context" [Milner, Harper, MacQueen and Tofte 1997, Section 4.11]. Our unicity conditions provides a precise, declarative specification for this informal restriction, filling a gap in the original specification.

Rossberg [2008, page 36] remarks that this restriction interacts poorly with `let`-polymorphism and therefore not supported in practice:

> *Under item 1 the Definition states that "the program context" must determine the exact type of flexible records, but it does not specify any bounds on the size of this context. Unlimited context is clearly infeasible since it is incompatible with let polymorphism: at the point of generalization the structure of a type must be determined precisely enough to know what we have to quantify over. We thus restrict the context for resolving flexible records to the innermost surrounding value declaration, as most other SML systems seem to do as well. This is in par with our treatment of overloading (see 5.8).*

[19]Each let-binding let $x = e_1$ in e_2 introduces a boxed polytype let $x = [e_1]$ in e_2, with each variable occurrence of x implicitly unboxing it $\langle x \rangle$. Freezing disables this implicit unboxing. By default, the implicit boxing operator $[e_1]$ infers a polytype $[\sigma]$, where σ is the most general type of e_1 when it is not already known from the surrounding context.

We have solved this difficult interaction between disambiguation and let-polymorphism. In particular, we implemented tuples in our prototype and formalize their meta-theory in Appendix §A; the system behaves as expected.

We also remark that Poly/ML [Matthews 2005] goes further than other SML implementations on this front, supporting examples that even rely on backpropagation:

```
let fun fst r = #1 r in (fst (1, 2), fst (true, false)) end;   Poly/ML ✓  OmniML ✓
```

Record field overloading in GHC. Haskell 98 derives selector functions for each record field, an approach that precludes declaring two record types with the same field names within a single module. GHC 6.8.1 (2007) introduced the `DisambiguateRecordField` extension for type-directed disambiguation of record labels,[20] and GHC 8.0.1 (2016) added `DuplicateRecordFields`, allowing distinct record types with overlapping field names in the same module.

However, the GHC developers found that type-based disambiguation using bidirectional type-checking is not sufficiently predictable in practice for users [Gundry 2017], and that the reliance on projection functions, rather than a dedicated projection syntax, further complicated the implementation by making all function calls potentially ambiguous. As a result, GHC is gradually moving away from type-based disambiguation towards an approach based on qualified types constraints of the form `HasField "x" a b` (`OverloadedRecordDot`, 2017).[21]

For instance, the projection $\mathtt{ex}_1$ from §2.1 would be written as:

```
ex₁ :: HasField "x" a b => a → b                    GHC ✓  OmniML ✗
ex₁ r = getField @"x" r
```

It is worth emphasizing that OmniML rejects this program. Since OmniML implements *static* overloading, it requires a unique resolution for each overloaded occurrence; as a result, programs such as $\mathtt{ex}_1$, whose projection is ambiguous, are ill-typed. In contrast, a type-class-based solution—*i.e.*, one using qualified types—implements *dynamic* rather than *static* overloading, allowing each instantiation to resolve the field independently. This distinction has practical consequences: dynamic overloading is able to naturally support field-polymorphic programs such as $\mathtt{ex}_1$, but it does not guarantee efficient code generation (though GHC is often able to optimize type-class programs effectively).

Beyond efficiency, the `HasField` approach is also limited in expressiveness. It works well for records with 'simple' field types (*i.e.*, field types with *uniform* typing rules for projections), but it does not scale to field types where the typing rule itself depends on type-directed disambiguation—such as fields containing polymorphic types or GADT existentials. In contrast, suspended match constraints naturally accommodate such non-uniformity in typing rules.

9 Conclusions

We presented a constraint-based framework for omnidirectional type inference, scaled to ML with *local let-generalization*. Central to our approach is a new declarative account of when a type is *known* from the context, rather than *guessed*. Our constraint solver is omnidirectional: constraints may be solved in *any way*, enabled by partial type schemes. Through two instantiations of our *omnidirectional recipe*, we obtained a sound, complete, and *principal* type inference algorithm—in short, principality held *anyway*, precisely because of omnidirectionality.

[20] https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/disambiguate_record_fields.html
[21] https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/hasfield.html

Future work. We aim to extend our framework to support more advanced features. One direction is generalized *static overloading*; another is *implicit first-class polymorphism*. We also plan to investigate *default rules*—a mechanism where ambiguity is resolved by falling back on a default, non-principal choice *e.g.*OCaml selects the most recent matching record type in scope for ambiguous field names.

Acknowledgments

Making of. The unsatisfactory situation of type-directed disambiguation in OCaml has been on our mind for years; a first concrete step to which this work can be traced is a specific example[22] of inconvenient order of propagation (between pattern and expression in a let-binding) presented by Andreas Rossberg as part of general feedback on the use of OCaml in the `wasm` reference interpreter [Rossberg 2016]. Gabriel Scherer proposed delaying the resolution of ambiguous constraints and advised Olivier Martinot as an intern on this topic in Summer 2020 [Martinot and Scherer 2021]. Together, they identified the implementation and specification difficulties; their implementation would only handle simple cases where no incremental generalization was necessary (in particular, it did not support backpropagation), and they did not have a precise declarative specification. Gabriel Scherer kept working on this infrequently until 2024, identified the need for incremental generalization and instantiation, but its implementation and declarative specifications remained incomplete.

Alistair O'Brien started working on suspended constraints in Fall 2024, coming from work on constraint-based presentations of GADT checking—suspended constraints could delay the checking of GADT patterns until the equalities to be introduced are known to be between rigid types. He proposed an alternative declarative semantics, and implemented a working prototype of incremental instantiation, described in the present paper.

Didier Rémy started working on suspended constraints in Winter 2024–2025, coming from questions of principal type inference in the presence of modular implicits, in collaboration with Samuel Vivien. Didier Rémy implemented a second prototype for incremental instantiation based on semi-unification, which is fairly different from the prototype described in the present paper, and brought the idea of handling polytypes as suspended constraints.

All co-authors contributed to the writing and metatheory, with Alistair O'Brien as the main contributor at every step.

Thanks. We thank the anonymous reviewers for their thoughtful feedback and questions, and Simon Peyton-Jones, whose insightful remarks, we hope, helped us make this work more approachable. We are also grateful to Adam Gundry for clarifying aspects of record-field disambiguation in GHC, and to Jeremy Yallop for his sage guidance throughout this project. Finally, we thank Brendan Coll, Daniel Gooding, Micheal Lee, and François Pottier for their helpful comments on earlier drafts. This work was partially funded by the OCaml Software Foundation.

References

Jiří Beneš and Jonathan Immanuel Brachthäuser. 2025. The Simple Essence of Overloading. (2025).

Ishan Bhanuka, Lionel Parreaux, David Binder, and Jonathan Immanuel Brachthäuser. 2023. Getting into the Flow: Towards Better Type Error Messages for Constraint-Based Type Inference. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 431–459. https://doi.org/10.1145/3622812

Arthur Charguéraud, Martin Bodin, Jana Dunfield, and Louis Riboulet. 2025. Typechecking of Overloading. In *Journées Francophones des Langages Applicatifs*.

Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2014. Extending Type Inference to Variational Programs. *ACM Transactions on Programming Languages and Systems* 36, 1 (March 2014), 1–54. https://doi.org/10.1145/2518190

[22] https://github.com/ocaml/ocaml/issues/7388

Luis Damas and Robin Milner. 1982. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Albuquerque, New Mexico) *(POPL '82)*. Association for Computing Machinery, New York, NY, USA, 207–212. https://doi.org/10.1145/582153.582176

Jana Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ICFP*. https://doi.org/10.1145/2500365.2500582

Jana Dunfield and Frank Pfenning. 2004. Tridirectional typechecking. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, Neil D. Jones and Xavier Leroy (Eds.). ACM, 281–292. https://doi.org/10.1145/964001.964025

Frank Emrich, Sam Lindley, Jan Stolarek, James Cheney, and Jonathan Coates. 2020. FreezeML: complete and easy type inference for first-class polymorphism. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 423–437. https://doi.org/10.1145/3385412.3386003

Lucilia Figueiredo and Carlos Camarao. 2004. Semi-unification and Periodicity of Turing Machines. (2004). https://www.researchgate.net/profile/Carlos-Camarao/publication/268426611_Semi-unification_and_Periodicity_of_Turing_Machines

Jacques Garrigue. 1998. Programming with polymorphic variants. In *ML workshop*, Vol. 13. Baltimore.

Jacques Garrigue and Didier Rémy. 1999. Extending ML with Semi-Explicit Higher-Order Polymorphism. *Information and Computation* 155, 1/2 (1999), 134–169. http://www.springerlink.com/content/m303472288241339/ A preliminary version appeared in TACS'97.

Jacques Garrigue and Didier Rémy. 2013. Ambivalent Types for Principal Type Inference with GADTs. In *Programming Languages and Systems - 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8301)*, Chung-chieh Shan (Ed.). Springer, 257–272. https://doi.org/10.1007/978-3-319-03542-0_19

Adam Gundry. 2017. Overloaded Record Fields. https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0023-overloaded-record-fields.rst GHC proposal/RFC.

Fritz Henglein. 1989. *Polymorphic Type Inference and Semi-Unification*. PhD. New York University.

Fritz Henglein. 1993. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.* 15, 2 (April 1993), 253–289. https://doi.org/10.1145/169701.169692

Roger Hindley. 1969. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society* 146 (1969), 29–60.

Gérard Huet. 1975. A unification algorithm for typed λ-calculus. *Theoretical Computer Science* 1, 1 (1975).

Mark P. Jones. 1995a. *Qualified types: theory and practice*. Cambridge University Press.

Mark P. Jones. 1995b. A System of Constructor Classes: Overloading and Implicit Higher-Order Polymorphism. *J. Funct. Program.* 5, 1 (1995), 1–35. https://doi.org/10.1017/S0956796800001210

A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. 1993. The Undecidability of the Semi-unification Problem. *Information and Computation* 102, 1 (1993), 83–101. https://doi.org/10.1006/inco.1993.1003

Didier Le Botlan and Didier Rémy. 2009. Recasting MLF. *Information and Computation* 207, 6 (2009), 726–785. https://doi.org/10.1016/j.ic.2008.12.006

Oukseh Lee and Kwangkeun Yi. 1998. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems* 20, 4 (July 1998), 707–723. https://doi.org/10.1145/291891.291892

Daan Leijen and Wenjia Ye. 2025. Principal Type Inference under a Prefix. In *PLDI'25*. ACM, 1–24. https://www.microsoft.com/en-us/research/publication/principal-type-inference-under-a-prefix/ Backup Publisher: ACM SIGPLAN.

Brad Lushman and Gordon V Cormack. 2008. *Constraint-Based Typing for ML via Semiunification*. Technical Report CS-2008-10. University of Waterloo, Ontario, Canada. 49 pages.

Olivier Martinot and Gabriel Scherer. 2021. Frozen inference constraints for type-directed disambiguation. In *ML Family Workshop*. online, South Korea. https://inria.hal.science/hal-03510890

David Matthews. 2005. Poly/ML. https://github.com/polyml/polyml version 4.1.

Dale Miller. 1991. Unification of Simply Typed Lamda-Terms as Logic Programming. In *Logic Programming, Proceedings of the Eigth International Conference, Paris, France, June 24-28, 1991*, Koichi Furukawa (Ed.). MIT Press, 255–269.

Robin Milner, Robert Harper, David MacQueen, and Mads Tofte. 1997. *The Definition of Standard ML*. The MIT Press. https://doi.org/10.7551/mitpress/2319.001.0001

Martin Odersky and Konstantin Läufer. 1996. Putting Type Annotations to Work. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, Hans-Juergen Boehm and Guy L. Steele Jr. (Eds.). ACM Press, 54–67. https://doi.org/10.1145/237721.237729

Martin Odersky, Martin Sulzmann, and Martin Wehr. 1999. Type inference with constrained types. *Theory and Practice of Object Systems* 5, 1 (1999), 35–55. https://doi.org/10.1002/(SICI)1096-9942(199901/03)5:1<35::AID-TAPO4>3.0.CO;2-4

Atsushi Ohori. 1995. A Polymorphic Record Calculus and Its Compilation. *ACM Trans. Program. Lang. Syst.* 17, 6 (1995), 844–895. https://doi.org/10.1145/218570.218572

Benjamin C. Pierce and David N. Turner. 1998. Local Type Inference. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, David B. MacQueen and Luca Cardelli (Eds.). ACM, 252–265. https://doi.org/10.1145/268946.268967

François Pottier. 2000. A Versatile Constraint-Based Type Inference System. *Nord. J. Comput.* 7, 4 (2000), 312–347.

François Pottier. 2014. Hindley-Milner elaboration in applicative style. In *ICFP*. http://cambium.inria.fr/~fpottier/publis/fpottier-elaboration.pdf

François Pottier. 2018. Inferno. A library for constraint-based Hindley-Milner type inference. https://gitlab.inria.fr/fpottier/inferno Available on opam https://opam.ocaml.org/.

François Pottier and Didier Rémy. 2005. The Essence of ML Type Inference. In *Advanced Topics in Types and Programming Languages*, Benjamin C. Pierce (Ed.). MIT Press, Chapter 10, 389–489. https://pauillac.inria.fr/~remy/attapl/

Didier Rémy. 1989. Type checking records and variants in a natural extension of ML. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '89)*. Association for Computing Machinery, New York, NY, USA, 77–88. https://doi.org/10.1145/75277.75284

Didier Rémy. 1990. *Algèbres Touffues. Application au Typage Polymorphe des Objets Enregistrements dans les Langages Fonctionnels.* Thèse de doctorat. Université de Paris 7.

Didier Rémy. 1992. *Extending ML Type System with a Sorted Equational Theory.* Research Report 1766. Institut National de Recherche en Informatique et Automatisme, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France.

Didier Rémy and Jerome Vouillon. 1997. Objective ML: A Simple Object-Oriented Extension of ML. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, Peter Lee, Fritz Henglein, and Neil D. Jones (Eds.). ACM Press, 40–53. https://doi.org/10.1145/263699.263707

Didier Rémy and Boris Yakobowski. 2008. Efficient Type Inference for the MLF Language: a graphical and constraints-based approach. In *The 13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*. Victoria, BC, Canada, 63–74. https://doi.org/10.1145/1411203.1411216

Andreas Rossberg. 2008. HaMLet: To Be Or Not To Be Standard ML. https://people.mpi-sws.org/~rossberg/hamlet/hamlet-1.3.1.pdf hamlet manual, version 1.3.1.

Andreas Rossberg. 2016. WebAssembly: high speed at low cost for everyone. presented at the ML Family Workshop.

Tom Schrijvers, Simon L. Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. 2009. Complete and decidable type inference for GADTs. In *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*, Graham Hutton and Andrew P. Tolmach (Eds.). ACM, 341–352. https://doi.org/10.1145/1596550.1596599

Alejandro Serrano, Jurriaan Hage, Simon Peyton Jones, and Dimitrios Vytiniotis. 2020. A quick look at impredicativity. *Proc. ACM Program. Lang.* 4, ICFP (2020), 89:1–89:29. https://doi.org/10.1145/3408971

Wenhao Tang, Shengyi Jiang, Bruno C. d. S. Oliveira, and Sam Lindley. 2025. Freezing Bidirectional Typing. presented at the ML Family Workshop.

Robert Endre Tarjan. 1975. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM* 22, 2 (1975), 215–225. https://doi.org/10.1145/321879.321884

Dimitrios Vytiniotis, Simon L. Peyton Jones, Tom Schrijvers, and Martin Sulzmann. 2011. OutsideIn(X) Modular type inference with local assumptions. *J. Funct. Program.* 21, 4-5 (2011), 333–412. https://doi.org/10.1017/S0956796811000098

Philip Wadler and Stephen Blott. 1989. How to Make ad-hoc Polymorphism Less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*. ACM Press, 60–76. https://doi.org/10.1145/75277.75283

Mitchell Wand. 1989. Type Inference for Record Concatenation and Multiple Inheritance. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*. IEEE Computer Society, 92–97. https://doi.org/10.1109/LICS.1989.39162

Leo White. 2023. Semi-explicit polymorphic parameters. (sep 2023). Presentation at the Higher-order, Typed, Inferred, Strict: ML Family workshops.

Leo White, Frédéric Bour, and Jeremy Yallop. 2014. Modular implicits. In *Proceedings ML Family/OCaml Users and Developers workshops, ML/OCaml 2014, Gothenburg, Sweden, September 4-5, 2014. (EPTCS, Vol. 198)*, Oleg Kiselyov and Jacques Garrigue (Eds.). 22–63. https://doi.org/10.4204/EPTCS.198.2

Ningning Xie and Bruno C. d. S. Oliveira. 2018. Let Arguments Go First. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science)*, Amal Ahmed (Ed.). Springer, 272–299. https://doi.org/10.1007/978-3-319-89884-1_10

Xu Xue, Chen Cui, Shengyi Jiang, and Bruno C. d. S. Oliveira. 2026. Local Contextual Type Inference. *Proc. ACM Program. Lang.* 10, POPL (2026), 294–323. https://doi.org/10.1145/3776653

Xu Xue and Bruno C. d. S. Oliveira. 2024. Contextual Typing. *Proc. ACM Program. Lang.* 8, ICFP (2024), 880–908. https://doi.org/10.1145/3674655

Organization of appendices

Reference appendix. §A gives a full reference for all definitions, grammars and figures in the paper, including all cases (even those omitted from the main paper for reasons of space).

Proof appendices. These appendices contain proofs for the formal claims in the article. They are typically written tersely.

- §B proves properties of the constraint language and its semantics. The main result is canonicalization, which morally establishes that uses of the contextual rule Match-Ctx can be "permuted down" in the proof until they are all at the bottom of the derivation, followed by a proof on a simple constraint.
- §C proves the correctness of the constraint solver with respect to the semantics.
- §D proves the properties about the OmniML type system, in particular the correctness of constraint generation.

A Full technical reference

This section repeats all the technical definitions mentioned in the paper, including the cases, rules, and definitions that were omitted from the main paper to save space. It can serve as a useful cheatsheet to understand a definition in full, or when studying the meta-theory of the system.

$$
\begin{array}{rcll}
\alpha, \beta, \gamma & \in & \mathcal{V} & \text{Type variables} \\
\tau & ::= & \alpha \mid 1 \mid \tau_1 \to \tau_2 \mid \Pi_{i=1}^{n} \tau_i \mid \mathsf{rcd}\ T\ \bar{\tau} \mid [\sigma] & \text{Types} \\
\sigma & ::= & \tau \mid \forall \alpha.\, \sigma & \text{Type schemes} \\
\mathfrak{g} & & & \text{Ground types} \\
\mathfrak{r} & ::= & (\mathfrak{g}, \phi) & \text{Ground region} \\
\mathfrak{S} & \subseteq & \mathcal{G} & \text{Sets of ground types} \\
\mathfrak{R} & \subseteq & \mathcal{R} & \text{Sets of ground regions} \\
C & ::= & \mathsf{true} \mid \mathsf{false} \mid C_1 \wedge C_2 \mid \exists \alpha.\, C \mid \forall \alpha.\, C \mid \tau_1 = \tau_2 & \text{Constraints} \\
 & \mid & \mathsf{let}\ x = \lambda \alpha.\, C_1\ \mathsf{in}\ C_2 \mid x\ \tau & \\
 & \mid & \mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi} & \\
 & \mid & \epsilon \mid \mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = C_1\ \mathsf{in}\ C_2 \mid \exists i^x.\, C \mid i(\alpha) \rightsquigarrow \tau & \\
 & \mid & t.\ell \leq \tau_1 \to \tau_2 \mid \mathsf{dom}\ t = \bar{\ell} \mid s \leq \tau \mid x \leq s & \\
\chi & ::= & \rho \Rightarrow C & \text{Branches} \\
\rho & ::= & \_ \mid \Pi\ \alpha_j \mid \mathsf{rcd}\ t\ \_ \mid [s] & \text{Shape patterns} \\
\phi & ::= & \emptyset \mid \phi[\alpha := \mathfrak{g}] \mid \phi[x := \mathfrak{S}] \mid \phi[x := \mathfrak{R}] \mid \phi[i := \phi'] & \text{Semantic environment} \\
U & ::= & \mathsf{true} \mid \mathsf{false} \mid U_1 \wedge U_2 \mid \exists \alpha.\, U \mid \epsilon & \text{Unification problems} \\
\epsilon & ::= & \emptyset \mid \tau = \epsilon & \text{Multi-equations} \\
\mathscr{C} & ::= & \square \mid \mathscr{C} \wedge C \mid C \wedge \mathscr{C} \mid \exists \alpha.\, \mathscr{C} \mid \forall \alpha.\, \mathscr{C} & \text{Constraint contexts} \\
 & \mid & \mathsf{let}\ x = \lambda \alpha.\, \mathscr{C}\ \mathsf{in}\ C \mid \mathsf{let}\ x = \lambda \alpha.\, C\ \mathsf{in}\ \mathscr{C} & \\
 & \mid & \mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = \mathscr{C}\ \mathsf{in}\ C \mid \mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = C\ \mathsf{in}\ \mathscr{C} \mid \exists i^x.\, \mathscr{C} & \\
\zeta & ::= & \nu \bar{\gamma}.\, \tau & \text{Shapes} \\
\varsigma & & & \text{Canonical principal shapes} \\
e & ::= & x \mid () \mid \lambda x.\, e \mid e_1\ e_2 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid (e : \exists \bar{\alpha}.\, \tau) & \text{Terms} \\
 & \mid & \{\overline{\ell = e}\} \mid e.\ell \mid T.\{\overline{\ell = e}\} \mid e.T.\ell & \\
 & \mid & (e_1, \ldots, e_n) \mid e.j \mid e.n.j & \\
 & \mid & [e] \mid [e : \exists \bar{\alpha}.\, \sigma] \mid \langle e \rangle \mid \langle e : \exists \bar{\alpha}.\, \sigma \rangle & \\
 & \mid & \square\ \mathsf{with}\ \bar{e} & \\
\mathscr{E} & ::= & \square \mid \mathscr{E}\ e \mid e\ \mathscr{E} \mid \lambda x.\, \mathscr{E} \mid \mathsf{let}\ x = \mathscr{E}\ \mathsf{in}\ e \mid \mathsf{let}\ x = e\ \mathsf{in}\ \mathscr{E} \mid (\mathscr{E} : \exists \bar{\alpha}.\, \tau) & \text{Term contexts} \\
 & \mid & \{\ell_1 = e_1\ \ldots\ \ell_i = \mathscr{E}\ \ldots\ \ell_n = e_n\} \mid \mathscr{E}.\ell & \\
 & \mid & T.\{\ell_1 = e_1\ \ldots\ \ell_i = \mathscr{E}\ \ldots\ \ell_n = e_n\} \mid \mathscr{E}.T.\ell & \\
 & \mid & (e_1, \ldots, \mathscr{E}, \ldots, e_n) \mid \mathscr{E}.j \mid \mathscr{E}.j^n & \\
 & \mid & [\mathscr{E}] \mid [\mathscr{E} : \exists \bar{\alpha}.\, \sigma] \mid \langle \mathscr{E} \rangle \mid \langle \mathscr{E} : \exists \bar{\alpha}.\, \sigma \rangle & \\
 & \mid & \square\ \mathsf{with}\ e_1, \ldots, \mathscr{E}, \ldots, e_n & \\
\Gamma & ::= & \emptyset \mid \Gamma, x : \sigma & \text{Typing contexts} \\
\Omega & ::= & \emptyset \mid \Omega, \ell : \forall \bar{\alpha}.\, \mathsf{rcd}\ T\ \bar{\alpha} \to \tau & \text{Label environment}
\end{array}
$$

$\boxed{\phi \vdash C}$ Under the environment ϕ, the constraint C is satisfiable.

$$\frac{}{\phi \vdash \mathsf{true}}\ \textsc{True} \qquad \frac{\phi \vdash C_1 \qquad \phi \vdash C_2}{\phi \vdash C_1 \land C_2}\ \textsc{Conj} \qquad \frac{\phi[\alpha := \mathfrak{g}] \vdash C}{\phi \vdash \exists \alpha.\, C}\ \textsc{Exists} \qquad \frac{\forall \mathfrak{g},\ \phi[\alpha := \mathfrak{g}] \vdash C}{\phi \vdash \forall \alpha.\, C}\ \textsc{Forall} \qquad \frac{\phi(\tau_1) = \phi(\tau_2)}{\phi \vdash \tau_1 = \tau_2}\ \textsc{Unif}$$

$$\frac{\phi[x := \mathfrak{S}] \vdash C_2 \qquad \mathfrak{S} = \phi(\lambda \alpha.\, C_1) \qquad \mathfrak{S} \neq \emptyset}{\phi \vdash \mathsf{let}\ x = \lambda \alpha.\, C_1\ \mathsf{in}\ C_2}\ \textsc{Let} \qquad \frac{\phi(\tau) \in \phi(x)}{\phi \vdash x\ \tau}\ \textsc{App}$$

$$\frac{\mathscr{C}[\tau\,!\,\varsigma] \qquad \phi \vdash \mathscr{C}[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}]}{\phi \vdash \mathscr{C}[\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}]}\ \textsc{Match-Ctx} \qquad \frac{\forall \tau \in \epsilon,\ \phi(\tau) = \mathfrak{g}}{\phi \vdash \epsilon}\ \textsc{Multi-Unif}$$

$$\frac{\phi[x := \mathfrak{R}] \vdash C_2 \qquad \mathfrak{R} = \phi(\lambda \alpha[\bar{\alpha}].\, C_1) \qquad \mathfrak{R} \neq \emptyset}{\phi \vdash \mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = C_1\ \mathsf{in}\ C_2}\ \textsc{LetR} \qquad \frac{(\phi(\tau), \_) \in \phi(x)}{\phi \vdash x\ \tau}\ \textsc{AppR}$$

$$\frac{\phi[i := \phi'] \vdash C \qquad (\_, \phi') \in \phi(x)}{\phi \vdash \exists i^x.\, C}\ \textsc{Exists-Inst} \qquad \frac{\phi(i)(\alpha) = \phi(\tau)}{\phi \vdash i(\alpha) \rightsquigarrow \tau}\ \textsc{Incr-Inst}$$

$$\begin{aligned}
T.\ell \leq \tau_1 \to \tau_2 &\triangleq \begin{cases} \exists \bar{\alpha}.\, \tau_1 = \mathsf{rcd}\ T\ \bar{\alpha} \land \tau_2 = \tau & \text{if } \Omega(T.\ell) = \forall \bar{\alpha}.\, \mathsf{rcd}\ T\ \bar{\alpha} \to \tau \\ \mathsf{false} & \text{otherwise} \end{cases} \\
\mathsf{dom}\ T = \bar{\ell} &\triangleq \begin{cases} \mathsf{true} & \text{if dom } (\Omega(T)) = \bar{\ell} \\ \mathsf{false} & \text{otherwise} \end{cases} \\
(\forall \bar{\alpha}.\, \tau') \leq \tau &\triangleq \exists \bar{\alpha}.\, \tau' = \tau \\
x \leq (\forall \bar{\alpha}.\, \tau) &\triangleq \forall \bar{\alpha}.\, x\ \tau \\
\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \rho \Rightarrow \bar{C} &\triangleq \begin{cases} \exists \bar{\alpha}.\, \tau = \varsigma\ \bar{\alpha} \land \theta(C_i) & \text{if } \rho_i \text{ matches } \varsigma\ \bar{\alpha} \hookrightarrow \theta \\ \mathsf{false} & \text{otherwise} \end{cases}
\end{aligned}$$

$$\phi(\lambda \alpha.\, C) \triangleq \{\mathfrak{g} \in \mathcal{G} : \phi[\alpha := \mathfrak{g}] \vdash C\}$$

$$\phi(\lambda \alpha[\bar{\alpha}].\, C) \triangleq \{(\mathfrak{g}, \phi[\alpha := \mathfrak{g}, \bar{\alpha} := \bar{\mathfrak{g}}]) \in \mathcal{R} : \phi[\alpha := \mathfrak{g}, \bar{\alpha} := \bar{\mathfrak{g}}] \vdash C\}$$

$$\mathscr{C}[\tau\,!\,\varsigma] \triangleq \forall \phi, \mathfrak{g}.\ \phi \vdash \lfloor \mathscr{C}[\tau = \mathfrak{g}] \rfloor \implies \mathsf{shape}(\mathfrak{g}) = \varsigma$$

$\boxed{\zeta \preceq \zeta'}$ The shape ζ' is an instance of ζ. Alternatively, ζ' is more general than ζ.

$$\frac{}{\nu \bar{\gamma}_1.\, \tau \preceq \nu \bar{\gamma}_2.\, \tau[\bar{\gamma}_1 := \bar{\tau}_1]}\ \textsc{Inst-Shape}$$

We write $\perp$ for the trivial shape $\nu\gamma.\,\gamma$. $\mathcal{S}$ denotes the set of shapes and $\mathcal{S}^*$ is the set of non-trivial shapes.

Definition A.1. A non-trivial shape $\zeta \in \mathcal{S}^*$ is the principal shape of the type τ iff:

(1) $\exists \bar{\tau}',\ \tau = \zeta\ \bar{\tau}'$
(2) $\forall \zeta' \in \mathcal{S}^*, \forall \bar{\tau}',\ \tau = \zeta'\ \bar{\tau}' \implies \zeta \preceq \zeta'$

A principal shape $\nu\bar{\gamma}.\,\tau$ is *canonical* if the sequence of its free variables $\bar{\gamma}$ appear in the order in which the variables occur in τ. $\text{shape}(\tau)$ is the canonical principal shape of τ.

$\boxed{\rho \text{ matches } \varsigma\ \bar{\gamma} \hookrightarrow \theta}$ The pattern ρ matches the shape ς with fresh components $\bar{\gamma}$ binding pattern variables in θ.

$$\begin{array}{rcll}
\Pi\ \alpha_j \text{ matches } (\nu\bar{\gamma}.\,\Pi_{i=1}^{n}\bar{\gamma})\ \bar{\gamma}' & \triangleq & [\alpha := \gamma'_j] & \text{if } n \geq j \\
\text{rcd } t\ _ \text{ matches } (\nu\bar{\gamma}.\,\text{rcd } T\ \bar{\gamma})\ \bar{\gamma}' & \triangleq & [t := T] & \\
[s] \text{ matches } (\nu\bar{\gamma}.\,[\sigma])\ \bar{\gamma}' & \triangleq & [s := \sigma[\bar{\gamma} := \bar{\gamma}']] &
\end{array}$$

$\boxed{C \text{ simple}}$ The constraint C is simple.

$$\textsc{Simple-True}\ \frac{}{\text{true simple}} \qquad \textsc{Simple-False}\ \frac{}{\text{false simple}} \qquad \textsc{Simple-Conj}\ \frac{C_1 \text{ simple} \qquad C_2 \text{ simple}}{C_1 \wedge C_2 \text{ simple}} \qquad \textsc{Simple-Exists}\ \frac{C \text{ simple}}{\exists\alpha.\,C \text{ simple}} \qquad \textsc{Simple-Forall}\ \frac{C \text{ simple}}{\forall\alpha.\,C \text{ simple}}$$

$$\textsc{Simple-Unif}\ \frac{}{\tau_1 = \tau_2 \text{ simple}} \qquad \textsc{Simple-Let}\ \frac{C_1 \text{ simple} \qquad C_2 \text{ simple}}{\text{let } x = \lambda\alpha.\,C_1 \text{ in } C_2 \text{ simple}} \qquad \textsc{Simple-App}\ \frac{}{x\ \tau \text{ simple}} \qquad \textsc{Simple-Multi-unif}\ \frac{}{\epsilon \text{ simple}}$$

$$\textsc{Simple-LetR}\ \frac{C_1 \text{ simple} \qquad C_2 \text{ simple}}{\text{let } x\ \alpha\ [\bar{\alpha}] = C_1 \text{ in } C_2 \text{ simple}} \qquad \textsc{Simple-Exists-Inst}\ \frac{C \text{ simple}}{\exists i^x.\,C \text{ simple}} \qquad \textsc{Simple-Incr-Inst}\ \frac{}{i(\alpha) \rightsquigarrow \tau \text{ simple}}$$

We write $\phi \vdash_{\text{simple}} C$ for the satisfiability of a simple constraint C.

$\boxed{\mathscr{C} \text{ simple}}$ The constraint context $\mathscr{C}$ is simple.

$$\textsc{Simple-Ctx-Hole}\ \frac{}{\square \text{ simple}} \qquad \textsc{Simple-Ctx-Conj-Left}\ \frac{\mathscr{C} \text{ simple} \qquad C \text{ simple}}{\mathscr{C} \wedge C \text{ simple}} \qquad \textsc{Simple-Ctx-Conj-Right}\ \frac{\mathscr{C} \text{ simple} \qquad C simple}{C \wedge \mathscr{C} \text{ simple}}$$

$$\textsc{Simple-Ctx-Exists}\ \frac{\mathscr{C} \text{ simple}}{\exists\alpha.\,\mathscr{C} \text{ simple}} \qquad \textsc{Simple-Ctx-Forall}\ \frac{\mathscr{C} \text{ simple}}{\forall\alpha.\,\mathscr{C} \text{ simple}} \qquad \textsc{Simple-Ctx-Let-Abs}\ \frac{\mathscr{C} \text{ simple} \qquad C \text{ simple}}{\text{let } x = \lambda\alpha.\,\mathscr{C} \text{ in } C \text{ simple}}$$

$$\textsc{Simple-Ctx-Let-In}\ \frac{C \text{ simple} \qquad \mathscr{C} \text{ simple}}{\text{let } x = \lambda\alpha.\,C \text{ in } \mathscr{C} \text{ simple}} \qquad \textsc{Simple-Ctx-Exists-Inst}\ \frac{\mathscr{C} \text{ simple}}{\exists i^x.\,\mathscr{C} \text{ simple}}$$

$\boxed{\lfloor C \rfloor}$ The erasure of C.

$$
\begin{array}{rcl}
\lfloor \mathsf{true} \rfloor & \triangleq & \mathsf{true} \\
\lfloor \mathsf{false} \rfloor & \triangleq & \mathsf{false} \\
\lfloor C_1 \wedge C_2 \rfloor & \triangleq & \lfloor C_1 \rfloor \wedge \lfloor C_2 \rfloor \\
\lfloor \exists \alpha. C \rfloor & \triangleq & \exists \alpha. \lfloor C \rfloor \\
\lfloor \forall \alpha. C \rfloor & \triangleq & \forall \alpha. \lfloor C \rfloor \\
\lfloor \tau_1 = \tau_2 \rfloor & \triangleq & \tau_1 = \tau_2 \\
\lfloor \mathsf{let}\ x = \lambda \alpha. C_1\ \mathsf{in}\ C_2 \rfloor & \triangleq & \mathsf{let}\ x = \lambda \alpha. \lfloor C_1 \rfloor\ \mathsf{in}\ \lfloor C_2 \rfloor \\
\lfloor x\ \tau \rfloor & \triangleq & x\ \tau \\
\lfloor \mathsf{match}\ \tau\ \mathsf{with}\ \bar{\rho} \Rightarrow \bar{C} \rfloor & \triangleq & \mathsf{true} \\
\lfloor \epsilon \rfloor & \triangleq & \epsilon \\
\lfloor \mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = C_1\ \mathsf{in}\ C_2 \rfloor & \triangleq & \mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = \lfloor C_1 \rfloor\ \mathsf{in}\ \lfloor C_2 \rfloor \\
\lfloor \exists i^x. C \rfloor & \triangleq & \exists i^x. \lfloor C \rfloor \\
\lfloor i(\alpha) \rightsquigarrow \tau \rfloor & \triangleq & i(\alpha) \rightsquigarrow \tau
\end{array}
$$

$\boxed{\phi \Vdash C}$ Under the semantic environment ϕ, the constraint C is canonically satisfiable.

$$
\textsc{Can-Simple}\ \frac{\phi \vdash_{\mathsf{simple}} C}{\phi \Vdash C}
\qquad
\textsc{Can-Match-Ctx}\ \frac{\mathscr{C}[\tau \,!\, \varsigma] \qquad \phi \Vdash \mathscr{C}[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}]}{\phi \Vdash \mathscr{C}[\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}]}
$$

$\boxed{T.\ell \leq \tau_1 \longrightarrow \tau_2}$ The label ℓ of the record T has the field type τ_2 and record type τ_1.

$$
\textsc{Lab-Inst}\ \frac{\Omega(T.\ell) = \forall \bar{\alpha}. \mathsf{rcd}\ T\ \bar{\alpha} \longrightarrow \tau}{T.\ell \leq \mathsf{rcd}\ T\ \bar{\tau} \longrightarrow \tau[\bar{\alpha} := \bar{\tau}]}
$$

$\boxed{\ell \triangleright T}$ The label ℓ infers the unique record name T.

$\boxed{\bar{\ell} \blacktriangleright T}$ The *closed* set of labels $\bar{\ell}$ infer the unique record name T.

$$
\textsc{Lab-Uni}\ \frac{\begin{array}{c}\ell \in \mathsf{dom}\ (\Omega(T)) \\ \forall T',\ \ell \in \mathsf{dom}\ (\Omega(T')) \implies T = T'\end{array}}{\ell \triangleright T}
\qquad
\textsc{Labs-Uni}\ \frac{\begin{array}{c}\bar{\ell} = \mathsf{dom}\ (\Omega(T)) \\ \forall T',\ \mathsf{dom}\ (\Omega(T')) = \bar{\ell} \implies T = T'\end{array}}{\bar{\ell} \blacktriangleright T}
$$

$\boxed{\Gamma \vdash e : \sigma}$ Under the typing context Γ, the term e is assigned the type σ

$$
\textsc{Var}\ \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}
\qquad
\textsc{Fun}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x. e : \tau_1 \longrightarrow \tau_2}
\qquad
\textsc{App}\ \frac{\Gamma \vdash e_1 : \tau_1 \longrightarrow \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}
\qquad
\textsc{Unit}\ \frac{}{\Gamma \vdash () : 1}
$$

$$
\textsc{Annot}\ \frac{\Gamma \vdash e : \tau[\bar{\alpha} := \bar{\tau}]}{\Gamma \vdash (e : \exists \bar{\alpha}. \tau) : \tau[\bar{\alpha} := \bar{\tau}]}
\qquad
\textsc{Gen}\ \frac{\Gamma \vdash e : \sigma \qquad \alpha \# \Gamma}{\Gamma \vdash e : \forall \alpha. \sigma}
\qquad
\textsc{Inst}\ \frac{\Gamma \vdash e : \forall \alpha. \sigma}{\Gamma \vdash e : \sigma[\alpha := \tau]}
$$

Let
$$\frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau}$$

Tuple
$$\frac{(\Gamma \vdash e_i : \tau_i)_{i=1}^{n}}{\Gamma \vdash (e_1, \ldots, e_n) : \Pi_{i=1}^{n} \tau_i}$$

Proj-X
$$\frac{\Gamma \vdash e : \Pi_{i=1}^{n} \tau_i \qquad 1 \leq j \leq n}{\Gamma \vdash e.j^n : \tau_j}$$

Proj-I
$$\frac{\mathscr{E}[e \triangleright \nu \bar{\gamma}.\, \Pi_{i=1}^{n} \bar{\gamma}] \qquad \Gamma \vdash \mathscr{E}[e.j^n] : \tau}{\Gamma \vdash \mathscr{E}[e.j] : \tau}$$

Poly-X
$$\frac{\Gamma \vdash e : \sigma[\bar{\alpha} := \bar{\tau}]}{\Gamma \vdash [e : \exists \bar{\alpha}.\, \sigma] : [\sigma[\bar{\alpha} := \bar{\tau}]]}$$

Poly-I
$$\frac{\mathscr{E}[\square \triangleleft \nu \bar{\gamma}.\, [\sigma] \mid e] \qquad \Gamma \vdash \mathscr{E}[[e : \exists \bar{\gamma}.\, \sigma]] : \tau}{\Gamma \vdash \mathscr{E}[[e]] : \tau}$$

Use-X
$$\frac{\Gamma \vdash e : [\sigma][\bar{\alpha} := \bar{\tau}]}{\Gamma \vdash \langle e : \exists \bar{\alpha}.\, \sigma \rangle : \sigma[\bar{\alpha} := \bar{\tau}]}$$

Use-I
$$\frac{\mathscr{E}[e \triangleright \nu \bar{\gamma}.\, [\sigma]] \qquad \Gamma \vdash \mathscr{E}[\langle e : \exists \bar{\gamma}.\, \sigma \rangle] : \tau}{\Gamma \vdash \mathscr{E}[\langle e \rangle] : \tau}$$

Rcd-X
$$\frac{\bar{\ell} = \text{dom } (\Omega(T)) \qquad (T.\ell_i \leq \tau \to \tau_i)_{i=1}^{n} \qquad (\Gamma \vdash e_i : \tau_i)_{i=1}^{n}}{\Gamma \vdash T.\{\ell_1 = e_1 \,; \ldots ; \ell_n = e_n\} : \tau}$$

Rcd-Closed
$$\frac{\bar{\ell} \blacktriangleright T \qquad \Gamma \vdash T.\{\overline{\ell = e}\} : \tau}{\Gamma \vdash \{\overline{\ell = e}\} : \tau}$$

Rcd-I
$$\frac{\mathscr{E}[\square \triangleleft \nu \bar{\gamma}.\, \text{rcd } T\ \bar{\gamma} \mid \bar{e}] \qquad \Gamma \vdash \mathscr{E}[T.\{\overline{\ell = e}\}] : \tau}{\Gamma \vdash \mathscr{E}[\{\overline{\ell = e}\}] : \tau}$$

Rcd-Proj-X
$$\frac{T.\ell \leq \tau_1 \to \tau_2 \qquad \Gamma \vdash e : \tau_1}{\Gamma \vdash e.T.\ell : \tau_2}$$

Rcd-Proj-Closed
$$\frac{\ell \triangleright T \qquad \Gamma \vdash e.T.\ell : \tau}{\Gamma \vdash e.\ell : \tau}$$

Rcd-Proj-I
$$\frac{\mathscr{E}[e \triangleright \nu \bar{\gamma}.\, \text{rcd } T\ \bar{\gamma}] \qquad \Gamma \vdash \mathscr{E}[e.T.\ell] : \tau}{\Gamma \vdash \mathscr{E}[e.\ell] : \tau}$$

Hole
$$\frac{(\Gamma \vdash e_i : \tau_i)_{i=1}^{n}}{\Gamma \vdash \square \text{ with } \bar{e} : \tau'}$$

$$\mathscr{E}[e \triangleright \varsigma] \triangleq \forall \Gamma, \tau, \mathfrak{g},\ \ \Gamma \vdash \lfloor \mathscr{E}[\square \text{ with } (e : \mathfrak{g})] \rfloor : \tau \implies \text{shape}(\mathfrak{g}) = \varsigma$$
$$\mathscr{E}[\square \triangleleft \varsigma \mid \bar{e}] \triangleq \forall \Gamma, \tau, \mathfrak{g},\ \ \Gamma \vdash \lfloor \mathscr{E}[((\square \text{ with } \bar{e}) : \mathfrak{g})] \rfloor : \tau \implies \text{shape}(\mathfrak{g}) = \varsigma$$

$\boxed{[\![\Gamma \vdash e : \tau]\!]}$ $[\![\Gamma \vdash e : \tau]\!]$ is satisfiable iff e has the expected *known* type τ under *known* context Γ

$\boxed{[\![e : \sigma]\!]}$ $[\![e : \sigma]\!]$ is satisfiable iff e has the expected *known* type scheme σ.

$\boxed{[\![e : \tau]\!]}$ $[\![e : \tau]\!]$ is satisfiable iff e has the expected *known* type τ.

$$
\begin{array}{lll}
[\![x : \tau]\!] & \triangleq & x\ \tau \\
[\![() : \tau]\!] & \triangleq & \tau = 1 \\
[\![\lambda x.\, e : \tau]\!] & \triangleq & \exists \alpha, \beta.\, \mathsf{let}\ x = \lambda \alpha'.\, \alpha' = \alpha\ \mathsf{in}\ [\![e : \beta]\!] \wedge \tau = \alpha \to \beta \\
[\![e_1\ e_2 : \tau]\!] & \triangleq & \exists \alpha, \beta.\, [\![e_1 : \beta]\!] \wedge [\![e_2 : \alpha]\!] \wedge \beta = \alpha \to \tau \\
[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau]\!] & \triangleq & \mathsf{let}\ x = \lambda \alpha.\, [\![e_1 : \alpha]\!]\ \mathsf{in}\ [\![e_2 : \tau]\!] \\
[\![(e : \exists \bar{\alpha}.\, \tau') : \tau]\!] & \triangleq & \exists \bar{\alpha}.\, [\![e : \tau']\!] \wedge \tau = \tau' \\
[\![(e_1, \ldots, e_n) : \tau]\!] & \triangleq & \exists \bar{\alpha}.\, \tau = \Pi_{i=1}^{n} \bar{\alpha} \wedge \bigwedge_{i=1}^{n} [\![e_i : \alpha_i]\!] \\
[\![e.j^n : \tau]\!] & \triangleq & \exists \alpha, \bar{\alpha}.\, [\![e : \alpha]\!] \wedge \alpha = \Pi_{i=1}^{n} \bar{\alpha} \wedge \tau = \alpha_j \\
[\![e.j : \tau]\!] & \triangleq & \exists \alpha.\, [\![e : \alpha]\!] \wedge \mathsf{match}\ \alpha\ \mathsf{with}\ \Pi\ \beta_j \Rightarrow \tau = \beta \\
[\![[e : \exists \bar{\alpha}.\, \sigma] : \tau]\!] & \triangleq & \exists \bar{\alpha}.\, [\![e : \sigma]\!] \wedge \tau = [\sigma] \\
[\![\langle e : \exists \bar{\alpha}.\, \sigma \rangle : \tau]\!] & \triangleq & \exists \bar{\alpha}, \beta.\, [\![e : \beta]\!] \wedge \beta = [\sigma] \wedge \sigma \leq \tau \\
[\![\langle e \rangle : \tau]\!] & \triangleq & \exists \alpha.\, [\![e : \alpha]\!] \wedge \mathsf{match}\ \alpha\ \mathsf{with}\ [s] \Rightarrow s \leq \tau \\
[\![[e] : \tau]\!] & \triangleq & \mathsf{let}\ x = \lambda \alpha.\, [\![e : \alpha]\!]\ \mathsf{in}\ \mathsf{match}\ \tau\ \mathsf{with}\ [s] \Rightarrow x \leq s \\
[\![e.\ell : \tau]\!] & \triangleq & \begin{cases} [\![e.T.\ell : \tau]\!] & \text{if } \ell \triangleright T \\ \exists \alpha.\, [\![e : \alpha]\!] \wedge \mathsf{match}\ \alpha\ \mathsf{with}\ \mathsf{rcd}\ t\ \_ \Rightarrow t.\ell \leq \alpha \to \tau & \text{otherwise} \end{cases} \\
[\![e.T.\ell : \tau]\!] & \triangleq & \exists \alpha.\, [\![e : \alpha]\!] \wedge T.\ell \leq \alpha \to \tau \\
[\![\{\overline{\ell = e}\} : \tau]\!] & \triangleq & \begin{cases} [\![T.\{\overline{\ell = e}\} : \tau]\!] & \text{if } \bar{\ell} \blacktriangleright T \\ \exists \bar{\alpha}.\ \bigwedge_{i=1}^{n} [\![e_i : \alpha_i]\!] & \text{otherwise} \\ \quad \wedge\ \mathsf{match}\ \tau\ \mathsf{with}\ \mathsf{rcd}\ t\ \_ \Rightarrow (\mathsf{dom}\ t = \bar{\ell} \wedge \bigwedge_{i=1}^{n} t.\ell_i \leq \tau \to \alpha_i) & \end{cases} \\
[\![T.\{\overline{\ell = e}\} : \tau]\!] & \triangleq & \exists \bar{\alpha}.\ \bigwedge_{i=1}^{n} [\![e_i : \alpha_i]\!] \wedge \mathsf{dom}\ T = \bar{\ell} \wedge \bigwedge_{i=1}^{n} T.\ell_i \leq \tau \to \alpha_i \\
[\![\square\ \mathsf{with}\ \bar{e} : \tau]\!] & \triangleq & \exists \bar{\alpha}.\ \bigwedge_{i=1}^{n} [\![e_i : \alpha_i]\!] \\
 & & \\
[\![e : \forall \bar{\alpha}.\, \tau]\!] & \triangleq & \forall \bar{\alpha}.\, [\![e : \tau]\!] \\
 & & \\
[\![\emptyset \vdash e : \tau]\!] & \triangleq & [\![e : \tau]\!] \\
[\![x : \sigma, \Gamma \vdash e : \tau]\!] & \triangleq & \mathsf{let}\ x = \lambda \alpha.\, \sigma \leq \alpha\ \mathsf{in}\ [\![\Gamma \vdash e : \tau]\!]
\end{array}
$$

Note: When τ is α it is considered an unknown *expected type.*

$\boxed{e\ \mathsf{simple}}$ The term e is simple.

$$
\textsc{Simple-Var}\ \frac{}{x\ \mathsf{simple}}
\qquad
\textsc{Simple-Fun}\ \frac{e\ \mathsf{simple}}{\lambda x.\, e\ \mathsf{simple}}
\qquad
\textsc{Simple-App}\ \frac{e_1\ \mathsf{simple} \qquad e_2\ \mathsf{simple}}{e_1\ e_2\ \mathsf{simple}}
\qquad
\textsc{Simple-Unit}\ \frac{}{()\ \mathsf{simple}}
$$

$$
\textsc{Simple-Let}\ \frac{e_1\ \mathsf{simple} \qquad e_2\ \mathsf{simple}}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \mathsf{simple}}
\qquad
\textsc{Simple-Annot}\ \frac{e\ \mathsf{simple}}{(e : \exists \bar{\alpha}.\, \tau)\ \mathsf{simple}}
\qquad
\textsc{Simple-Tuple}\ \frac{(e_i\ \mathsf{simple})_{i=1}^{n}}{(e_1, \ldots, e_n)\ \mathsf{simple}}
\qquad
\textsc{Simple-Proj-X}\ \frac{e\ \mathsf{simple}}{e.j^n\ \mathsf{simple}}
$$

$$\frac{e \text{ simple}}{[e : \exists \bar{\alpha}.\, \sigma] \text{ simple}}\ \textsc{Simple-Poly-X} \qquad \frac{e \text{ simple}}{\langle e : \exists \bar{\alpha}.\, \sigma\rangle \text{ simple}}\ \textsc{Simple-Use-X}$$

$$\frac{(e_i \text{ simple})_{i=1}^n}{T.\{\ell_1 = e_1 \ \dots\ \ell_n = e_n\}}\ \textsc{Simple-Rcd-X} \qquad \frac{(e_i \text{ simple})_{i=1}^n \qquad \bar{\ell} \blacktriangleright T}{\{\ell_1 = e_1 \ \dots\ \ell_n = e_n\}}\ \textsc{Simple-Rcd-Closed}$$

$$\frac{e \text{ simple}}{e.T.\ell \text{ simple}}\ \textsc{Simple-Rcd-Proj-X} \qquad \frac{e \text{ simple} \qquad \ell \triangleright T}{e.\ell \text{ simple}}\ \textsc{Simple-Rcd-Proj-Closed} \qquad \frac{(e_i \text{ simple})_{i=1}^n}{(\square \text{ with } \bar{e}) \text{ simple}}\ \textsc{Simple-Hole}$$

$\boxed{\lfloor e \rfloor}$ The erasure of e.

$$\begin{array}{rcl}
\lfloor x \rfloor & \triangleq & x \\
\lfloor \lambda x.\, e \rfloor & \triangleq & \lambda x.\, \lfloor e \rfloor \\
\lfloor e_1\ e_2 \rfloor & \triangleq & \lfloor e_1 \rfloor\ \lfloor e_2 \rfloor \\
\lfloor () \rfloor & \triangleq & () \\
\lfloor \text{let } x = e_1 \text{ in } e_2 \rfloor & \triangleq & \text{let } x = \lfloor e_1 \rfloor \text{ in } \lfloor e_2 \rfloor \\
\lfloor (e : \exists \bar{\alpha}.\, \tau) \rfloor & \triangleq & (\lfloor e \rfloor : \exists \bar{\alpha}.\, \tau) \\
\lfloor (e_1, \dots, e_n) \rfloor & \triangleq & (\lfloor e_1 \rfloor, \dots, \lfloor e_n \rfloor) \\
\lfloor e.j \rfloor & \triangleq & \square \text{ with } \lfloor e \rfloor \\
\lfloor e.j^n \rfloor & \triangleq & \lfloor e \rfloor.j^n \\
\lfloor [e : \exists \bar{\alpha}.\, \sigma] \rfloor & \triangleq & [\lfloor e \rfloor : \exists \bar{\alpha}.\, \sigma] \\
\lfloor [e] \rfloor & \triangleq & \square \text{ with } \lfloor e \rfloor \\
\lfloor \langle e \rangle \rfloor & \triangleq & \square \text{ with } \lfloor e \rfloor \\
\lfloor \langle e : \exists \bar{\alpha}.\, \sigma \rangle \rfloor & \triangleq & \langle \lfloor e \rfloor : \exists \bar{\alpha}.\, \sigma \rangle \\
\lfloor \{\ell_1 = e_1 \,;\, \dots \,;\, \ell_n = e_n\} \rfloor & \triangleq & \begin{cases} \{\ell_1 = \lfloor e_1 \rfloor \,;\, \dots \,;\, \ell_n = \lfloor e_n \rfloor\} & \text{if } \bar{\ell} \blacktriangleright T \\ \square \text{ with } \lfloor e_1 \rfloor, \dots, \lfloor e_n \rfloor & \text{otherwise} \end{cases} \\
\lfloor T.\{\ell_1 = e_1 \,;\, \dots \,;\, \ell_n = e_n\} \rfloor & \triangleq & T.\{\ell_1 = \lfloor e_1 \rfloor \,;\, \dots \,;\, \ell_n = \lfloor e_n \rfloor\} \\
\lfloor e.\ell \rfloor & \triangleq & \begin{cases} \lfloor e \rfloor.\ell & \text{if } \ell \triangleright T \\ \square \text{ with } \lfloor e \rfloor & \text{otherwise} \end{cases} \\
\lfloor e.T.\ell \rfloor & \triangleq & \lfloor e \rfloor.T.\ell \\
\lfloor \square \text{ with } \bar{e} \rfloor & \triangleq & \square \text{ with } (\lfloor e_i \rfloor)_{i=1}^n
\end{array}$$

$\boxed{\Gamma \vdash^{\text{sd}}_{\text{simple}} e : \tau}$ Under the typing context Γ, the simple term e has the type τ.

$$\frac{x : \forall \bar{\alpha}.\, \tau \in \Gamma}{\Gamma \vdash^{\text{sd}}_{\text{simple}} x : \tau[\bar{\alpha} := \bar{\tau}]}\ \textsc{Var-SD} \qquad \frac{\Gamma \vdash^{\text{sd}}_{\text{simple}} e_1 : \tau_1 \qquad \bar{\alpha} \# \text{fv}(\Gamma) \qquad \Gamma, x : \forall \bar{\alpha}.\, \tau_1 \vdash^{\text{sd}}_{\text{simple}} e_2 : \tau_2}{\Gamma \vdash^{\text{sd}}_{\text{simple}} \text{let } x = e_1 \text{ in } e_2 : \tau_2}\ \textsc{Let-SD}$$

$$\frac{\Gamma \vdash^{\text{sd}}_{\text{simple}} e : \tau[\bar{\alpha} := \bar{\tau}] \qquad \bar{\beta} \# \Gamma}{\Gamma \vdash^{\text{sd}}_{\text{simple}} [e : \exists \bar{\alpha}.\, \forall \bar{\beta}.\, \tau] : [\sigma[\bar{\alpha} := \bar{\tau}]]}\ \textsc{Poly-X} \qquad \frac{\Gamma \vdash e : [\forall \bar{\beta}.\, \tau][\bar{\alpha} := \bar{\tau}]}{\Gamma \vdash \langle e : \exists \bar{\alpha}.\, \forall \bar{\beta}.\, \tau \rangle : \tau[\bar{\alpha} := \bar{\tau}, \bar{\beta} := \bar{\tau}']}\ \textsc{Use-X}$$

$\boxed{\Vdash e : \tau}$ The term e canonically has the type τ.

$$\textsc{Can-Base}\quad \frac{\emptyset \vdash^{sd}_{simple} e : \tau}{\Vdash e : \tau} \qquad \textsc{Can-Proj-I}\quad \frac{\mathscr{E}[e \triangleright \nu\bar{\gamma}.\, \Pi^n_{i=1}\bar{\gamma}] \qquad \Vdash \mathscr{E}[e.j^n] : \tau}{\Vdash \mathscr{E}[e.j] : \tau}$$

$$\textsc{Can-Poly-I}\quad \frac{\mathscr{E}[\square \triangleleft \nu\bar{\gamma}.\, [\sigma] \mid e] \qquad \Vdash \mathscr{E}[[e : \exists\bar{\gamma}.\, \sigma]] : \tau}{\Vdash \mathscr{E}[[e]] : \tau} \qquad \textsc{Can-Use-I}\quad \frac{\mathscr{E}[e \triangleright \nu\bar{\gamma}.\, [\sigma]] \qquad \Vdash \mathscr{E}[\langle e : \exists\bar{\gamma}.\, \sigma\rangle] : \tau}{\Vdash \mathscr{E}[\langle e\rangle] : \tau}$$

$$\textsc{Can-Rcd-I}\quad \frac{\mathscr{E}[\square \triangleleft \nu\bar{\gamma}.\, \mathsf{rcd}\ T\ \bar{\gamma} \mid \bar{e}] \qquad \Vdash \mathscr{E}[T.\{\ell_1 = e_1\ ;\ \ldots\ ;\ \ell_n = e_n\}] : \tau}{\Vdash \mathscr{E}[\{\ell_1 = e_1\ ;\ \ldots\ ;\ \ell_n = e_n\}] : \tau}$$

$$\textsc{Can-Rcd-Proj-I}\quad \frac{\mathscr{E}[\square \triangleleft \nu\bar{\gamma}.\, \mathsf{rcd}\ T\ \bar{\gamma} \mid e] \qquad \Vdash \mathscr{E}[e.T.\ell] : \tau}{\Vdash \mathscr{E}[e.\ell] : \tau}$$

$\boxed{U \longrightarrow U'}$ The unifier rewrites U to U'.

$$\textsc{U-Exists}\quad \frac{(\exists\alpha.\, U_1) \wedge U_2 \qquad \alpha \# U_2}{\exists\alpha.\, U_1 \wedge U_2} \qquad \textsc{U-Cycle}\quad \frac{U \qquad \mathsf{cyclic}(U)}{\mathsf{false}} \qquad \textsc{U-True}\quad \frac{U \wedge \mathsf{true}}{U} \qquad \textsc{U-False}\quad \frac{\mathscr{U}[\mathsf{false}] \qquad \mathscr{U} \neq \square}{\mathsf{false}}$$

$$\textsc{U-Merge}\quad \frac{\alpha = \epsilon_1 \wedge \alpha = \epsilon_2}{\alpha = \epsilon_1 = \epsilon_2} \qquad \textsc{U-Stutter}\quad \frac{\alpha = \alpha = \epsilon}{\alpha = \epsilon} \qquad \textsc{U-Name}\quad \frac{\varsigma\ (\bar{\tau}, \tau_i, \bar{\tau}') = \epsilon \qquad \alpha \# \bar{\tau}, \bar{\tau}', \epsilon \qquad \tau_i \notin \mathcal{V}}{\exists\alpha.\, \alpha = \tau_i \wedge \varsigma\ (\bar{\tau}, \alpha, \bar{\tau}') = \epsilon} \qquad \textsc{U-Decomp}\quad \frac{\varsigma\ \bar{\alpha} = \varsigma\ \bar{\beta} = \epsilon}{\varsigma\ \bar{\alpha} = \epsilon \wedge \bar{\alpha} = \bar{\beta}}$$

$$\textsc{U-Clash}\quad \frac{\varsigma\ \bar{\alpha} = \varsigma'\ \bar{\beta} = \epsilon \qquad \varsigma \neq \varsigma'}{\mathsf{false}} \qquad \textsc{U-Trivial}\quad \frac{\epsilon \qquad |\epsilon| \leq 1}{\mathsf{true}}$$

$\boxed{\alpha \prec_U \beta}$ α occurs in β in the constraint U.

$\boxed{\mathsf{cyclic}(U)}$ U contains a cyclic type.

$$\textsc{Occurs}\quad \frac{\alpha \in \mathsf{fv}(\tau) \qquad \tau \notin \mathcal{V} \qquad \alpha, \beta \# \mathsf{bv}(\mathscr{U})}{\alpha \prec_{\mathscr{U}[\beta = \tau = \epsilon]} \beta} \qquad \textsc{Cycle}\quad \frac{\alpha \prec^+_U \alpha}{\mathsf{cyclic}(U)}$$

Definition A.2 (Solved form $\hat{U}$). We write $\hat{U}$ for constraints in *solved form*, that is, constraints of the form $\exists\bar{\alpha}.\, \bar{\epsilon}$, where: (1) each ϵ_i contains at most one non-variable type; (2) each type variable may occur as a member of at most one multi-equation ϵ_i; (3) the constraint is acyclic.

$\boxed{C \longrightarrow C'}$ The constraint solver rewrites C to C'.

S-Unif
$$\frac{U_1 \qquad U_1 \longrightarrow U_2}{U_2}$$

S-True
$$\frac{C \wedge \text{true}}{C}$$

S-False
$$\frac{\mathscr{C}[\text{false}] \qquad \mathscr{C} \neq \square}{\text{false}}$$

S-Let
$$\frac{\text{let } x = \lambda\alpha. C_1 \text{ in } C_2}{\text{let } x\ \alpha\ [\emptyset] = C_1 \text{ in } C_2}$$

S-Exists-Conj
$$\frac{(\exists\alpha. C_1) \wedge C_2 \qquad \alpha \# C_2}{\exists\alpha. C_1 \wedge C_2}$$

S-Let-ExistsLeft
$$\frac{\text{let } x\ \alpha\ [\bar{\alpha}] = \exists\beta. C_1 \text{ in } C_2 \qquad \beta \# \alpha, \bar{\alpha}, C_2}{\text{let } x\ \alpha\ [\bar{\alpha}, \beta] = C_1 \text{ in } C_2}$$

S-Let-ExistsRight
$$\frac{\text{let } x\ \alpha\ [\bar{\alpha}] = C_1 \text{ in } \exists\beta. C_2 \qquad \beta \# \alpha, \bar{\alpha}, C_1}{\exists\beta. \text{ let } x = \lambda\bar{\alpha}. C_1 \text{ in } C_2}$$

S-Let-ConjLeft
$$\frac{\text{let } x\ \alpha\ [\bar{\alpha}] = C_1 \wedge C_2 \text{ in } C_3 \qquad C_1 \# \alpha, \bar{\alpha}}{C_1 \wedge \text{let } x\ \alpha\ [\bar{\alpha}] = C_2 \text{ in } C_3}$$

S-Let-ConjRight
$$\frac{\text{let } x\ \alpha\ [\bar{\alpha}] = C_1 \text{ in } (C_2 \wedge C_3) \qquad x \# C_3}{C_3 \wedge \text{let } x\ \alpha = C_1 \text{ in } C_2}$$

S-Match-Ctx
$$\frac{\mathscr{C}[\text{match } \tau \text{ with } \bar{\chi}] \qquad \vdash \mathscr{C}[\tau\,!\,\varsigma]}{\mathscr{C}[\text{match } \tau := \varsigma \text{ with } \bar{\chi}]}$$

S-Inst-Name
$$\frac{i(\alpha) \rightsquigarrow \tau \qquad \tau \notin \mathcal{V}}{\exists\gamma. \gamma = \tau \wedge i(\alpha) \rightsquigarrow \gamma}$$

S-Let-AppR
$$\frac{\text{let } x\ \alpha\ [\bar{\alpha}] = C \text{ in } \mathscr{C}[x\ \tau] \qquad \gamma \# \tau \qquad x \# \text{bv}(\mathscr{C})}{\text{let } x\ \alpha\ [\bar{\alpha}] = C \text{ in } \mathscr{C}[\exists\gamma, i^x. i(\alpha) \rightsquigarrow \gamma \wedge \gamma = \tau]}$$

S-Inst-Copy
$$\frac{\begin{array}{c}\text{let } x\ \alpha\ [\bar{\alpha}] = C \text{ in } \mathscr{C}[i^x(\alpha') \rightsquigarrow \gamma] \qquad C = C' \wedge \alpha' = \varsigma\,\bar{\beta} = \epsilon \\ \alpha' \in \alpha, \bar{\alpha} \qquad \neg\text{cyclic}(C) \qquad \bar{\beta}' \# \alpha', \gamma, \bar{\beta} \qquad x \# \text{bv}(\mathscr{C})\end{array}}{\text{let } x\ \alpha\ [\bar{\alpha}] = C \text{ in } \mathscr{C}[\exists\bar{\beta}'. \gamma = \varsigma\,\bar{\beta}' \wedge \overline{i^x(\beta) \rightsquigarrow \beta'}]}$$

S-Inst-Unify
$$\frac{i(\alpha) \rightsquigarrow \gamma_1 \wedge i(\alpha) \rightsquigarrow \gamma_2}{i(\alpha) \rightsquigarrow \gamma_1 \wedge \gamma_1 = \gamma_2}$$

S-Inst-Poly
$$\frac{\begin{array}{c}\text{let } x\ \alpha\ [\bar{\alpha}] = \bar{\epsilon} \wedge C \text{ in } \mathscr{C}[i^x(\alpha') \rightsquigarrow \gamma] \qquad \forall\beta. \exists\bar{\beta}. \bar{\epsilon} \equiv \text{true} \\ \beta, \bar{\beta} \subseteq \alpha, \bar{\alpha} \qquad \bar{\beta} \# C \qquad i(\beta) \# \text{insts}(\mathscr{C}) \qquad x \# \text{bv}(\mathscr{C})\end{array}}{\text{let } x\ \alpha\ [\bar{\alpha}] = \bar{\epsilon} \wedge C \text{ in } \mathscr{C}[\text{true}]}$$

S-Inst-Mono
$$\frac{\begin{array}{c}\text{let } x\ \alpha\ [\bar{\alpha}] = C \text{ in } \mathscr{C}[i^x(\beta) \rightsquigarrow \gamma] \\ \beta \notin \alpha, \bar{\alpha} \qquad x, \beta \# \text{bv}(\mathscr{C})\end{array}}{\text{let } x\ \alpha\ [\bar{\alpha}] = C \text{ in } \mathscr{C}[\beta = \gamma]}$$

S-Compress
$$\frac{\text{let } x\ \alpha\ [\bar{\alpha}, \beta] = C_1 \wedge \beta = \gamma = \epsilon \text{ in } C_2 \qquad \beta \neq \gamma}{\text{let } x\ \alpha\ [\bar{\alpha}] = C_1[\beta := \gamma] \wedge \gamma = \epsilon[\beta := \gamma] \text{ in } C_2[x(\beta) := \gamma]}$$

S-Gc
$$\frac{\text{let } x\ \alpha\ [\bar{\alpha}, \beta] = C_1 \wedge \beta = \epsilon \text{ in } C_2 \qquad \beta \# C_1, \epsilon, C_2}{\text{let } x\ \alpha\ [\bar{\alpha}] = C_1 \wedge \epsilon \text{ in } C_2}$$

S-Exists-Lower
$$\frac{\text{let } x\ \alpha\ [\bar{\alpha}, \bar{\beta}] = C_1 \text{ in } C_2 \qquad \vdash \exists\alpha, \bar{\alpha}. C_1 \text{ determines } \bar{\beta}}{\exists\bar{\beta}. \text{ let } x\ \alpha\ [\bar{\alpha}] = C_1 \text{ in } C_2}$$

S-Exists-Exists-Inst
$$\frac{\exists i^x. \exists\alpha. C}{\exists\alpha. \exists i^x. C}$$

$$\textsc{S-Exists-Inst-Conj}\quad \frac{\exists i^x.\, C_1 \wedge C_2 \qquad i \# C_1}{C_1 \wedge \exists i^x.\, C_2}$$

$$\textsc{S-Exists-Inst-Let}\quad \frac{\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = C_1\ \mathsf{in}\ \exists i^{x'}.\, C_2 \qquad x \neq x'}{\exists i^{x'}.\, \mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = C_1\ \mathsf{in}\ C_2}$$

$$\textsc{S-Exists-Inst-Solve}\quad \frac{\exists i^x.\, C \qquad i \# C}{C}$$

$$\textsc{S-All-Conj}\quad \frac{\forall \bar{\alpha}.\, \exists \bar{\beta}.\, C_1 \wedge C_2 \qquad \bar{\alpha}, \bar{\beta} \# C_1}{C_1 \wedge \forall \bar{\alpha}.\, \exists \bar{\beta}.\, C_2}$$

$$\textsc{S-Exists-All}\quad \frac{\forall \bar{\alpha}.\, \exists \bar{\beta}, \bar{\gamma}.\, C \qquad \vdash \exists \bar{\alpha}, \bar{\beta}.\, C\ \mathsf{determines}\ \bar{\gamma}}{\exists \bar{\gamma}.\, \forall \bar{\alpha}.\, \exists \bar{\beta}.\, C}$$

$$\textsc{S-All-Escape}\quad \frac{\forall \bar{\alpha}, \alpha.\, \exists \bar{\beta}.\, C \wedge \bar{\epsilon} \qquad \alpha \prec^*_{\bar{\epsilon}} \gamma \qquad \gamma \# \alpha, \bar{\beta} \qquad \alpha \# \bar{\beta}}{\mathsf{false}}$$

$$\textsc{S-All-Rigid}\quad \frac{\forall \bar{\alpha}, \alpha.\, \exists \bar{\beta}.\, C \wedge \alpha = \tau = \epsilon \qquad \tau \notin \mathcal{V} \qquad \alpha \# \bar{\beta}}{\mathsf{false}}$$

$$\textsc{S-All-Solve}\quad \frac{\forall \bar{\alpha}.\, \exists \bar{\beta}.\, \bar{\epsilon} \qquad \exists \bar{\beta}.\, \bar{\epsilon} \equiv \mathsf{true}}{\mathsf{true}}$$

$\boxed{\vdash \mathscr{C}[\tau \,!\, \varsigma]}$ Under $\mathscr{C}$, the type τ has the provably unique canonical shape ς.

$$\textsc{S-Uni-Var}\quad \frac{\alpha \# \mathsf{bv}(\mathscr{C}_2)}{\vdash \mathscr{C}_1[\alpha = \tau = \epsilon \wedge \mathscr{C}_2[-]][\alpha \,!\, \mathsf{shape}(\tau)]}$$

$$\textsc{S-Uni-Type}\quad \frac{\tau \notin \mathcal{V}}{\vdash \mathscr{C}[\tau \,!\, \mathsf{shape}(\tau)]}$$

$$\textsc{S-Uni-BackProp}\quad \frac{\vdash \mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = \mathscr{C}_1[\mathsf{true}]\ \mathsf{in}\ \mathscr{C}_2[i^x(\alpha') \rightsquigarrow \gamma \wedge -][\gamma \,!\, \varsigma] \qquad \alpha' \in \alpha, \bar{\alpha} \qquad x \# \mathsf{bv}(\mathscr{C}_2) \qquad \alpha' \# \mathsf{bv}(\mathscr{C}_1)}{\vdash \mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = \mathscr{C}_1[-]\ \mathsf{in}\ \mathscr{C}_2[i^x(\alpha') \rightsquigarrow \gamma][\alpha \,!\, \varsigma]}$$

Definition A.3. C determines $\bar{\beta}$ if and only if every ground assignments ϕ and ϕ' that satisfy (the erasure of) C and coincide outside of β coincide on $\bar{\beta}$ as well.

$$C\ \mathsf{determines}\ \beta \ \triangleq\ \forall \phi, \phi'.\ \ \phi \vdash \lfloor C \rfloor \wedge \phi' \vdash \lfloor C \rfloor \wedge \phi =_{\backslash \bar{\beta}} \phi' \implies \phi = \phi'$$

$\boxed{\vdash C\ \mathsf{determines}\ \bar{\alpha}}$ C provably determines $\bar{\alpha}$.

$$\textsc{S-Det-Dom}\quad \frac{\gamma \# \bar{\beta}, \bar{\alpha} \qquad \bar{\alpha} \subseteq \mathsf{fv}(\epsilon)}{\vdash \exists \bar{\beta}.\, C \wedge \gamma = \epsilon\ \mathsf{determines}\ \bar{\alpha}}$$

$$\textsc{S-Det-Esc}\quad \frac{\mathsf{fv}(\tau) \# \bar{\alpha}, \bar{\beta}}{\vdash \exists \bar{\beta}.\, C \wedge \bar{\alpha} = \tau = \epsilon\ \mathsf{determines}\ \bar{\alpha}}$$

$\boxed{\mathsf{insts}(C)}$ The set of instantiations in C.

$$
\begin{array}{rcl}
\mathsf{insts}(\mathsf{true}) & \triangleq & \emptyset \\
\mathsf{insts}(\mathsf{false}) & \triangleq & \emptyset \\
\mathsf{insts}(C_1 \wedge C_2) & \triangleq & \mathsf{insts}(C_1) \cup \mathsf{insts}(C_2) \\
\mathsf{insts}(\exists \alpha. C) & \triangleq & \mathsf{insts}(C) \\
\mathsf{insts}(\forall \alpha. C) & \triangleq & \mathsf{insts}(C) \\
\mathsf{insts}(\tau = \tau') & \triangleq & \emptyset \\
\mathsf{insts}(\mathsf{let}\ x = \lambda\alpha. C_1\ \mathsf{in}\ C_2) & \triangleq & \mathsf{insts}(C_1) \cup \mathsf{insts}(C_2) \\
\mathsf{insts}(x\ \tau) & \triangleq & \emptyset \\
\mathsf{insts}(\epsilon) & \triangleq & \emptyset \\
\mathsf{insts}(\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = C_1\ \mathsf{in}\ C_2) & \triangleq & \mathsf{insts}(C_1) \cup \mathsf{insts}(C_2) \\
\mathsf{insts}(\exists i^x. C) & \triangleq & \mathsf{insts}(C) \\
\mathsf{insts}(i(\alpha) \rightsquigarrow \tau) & \triangleq & \{i(\alpha)\}
\end{array}
$$

Definition A.4 (Normal forms). A constraint C is in *normal form*, written $\hat{C}$, if no rewriting rules apply, *i.e.*, $C \not\longrightarrow$. Similarly, a constraint context $\mathscr{C}$ is in *normal form*, written $\hat{\mathscr{C}}$, if $\mathscr{C}[\mathsf{true}]$ is in normal form.

Definition A.5 (Measure). For the relation $\phi \vdash C$, the following measure enables a useful induction principle:

$$\|C\| \;\triangleq\; \langle \#\mathsf{match}\ C, |C| \rangle$$

where $\langle \ldots \rangle$ denotes a pair with lexicographic ordering, and:

(1) $\#\mathsf{match}\ C$ is the number of match τ with $\bar{\chi}$ constraints in C.
(2) the last component $|C|$ is a structural measure of constraints *i.e.*, a conjunction $C_1 \wedge C_2$ is larger than the two conjuncts C_1, C_2.

$\boxed{[\![\mathscr{E}[\square : \tau'] : \tau]\!]}$ $[\![\mathscr{E}[\square : \tau'] : \tau]\!]$ is a satisfiable context iff the context $\mathscr{E}$ has the expected type τ given the hole has the type τ'.

$$
\begin{array}{ll}
[\![\square[\square : \tau] : \tau]\!] & \triangleq \square \\
[\![(\mathscr{E}\ e)[\square : \tau'] : \tau]\!] & \triangleq \exists \alpha, \beta. \alpha = \beta \to \tau \wedge [\![\mathscr{E}[\square : \tau'] : \alpha]\!] \wedge [\![e : \beta]\!] \\
[\![(e\ \mathscr{E})[\square : \tau'] : \tau]\!] & \triangleq \exists \alpha, \beta. \alpha = \beta \to \tau \wedge [\![e : \alpha]\!] \wedge [\![\mathscr{E}[\square : \tau'] : \beta]\!] \\
[\![(\mathsf{let}\ x = \mathscr{E}\ \mathsf{in}\ e)[\square : \tau'] : \tau]\!] & \triangleq \mathsf{let}\ x = \lambda\alpha. [\![\mathscr{E}[\square : \tau'] : \alpha]\!]\ \mathsf{in}\ [\![e : \tau]\!] \\
[\![(\mathsf{let}\ x = e\ \mathsf{in}\ \mathscr{E})[\square : \tau'] : \tau]\!] & \triangleq \mathsf{let}\ x = \lambda\alpha. [\![e : \alpha]\!]\ \mathsf{in}\ [\![\mathscr{E}[\square : \tau'] : \tau]\!] \\
[\![(\mathscr{E} : \exists \bar{\alpha}. \tau'')[\square : \tau'] : \tau]\!] & \triangleq \exists \bar{\alpha}. \tau = \tau'' \wedge [\![\mathscr{E}[\square : \tau'] : \tau]\!] \\
[\![(e_1, \ldots, \mathscr{E}_j, \ldots, e_n)[\square : \tau'] : \tau]\!] & \triangleq \exists \bar{\alpha}. \tau = \Pi_{i=1}^{n} \alpha_i \wedge \bigwedge_{i \neq j} [\![e_i : \alpha_i]\!] \wedge [\![\mathscr{E}_j[\square : \tau'] : \alpha_j]\!] \\
[\![(\mathscr{E}.j^n)[\square : \tau'] : \tau]\!] & \triangleq \exists \alpha, \bar{\alpha}. [\![\mathscr{E}[\square : \tau'] : \alpha]\!] \wedge \alpha = \Pi_{i=1}^{n} \alpha_i \wedge \tau = \alpha_j \\
[\![(\mathscr{E}.j)[\square : \tau'] : \tau]\!] & \triangleq \exists \alpha. [\![\mathscr{E}[\square : \tau'] : \alpha]\!] \wedge \mathsf{match}\ \alpha\ \mathsf{with}\ \Pi\ \gamma_j \Rightarrow \tau = \gamma \\
[\![[\mathscr{E} : \exists \bar{\alpha}. \sigma][\square : \tau'] : \tau]\!] & \triangleq \exists \bar{\alpha}. [\![\mathscr{E}[\square : \tau'] : \sigma]\!] \wedge \tau = [\sigma] \\
[\![\langle \mathscr{E} : \exists \bar{\alpha}. \sigma \rangle[\square : \tau'] : \tau]\!] & \triangleq \exists \bar{\alpha}, \beta. [\![\mathscr{E}[\square : \tau'] : \beta]\!] \wedge \beta = [\sigma] \wedge \sigma \leq \tau \\
[\![[\mathscr{E}][\square : \tau'] : \tau]\!] & \triangleq \mathsf{let}\ x = \lambda\alpha. [\![\mathscr{E}[\square : \tau'] : \alpha]\!]\ \mathsf{in} \\
 & \quad\ \mathsf{match}\ \tau\ \mathsf{with}\ [s] \Rightarrow x \leq s \\
[\![\langle \mathscr{E} \rangle[\square : \tau'] : \tau]\!] & \triangleq \exists \alpha. [\![\mathscr{E}[\square : \tau'] : \alpha]\!] \wedge \mathsf{match}\ \alpha\ \mathsf{with}\ [s] \Rightarrow s \leq \tau
\end{array}
$$

$$\llbracket (\mathscr{E}.\ell)[\square : \tau'] : \tau \rrbracket \triangleq \begin{cases} \llbracket (\mathscr{E}.T.\ell)[\square : \tau'] : \tau \rrbracket & \text{if } \ell \triangleright T \\ \exists \alpha.\, \llbracket \mathscr{E}[\square : \tau'] : \alpha \rrbracket & \text{otherwise} \\ \wedge\ \text{match } \alpha \text{ with rcd } t\ \_ \Rightarrow t.\ell \leq \alpha \rightarrow \tau & \end{cases}$$

$$\llbracket (\mathscr{E}.T.\ell)[\square : \tau'] : \tau \rrbracket \triangleq \exists \alpha.\, \llbracket \mathscr{E}[\square : \tau'] : \alpha \rrbracket \wedge T.\ell \leq \alpha \rightarrow \tau$$

$$\llbracket \{\ell_1 = e_1 \ ; \dots ;\ \ell_j = \mathscr{E}_j \ ; \dots ;\ \ell_n = e_n\}[\square : \tau'] : \tau \rrbracket \triangleq \begin{cases} \llbracket T.\{\dots ;\ \ell_j = \mathscr{E}_j \ ; \dots\}[\square : \tau'] : \tau \rrbracket & \text{if } \bar{\ell} \blacktriangleright T \\ \exists \bar{\alpha}.\, \bigwedge_{i \neq j} \llbracket e_i : \alpha_i \rrbracket \wedge \llbracket \mathscr{E}_j[\square : \tau'] : \alpha_j \rrbracket & \text{otherwise} \\ \quad \wedge\ \text{match } \tau \text{ with rcd } t\ \_ \Rightarrow & \\ \qquad \text{dom } t = \bar{\ell} \wedge \bigwedge_{i=1}^{n} t.\ell_i \leq \tau \rightarrow \alpha_i & \end{cases}$$

$$\llbracket T.\{\ell_1 = e_1 \ ; \dots ;\ \ell_j = \mathscr{E}_j \ ; \dots ;\ \ell_n = e_n\}[\square : \tau'] : \tau \rrbracket \triangleq \exists \bar{\alpha}.\, \bigwedge_{i \neq j} \llbracket e_i : \alpha_i \rrbracket \wedge \llbracket \mathscr{E}_j[\square : \tau'] : \alpha_j \rrbracket \wedge \text{dom } T = \bar{\ell} \wedge \bigwedge_{i=1}^{n} T.\ell_i \leq \tau \rightarrow \alpha_i$$

$$\llbracket (\square \text{ with } e_1, \dots, \mathscr{E}_j, \dots, e_n)[\square : \tau'] : \tau \rrbracket \triangleq \exists \bar{\alpha}.\, \bigwedge_{i \neq j} \llbracket e_i : \alpha_i \rrbracket \wedge \llbracket \mathscr{E}_j[\square : \tau'] : \alpha_j \rrbracket$$

$$\llbracket \mathscr{E}[\square : \tau'] : \forall \bar{\alpha}.\, \tau \rrbracket \triangleq \forall \bar{\alpha}.\, \llbracket \mathscr{E}[\square : \tau'] : \tau \rrbracket$$

B Properties of the constraint language

This appendix establishes key properties of the constraint language. The first is the principality of shapes Theorem 3.2: any non-variable type τ admits a non-trivial principal shape ς.

The second is the canonicalization of satisfiability derivations $\phi \vdash C$, which enables a simple induction principal for reasoning about unicity. This canonical form for derivations is a crucial tool in our proof of soundness and completeness in §D.

B.1 Principality of shapes

Theorem 3.2 (Principal shapes). *Any non-variable type τ has a principal shape ζ.*

Proof. Let us assume τ is a non-variable type.

Case *τ is a type constructor* $\mathsf{c}\ \bar{\tau}$.

c is a top-level type constructor of arity n, which in our setting may be the nullary 1, the binary arrow, the n-ary product, or a nominal record type. In all these cases, the shape of τ is $\nu\bar{\gamma}.\, \mathsf{c}\ \bar{\gamma}$ where $\bar{\gamma}$ is a sequence of n distinct type variables. This is clearly principal.

Case *τ is a polytype* $[\forall \bar{\alpha}.\, \tau]$.

We may assume *w.l.o.g.* that each variable of $\bar{\alpha}$ occurs free in τ. Let $(\pi_i)_{i=1}^{n}$ be the sequence of shortest paths in τ that cannot be extended to reach a (polymorphic) variable in $\bar{\alpha}$, in lexicographic order and $\bar{\gamma}$ be a sequence $(\gamma_i)_{i=1}^{n}$ of distinct variables that do not appear in τ. Let τ_0 be $\tau[\pi_i := \gamma_i]_{i=1}^{n}$, *i.e.*, the term τ where each path π_i has been substituted by the variable γ_i. Let ζ be the shape $\nu\bar{\gamma}.\, [\forall \bar{\alpha}.\, \tau_0]$. We claim that ζ is actually the principal shape of $[\forall \bar{\alpha}.\, \tau]$.

By construction, τ is equal to $\zeta\ \bar{\tau}$ (**1**). where $\bar{\tau}$ is the sequence composed of τ_i equal to τ/π_i for i ranging from 1 to n. Indeed, by definition, $\zeta\ \bar{\tau}$ is equal to $(\tau[\pi_i := \gamma_i]_{i=1}^{n})[\gamma_i := \tau_i]$ which is obviously equal to τ. The remaining of the proof checks that ζ is minimal (**2**), that is, we assume that ζ' is another shape such that $[\forall \bar{\alpha}.\, \tau]$ is equal to $\zeta'\ \bar{\tau}'$ for some $\bar{\tau}'$ (**3**) and show that $\zeta \preceq \zeta'$ (**4**).

It follows from (3) that ζ' must be a polytype shape, *i.e.*, of the form $\nu\bar{\gamma}'.\, [\forall \bar{\beta}.\, \tau']$ and $[\forall \bar{\alpha}.\, \tau]$ is equal to $[\forall \bar{\beta}.\, \tau'][\bar{\gamma}' := \bar{\tau}']$ (**5**). We may assume *w.l.o.g.* that $\bar{\beta}$ and $\bar{\gamma}'$ are disjoint, that $\bar{\gamma}'$ does not contain useless variables, *i.e.*, that they all appear in τ' and that they actually appear in lexicographic order. Now that never term contains useless variables, (5) implies that the sequences

$\bar{\alpha}$ and $\bar{\beta}$ can be put in one-to-one correspondences. Besides, since they all ordered in the order of appearance in terms, they the correspondence respects the ordering. Hence, the substitution $[\bar{\beta} := \bar{\alpha}]$ is a renaming. Therefore, we can assume *w.l.o.g.* that $\bar{\beta}$ is $\bar{\alpha}$, That is, (5) becomes that $[\forall\bar{\alpha}.\,\tau]$ is equal to $[\forall\bar{\alpha}.\,\tau'[\bar{\gamma}' := \bar{\tau}']]$, which given that variables $\bar{\alpha}$ appear in the same order in both terms, implies that τ is equal to $\tau'[\bar{\gamma}' := \bar{\tau}']$ (**6**).

Since $\bar{\tau}'$ does not contain any variable in $\bar{\alpha}$, every path π_i is a path in τ'. Thus, we may write τ' as $\tau'[\pi_i := \tau_i'']_{i=1}^n$ where τ_i'' is τ'/π_i. This is also equal to $(\tau'[\pi_i := \gamma_i]_{i=1}^n)[\gamma_i := \tau_i'']_{i=1}^n$, that is $\tau_0[\gamma_i := \tau_i'']_{i=1}^n$. In summary, we have τ' is equal to $\tau_0[\gamma_i := \tau_i'']_{i=1}^n$, which implies that $[\forall\bar{\alpha}.\,\tau']$ is equal to $[\forall\bar{\alpha}.\,\tau_0[\gamma_i := \tau_i'']_{i=1}^n]$, *i.e.*, $[\forall\bar{\alpha}.\,\tau_0][\gamma_i := \tau_i'']_{i=1}^n$ (**7**). By Inst-Shape, we have $\nu\bar{\gamma}.\,[\forall\bar{\alpha}.\,\tau_0] \preceq \nu\bar{\gamma}'.\,[\forall\bar{\alpha}.\,\tau_0][\gamma_i := \tau_i'']_{i=1}^n$, which, given (7), is exactly (4).

□

B.2 Canonicalization of satisfiability

They key result in this section is that our semantic derivations $\phi \vdash C$ can always be rewritten to only apply the rule Match-Ctx at the very bottom of the derivation, rather than in the middle of derivations. This corresponds to explicitating the unique shapes of all suspended constraints (in some order that respects the dependency between suspended constraints), and then continuing with a syntax-directed proof of a fully-discharged constraint.

We did not impose this ordering in our definition of the semantics to make it more flexible and more declarative, but the inversion principle that it provides will be helpful when reasoning about the solver in §C.

We define in §A a formal judgment C simple that says that C does not contain any suspended match constraint, and extend it trivially to constraint contexts: $\mathscr{C}$ simple. In particular, the erasure $\lfloor C \rfloor$ of a constraint (Definition 4.2) is always simple. We then introduce in §A a "canonical" semantic judgment $\phi \Vdash C$ that enforces the structure we mentioned: its derivation starts by discharging suspended constraints, until eventually we reach a simple constraint C. Below we prove that any semantic derivation $\phi \vdash C$ can be turned into a canonical semantic derivation $\phi \Vdash C$.

We can think of this result as controlling the amount of non-syntax-directness in our rules: we need some of it, but it suffices to have it only at the outside, and it contains a more standard derivation that is easy to reason about.

Inversion. When C is simple, a derivation of $\phi \vdash C$ does not use the contextual rule (it is a derivation in $\phi \vdash_{\mathsf{simple}} C$), so it enjoys the usual inversion principle on syntax-directed judgments; for example, if $\phi \vdash_{\mathsf{simple}} C_1 \wedge C_2$ then by inversion $\phi \vdash_{\mathsf{simple}} C_1$ and $\phi \vdash_{\mathsf{simple}} C_2$, etc.

Congruence. Congruence does not hold in general in our system due to the contextual rule. For example, $C_1 \triangleq (\mathsf{match}\ \alpha\ \mathsf{with}\ \_ \Rightarrow \mathsf{true})$ is unsatisfiable so we have $C_1 \equiv \mathsf{false}$, but for $\mathscr{C} \triangleq (\exists\alpha.\,\alpha = \mathsf{int} \wedge \square)$ we have $\mathscr{C}[C_1] \equiv \mathsf{true}$ and $\mathscr{C}[\mathsf{false}] \equiv \mathsf{false}$. It holds simply for simple constraints.

Lemma B.1 (Simple congruence). *Given simple constraints C_1, C_2 and simple context $\mathscr{C}$. If $C_1 \vDash C_2$, then $\mathscr{C}[C_1] \vDash \mathscr{C}[C_2]$.*

Proof. Induction on the derivation of $\mathscr{C}$ simple. □

Composability. The composability result below is an important test of our definition of the unicity condition $\mathscr{C}[\tau\,!\,\varsigma]$, which is in part engineered for this lemma to be simple to prove. In the past we used a definition of unicity that also required $\mathscr{C}[\mathsf{true}]$ to be satisfiable, which broke the composability property.

Lemma B.2 (Composability of unicity). *If* $\mathscr{C}_1[\tau \,!\, \varsigma]$*, then* $\mathscr{C}_2[\mathscr{C}_1][\tau \,!\, \varsigma]$.

Proof. Induction on the structure of $\mathscr{C}_2$.

Case $\square$. immediate.

Case $\mathscr{C}_3 \wedge C$.

$\mathscr{C}_1[\tau \,!\, \varsigma]$	Premise
$\mathscr{C}_3[\mathscr{C}_1][\tau \,!\, \varsigma]$	By *i.h.*
For all $\phi, \mathfrak{g}$	Definition of $(\mathscr{C}_3[\mathscr{C}_1] \wedge C)[\tau \,!\, \varsigma]$
$\phi \vdash \lfloor \mathscr{C}_3[\mathscr{C}_1][\tau = \mathfrak{g}] \rfloor \wedge \lfloor C \rfloor$	$\Longrightarrow$ I
$\phi \vdash \lfloor \mathscr{C}_3[\mathscr{C}_1][\tau = \mathfrak{g}] \rfloor$	Simple inversion
$\text{shape}(\mathfrak{g}) = \varsigma$	$\Longrightarrow$ E on $\mathscr{C}_3[\mathscr{C}_1][\tau \,!\, \varsigma]$
☞ $(\mathscr{C}_3[\mathscr{C}_1] \wedge C)[\tau \,!\, \varsigma]$	Above

Case $C \wedge \mathscr{C}_3$.

Similar to the $\mathscr{C}_3 \wedge C$ case.

Case $\exists \alpha. \mathscr{C}_3$.

$\mathscr{C}_1[\tau \,!\, \varsigma]$	Premise
$\mathscr{C}_3[\mathscr{C}_1][\tau \,!\, \varsigma]$	By *i.h.*
For all $\phi, \mathfrak{g}$	Definition of $(\exists \alpha. \mathscr{C}_3[\mathscr{C}_1])[\tau \,!\, \varsigma]$
$\phi \vdash \exists \alpha. \lfloor \mathscr{C}_3[\mathscr{C}_1][\tau = \mathfrak{g}] \rfloor$	$\Longrightarrow$ I
$\phi[\alpha := \mathfrak{g}'] \vdash \lfloor \mathscr{C}_3[\mathscr{C}_1][\tau = \mathfrak{g}] \rfloor$	Simple inversion
$\text{shape}(\mathfrak{g}) = \varsigma$	$\Longrightarrow$ E on $\mathscr{C}_3[\mathscr{C}_1][\tau \,!\, \varsigma]$
☞ $(\exists \alpha. \mathscr{C}_3[\mathscr{C}_1])[\tau \,!\, \varsigma]$	Above

Case $\forall \alpha. \mathscr{C}_3$.

Similar to $\exists \alpha. \mathscr{C}_3$ case.

Case $\exists i^x. \mathscr{C}_3$.

Similar to $\exists \alpha. \mathscr{C}_3$ case.

Case $\text{let } x = \lambda \alpha. \mathscr{C}_3 \text{ in } C$.

$\mathscr{C}_1[\tau \,!\, \varsigma]$	Premise
$\mathscr{C}_3[\mathscr{C}_1][\tau \,!\, \varsigma]$	By *i.h.*
For all $\phi, \mathfrak{g}$	Definition of $(\text{let } x \ldots)[\tau \,!\, \varsigma]$
$\phi \vdash \text{let } x = \lambda \alpha. \lfloor \mathscr{C}_3[\mathscr{C}_1][\tau = \mathfrak{g}] \rfloor \text{ in } \lfloor C \rfloor$	$\Longrightarrow$ I
$\phi \vdash \exists \alpha. \lfloor \mathscr{C}_3[\mathscr{C}_1][\tau = \mathfrak{g}] \rfloor$	Simple inversion
$\phi[\alpha := \mathfrak{g}'] \vdash \lfloor \mathscr{C}_3[\mathscr{C}_1][\tau = \mathfrak{g}] \rfloor$	Simple inversion
$\text{shape}(\mathfrak{g}) = \varsigma$	$\Longrightarrow$ E on $\mathscr{C}_3[\mathscr{C}_1][\tau \,!\, \varsigma]$
☞ $(\text{let } x = \lambda \alpha. \mathscr{C}_3[\mathscr{C}_1] \text{ in } C)[\tau \,!\, \varsigma]$	Above

Case $\text{let } x = \lambda \alpha. C \text{ in } \mathscr{C}_3$.

Similar to $\text{let } x = \lambda \alpha. \mathscr{C}_3 \text{ in } C$ case.

Case $\text{let } x\ \alpha\ [\bar{\alpha}] = \mathscr{C}_3 \text{ in } C$.

Similar to $\text{let } x = \lambda \alpha. \mathscr{C}_3 \text{ in } C$ case.

Case $\text{let } x\ \alpha\ [\bar{\alpha}] = C \text{ in } \mathscr{C}_3$.

Similar to $\text{let } x = \lambda \alpha. C \text{ in } \mathscr{C}_3$ case.

$\square$

Lemma B.3 (Inversion of unicity).

(i) If $(\exists\alpha.\mathscr{C})[\tau\,!\,\varsigma]$, then $\mathscr{C}[\tau\,!\,\varsigma]$.

(ii) If $(\forall\alpha.\mathscr{C})[\tau\,!\,\varsigma]$, then $\mathscr{C}[\tau\,!\,\varsigma]$.

Proof. The definition of $\mathscr{C}[\tau\,!\,\varsigma]$ uses simple semantics on the erasure $\lfloor\mathscr{C}\rfloor$, so these results are easily shown by simple inversion. □

Lemma B.4 (Decanonicalization). *If* $\phi \Vdash C$, *then* $\phi \vdash C$.

Proof. Induction on the given derivation $\phi \Vdash C$ □

Theorem B.5 (Canonicalization). *If* $\phi \vdash C$, *then* $\phi \Vdash C$.

Proof. We proceed by induction on $\phi \vdash C$ with the measure $\|C\|$.

Case

$$\frac{}{\phi \vdash \mathsf{true}}\ \textsc{True}$$

☞ $\phi \Vdash \mathsf{true}$ immediate by Can-Base

Case

$$\frac{\phi(\tau_1) = \phi(\tau_2)}{\phi \vdash \tau_1 = \tau_2}\ \textsc{Unif}$$

Similar to the True case.

Case

$$\frac{\phi \vdash C_1 \qquad \phi \vdash C_2}{\phi \vdash C_1 \wedge C_2}\ \textsc{Conj}$$

$\phi \vdash C_1$ Premise
$\phi \vdash C_2$ Premise
$\phi \Vdash C_1$ By *i.h.*
$\phi \Vdash C_2$ By *i.h.*

By cases on $\phi \Vdash C_1, \phi \Vdash C_2$.

Subcase

$$\frac{\phi \vdash C_1 \qquad C_1\ \mathsf{simple}}{\phi \Vdash C_1}\ \textsc{Can-Base}$$

$$\frac{\phi \vdash C_2 \qquad C_2\ \mathsf{simple}}{\phi \Vdash C_2}\ \textsc{Can-Base}$$

☞ $\phi \Vdash C_1 \wedge C_2$ immediate by Can-Base

Subcase

$$\frac{\mathscr{C}[\tau\,!\,\varsigma] \qquad \phi \Vdash \mathscr{C}[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}]}{\phi \Vdash \underbrace{\mathscr{C}[\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}]}_{C_1}}\ \textsc{Can-Match-Ctx}$$

$\phi \Vdash C_2$

$\phi \Vdash \mathscr{C}[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}]$ Premise
$\phi \vdash \mathscr{C}[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}]$ Lemma B.4

$\phi \vdash \mathscr{C}[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}] \wedge C_2$	By Conj
$\phi \Vdash \mathscr{C}[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}] \wedge C_2$	By *i.h.*
$\mathscr{C}[\alpha\,!\,\varsigma]$	Premise
$(\mathscr{C} \wedge C_2)[\alpha\,!\,\varsigma]$	Lemma B.2
☞ $\phi \Vdash \mathscr{C}[\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}]$	By Can-Match-Ctx

Subcase

$\phi \Vdash C_1$

$$\frac{\mathscr{C}[\tau\,!\,\varsigma] \qquad \phi \Vdash \mathscr{C}[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}]}{\phi \Vdash \underbrace{\mathscr{C}[\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}]}_{C_2}}\ \textsc{Can-Match-Ctx}$$

Symmetric to the above case.

Case

$$\frac{\phi[\alpha := \mathfrak{g}] \vdash C}{\phi \vdash \exists \alpha.\, C}\ \textsc{Exists}$$

$\phi[\alpha := \mathfrak{g}] \vdash C$	Premise
$\phi[\alpha := \mathfrak{g}] \Vdash C$	By *i.h.*

By cases on $\phi[\alpha := \mathfrak{g}] \Vdash C$.

Subcase

$$\frac{\phi[\alpha := \mathfrak{g}] \vdash C \qquad C\ \text{simple}}{\phi[\alpha := \mathfrak{g}] \Vdash C}\ \textsc{Can-Base}$$

☞ $\phi \Vdash \exists \alpha.\, C$ Immediate by Can-Base

Subcase

$$\frac{\mathscr{C}[\tau\,!\,\varsigma] \qquad \phi[\alpha := \mathfrak{g}] \Vdash \mathscr{C}[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}]}{\phi \Vdash \underbrace{\mathscr{C}[\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}]}_{C}}\ \textsc{Can-Match-Ctx}$$

$\phi[\alpha := \mathfrak{g}] \Vdash \mathscr{C}[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}]$	Premise
$\phi[\alpha := \mathfrak{g}] \vdash \mathscr{C}[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}]$	Lemma B.4

$\phi \vdash \exists \alpha.\, \mathscr{C}[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}]$	By Exists
$\phi \Vdash \exists \alpha.\, \mathscr{C}[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}]$	By *i.h.*
$\mathscr{C}[\tau\,!\,\varsigma]$	Premise
$(\exists \alpha.\, \mathscr{C})[\tau\,!\,\varsigma]$	Lemma B.2
☞ $\phi \Vdash \exists \alpha.\, \mathscr{C}[\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}]$	By Can-Match-Ctx

Case

$$\frac{\forall \mathfrak{g},\ \phi[\alpha := \mathfrak{g}] \vdash C}{\phi \vdash \forall \alpha.\, C}\ \textsc{Forall}$$

Similar to the Exists case.

Case

$$\frac{\phi[x := \mathfrak{S}] \vdash C_2 \qquad \mathfrak{S} = \phi(\lambda\alpha.\, C_1) \qquad \mathfrak{S} \neq \emptyset}{\phi \vdash \text{let } x = \lambda\alpha.\, C_1 \text{ in } C_2}\ \textsc{Let}$$

$\phi[x := \mathfrak{S}] \Vdash C_2$ By *i.h.*

By cases on $\phi[x := \mathfrak{S}] \Vdash C_2$.

Subcase

$$\frac{\phi[x := \mathfrak{S}] \vdash C_2 \qquad C_2 \text{ simple}}{\phi[x := \mathfrak{S}] \Vdash C_2}\ \textsc{Can-Base}$$

☞ $\phi \Vdash \text{let } x = \lambda\alpha.\, C_1 \text{ in } C_2$ Immediate by Can-Base

Subcase

$$\frac{\mathscr{C}[\tau\,!\,\varsigma] \qquad \phi[x := \mathfrak{S}] \Vdash \mathscr{C}[\text{match } \tau := \varsigma \text{ with } \bar{\chi}]}{\phi[x := \mathfrak{S}] \Vdash \underbrace{\mathscr{C}[\text{match } \tau \text{ with } \bar{\chi}]}_{C_2}}\ \textsc{Can-Match-Ctx}$$

$\mathscr{C}[\tau\,!\,\varsigma]$	Premise
$(\text{let } x = \lambda\alpha.\, C_1 \text{ in } \mathscr{C})[\tau\,!\,\varsigma]$	Lemma B.2
$\phi[x := \mathfrak{S}] \Vdash \mathscr{C}[\text{match } \tau := \varsigma \text{ with } \bar{\chi}]$	Premise
$\phi[x := \mathfrak{S}] \vdash \mathscr{C}[\text{match } \tau := \varsigma \text{ with } \bar{\chi}]$	Lemma B.4
$\phi \vdash \text{let } x = \lambda\alpha.\, C_1 \text{ in } \mathscr{C}[\text{match } \tau := \varsigma \text{ with } \bar{\chi}]$	By Let
$\phi \Vdash \text{let } x = \lambda\alpha.\, C_1 \text{ in } \mathscr{C}[\text{match } \tau := \varsigma \text{ with } \bar{\chi}]$	By *i.h.*
☞ $\phi \Vdash \text{let } x = \lambda\alpha.\, C_1 \text{ in } \mathscr{C}[\text{match } \tau \text{ with } \varsigma]$	By Can-Match-Ctx

Case

$$\frac{\phi(\tau) \in \phi(x)}{\phi \vdash x\ \tau}\ \textsc{App}$$

Similar to the True case.

Case

$$\frac{\mathscr{C}[\tau\,!\,\varsigma] \qquad \phi \vdash \mathscr{C}[\text{match } \tau := \varsigma \text{ with } \bar{\chi}]}{\phi \vdash \mathscr{C}[\text{match } \tau \text{ with } \bar{\chi}]}\ \textsc{Match-Ctx}$$

$\mathscr{C}[\tau\,!\,\varsigma]$	Premise
$\phi \Vdash \mathscr{C}[\text{match } \tau := \varsigma \text{ with } \bar{\chi}]$	Premise
$\phi \vdash \mathscr{C}[\text{match } \tau := \varsigma \text{ with } \bar{\chi}]$	By *i.h.*
☞ $\phi \vdash \mathscr{C}[\text{match } \tau \text{ with } \bar{\chi}]$	By Can-Match-Ctx

Case

$$\frac{\phi[x := \mathfrak{R}] \vdash C_2 \qquad \mathfrak{R} = \phi(\lambda\alpha[\bar{\alpha}].\, C_1) \qquad \mathfrak{R} \neq \emptyset}{\phi \vdash \text{let } x\ \alpha\ [\bar{\alpha}] = C_1 \text{ in } C_2}\ \textsc{LetR}$$

Similar to the Let case.

Case

$$\frac{(\phi(\tau), \_) \in \phi(x)}{\phi \vdash x\ \tau}\ \textsc{AppR}$$

Similar to the App case.

Case

$$\frac{\begin{array}{l}(\_, \phi') \in \phi(x)\\ \phi[i := \phi'] \vdash C\end{array}}{\phi \vdash \exists i^x .\, C}\ \textsc{Exists-Inst}$$

Similar to the Exists case.

Case

$$\frac{\forall \tau \in \epsilon,\ \phi(\tau) = \mathfrak{g}}{\phi \vdash \epsilon}\ \textsc{Multi-Unif}$$

Similar to the Unif case.

Case

$$\frac{\phi(i)(\alpha) = \phi(\tau)}{\phi \vdash i(\alpha) \rightsquigarrow \tau}\ \textsc{Incr-Inst}$$

Similar to the App case.

□

Lemma B.6 (Inversion of suspension). *If* $\phi \vdash \mathscr{C}[\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}]$ *and* $\mathscr{C}[\tau\,!\,\varsigma]$, *then* $\phi \vdash \mathscr{C}[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}]$.

Proof. We use canonicalization (Theorem B.5) to induct on $\phi \Vdash \mathscr{C}[\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}]$ instead of $\phi \vdash \mathscr{C}[\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}]$.

Case

$$\frac{\phi \vdash \mathscr{C}[\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}] \qquad \mathscr{C}[\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}]\ \mathsf{simple}}{\phi \Vdash \mathscr{C}[\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}]}\ \textsc{Can-Base}$$

The second premise is a contradiction.

Case

$$\frac{\mathscr{C}'[\tau'\,!\,\varsigma'] \qquad \phi \Vdash \mathscr{C}'[\mathsf{match}\ \tau' := \varsigma'\ \mathsf{with}\ \bar{\chi}']}{\phi \Vdash \underbrace{\mathscr{C}'[\mathsf{match}\ \tau'\ \mathsf{with}\ \bar{\chi}']}_{\mathscr{C}[\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}]}}\ \textsc{Can-Match-Ctx}$$

By cases on $\mathscr{C} = \mathscr{C}'$.

Subcase $\mathscr{C} = \mathscr{C}'$.

	$\mathscr{C} = \mathscr{C}'$	Premise
	$\tau' = \tau$	
	$\varsigma' = \varsigma$	
	$\bar{\chi}' = \bar{\chi}$	
☞	$\phi \Vdash \mathscr{C}[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}]$	Premise

Subcase $\mathscr{C} \neq \mathscr{C}'$.

$$
\begin{array}{ll}
\mathscr{C}_2[\text{match } \tau \text{ with } \bar{\chi}, \text{match } \tau' \text{ with } \bar{\chi}'] & \\
\quad = \mathscr{C}[\text{match } \tau \text{ with } \bar{\chi}] & \text{For some 2-hole context } \mathscr{C}_2 \\
\quad = \mathscr{C}'[\text{match } \tau' \text{ with } \bar{\chi}'] &
\end{array}
$$

$\phi \Vdash \mathscr{C}_2[\text{match } \tau \text{ with } \bar{\chi}, \text{match } \tau' := \varsigma' \text{ with } \bar{\chi}']$ Premise

For all $\phi', \mathfrak{g}'$ Defn. of $\mathscr{C}_2[\square, \text{match } \tau' := \varsigma' \text{ with } \bar{\chi}'][\tau \,!\, \varsigma]$

$$
\begin{array}{ll}
\phi' \vdash \lfloor \mathscr{C}_2[\tau = \mathfrak{g}', \text{match } \tau' := \varsigma' \text{ with } \bar{\chi}'] \rfloor & \Longrightarrow \text{I} \\
\phi' \vdash \lfloor \mathscr{C}_2[\tau = \mathfrak{g}', \text{true}] \rfloor & \text{Lemma B.1}
\end{array}
$$

$$
\begin{array}{rll}
\lfloor \mathscr{C}_2[\tau = \mathfrak{g}', \text{true}] \rfloor = & \lfloor \mathscr{C}_2[\tau = \mathfrak{g}', \lfloor \text{match } \tau' \text{ with } \bar{\chi}' \rfloor] \rfloor & \text{By definition} \\
= & \lfloor \mathscr{C}[\tau = \mathfrak{g}'] \rfloor & \text{By definition} \\
\phi' \vdash & \lfloor \mathscr{C}[\tau = \mathfrak{g}'] \rfloor & \text{Above} \\
\text{shape}(\mathfrak{g}') = & \varsigma & \Longrightarrow \text{E on } \mathscr{C}[\tau \,!\, \varsigma] \\
& \mathscr{C}_2[\square, \text{match } \tau' := \varsigma' \text{ with } \bar{\chi}'][\tau \,!\, \varsigma] & \text{Above} \\
\phi \Vdash & \mathscr{C}_2[\text{match } \tau := \varsigma \text{ with } \bar{\chi}, \text{match } \tau' := \varsigma' \text{ with } \bar{\chi}'] & \text{By } \textit{i.h.}
\end{array}
$$

For all $\phi', \mathfrak{g}'$ Defn. of $\mathscr{C}_2[\text{match } \tau := \varsigma \text{ with } \bar{\chi}, \square][\tau' \,!\, \varsigma']$

$$
\begin{array}{rll}
\phi' \vdash & \lfloor \mathscr{C}_2[\text{match } \tau := \varsigma \text{ with } \bar{\chi}, \tau' = \mathfrak{g}'] \rfloor & \Longrightarrow \text{I} \\
\phi' \vdash & \lfloor \mathscr{C}_2[\text{true}, \tau' = \mathfrak{g}'] \rfloor & \text{Lemma B.1} \\
\lfloor \mathscr{C}_2[\text{true}, \tau' = \mathfrak{g}'] \rfloor = & \lfloor \mathscr{C}_2[\lfloor \text{match } \tau \text{ with } \bar{\chi} \rfloor, \tau' = \mathfrak{g}'] \rfloor & \text{By definition} \\
= & \lfloor \mathscr{C}'[\tau' = \mathfrak{g}'] \rfloor & \text{By definition} \\
\phi' \vdash & \lfloor \mathscr{C}[\tau = \mathfrak{g}'] \rfloor & \text{Above} \\
& \mathscr{C}'[\tau' \,!\, \varsigma'] & \text{Premise} \\
\text{shape}(\mathfrak{g}') = & \varsigma' & \Longrightarrow \text{E on } \mathscr{C}'[\tau' \,!\, \varsigma'] \\
& \mathscr{C}_2[\text{match } \tau := \varsigma \text{ with } \bar{\chi}, \square][\tau' \,!\, \varsigma'] & \text{Above} \\
☞ \quad \phi \Vdash & \mathscr{C}_2[\text{match } \tau := \varsigma \text{ with } \bar{\chi}, \text{match } \tau' \text{ with } \bar{\chi}'] & \text{By Can-Match-Ctx}
\end{array}
$$

□

C Properties of the constraint solver

The primary requirement of our constraint solver is correctness: a constraint C is satisfiable if and only if the solver terminates with a solution.

This section decomposes this requirement into three properties: preservation, progress, and termination—and provides proofs for each. Correctness then follows as a corollary of these results.

C.1 Preservation

This section details the proof of *preservation* for the solver: if $C_1 \longrightarrow C_2$, then $C_1 \equiv C_2$. Since rewriting may occur under arbitrary contexts, it suffices to check for each rule, that the equivalence $C_1 \equiv C_2$ holds under all contexts $\mathscr{C}$.

However, the introduction of suspended match constraints breaks congruence of equivalence. That is, it is no longer the case that $C_1 \equiv C_2$ implies $\mathscr{C}[C_1] \equiv \mathscr{C}[C_2]$. For instance, we have $\text{match } \alpha \text{ with } \bar{\chi} \equiv \text{false}$, yet $\mathscr{C}[\text{match } \alpha \text{ with } \bar{\chi}] \not\equiv \mathscr{C}[\text{false}]$ for $\mathscr{C} := \square \wedge \alpha = \text{int}$.

As a result, we must prove *contextual equivalence* for each rewriting rule explicitly. This is both non-trivial and tedious. To simplify the task, we first present a series of auxiliary lemmas that recover contextual equivalence for many common cases. Whenever possible, we prefer to work with equivalences on *simple* constraints, as these retain the desired congruence properties that do not hold generally in our system.

Definition C.1 (Contextual eqiuvalence). Two constraints C_1 and C_2 are contextually equivalence, written $C_1 \equiv_{\text{ctx}} C_2$, iff:

$$C_1 \equiv_{\text{ctx}} C_2 \triangleq \forall \mathscr{C}.\ \mathscr{C}[C_1] \equiv \mathscr{C}[C_2]$$

Corollary C.2 (Simple equivalence is congruent). *Given simple constraints C_1, C_2 and simple context $\mathscr{C}$. If $C_1 \equiv C_2$, then $\mathscr{C}[C_1] \equiv \mathscr{C}[C_2]$.*

Proof. Follows from Lemma B.1. □

Lemma C.3 (Simple equivalence is contextual). *For simple constraints C_1, C_2. If $C_1 \equiv C_2$, then $C_1 \equiv_{\text{ctx}} C_2$.*

Proof. We proceed by induction on the number of suspended match constraints n in $\mathscr{C}$.

Case *n is 0.* Follows from Corollary C.2.

Case *n is $k + 1$.*

Subcase $\Longrightarrow$.

$\phi \vdash \mathscr{C}[C_1]$	Premise
$\phi \Vdash \mathscr{C}[C_1]$	Theorem B.5
$\mathscr{C}'[\tau \,!\, \varsigma]$	Inversion of Can-Match-Ctx
$\phi \Vdash \mathscr{C}'[\text{match } \tau := \varsigma \text{ with } \bar{\chi}]$	″
$\mathscr{C}[C_1] = \mathscr{C}'[\text{match } \tau \text{ with } \bar{\chi}]$	″
$= \mathscr{C}_2[\text{match } \tau \text{ with } \bar{\chi}, C_1]$	For some two-hole context $\mathscr{C}_2$
$\phi \vdash \mathscr{C}_2[\text{match } \tau := \varsigma \text{ with } \bar{\chi}, C_2]$	By *i.h.*
For all $\phi', \mathfrak{g}$	Defn of $\mathscr{C}'[\tau \,!\, \varsigma]$
$\phi' \vdash \lfloor \mathscr{C}_2[\tau := \mathfrak{g}, C_2] \rfloor$	Premise
$\phi' \vdash \lfloor \mathscr{C}_2[\tau := \mathfrak{g}, C_1] \rfloor$	Corollary C.2
$\phi' \vdash \lfloor \mathscr{C}'[\tau := \mathfrak{g}] \rfloor$	Above
$\text{shape}(\mathfrak{g}) = \varsigma$	$\Longrightarrow$ E on $\mathscr{C}'[\tau \,!\, \varsigma]$

$\mathscr{C}_2[\square, C_2][\tau \,!\, \varsigma]$	Above
☞ $\phi \vdash \mathscr{C}_2[\text{match } \tau \text{ with } \bar{\chi}, C_2]$	By Match-Ctx

Subcase $\Longleftarrow$.

Symmetric argument.

□

Lemma C.4 (Unification is simple). *For all unification problems U, U* simple.

Proof. By induction on the structure of U. □

Definition C.5 (Context equivalence). Two contexts $\mathscr{C}_1$ and $\mathscr{C}_2$ are equivalent with guard P, written $\mathscr{C}_1 \equiv^P_\square \mathscr{C}_2$ iff:

$$\mathscr{C}_1 \equiv^P_\square \mathscr{C}_2 \triangleq \forall \bar{C}.\ P(\bar{C}) \Longrightarrow \mathscr{C}_1[\bar{C}] \equiv_{\text{ctx}} \mathscr{C}_2[C]s$$

Definition C.6 (Match-closed). A predicate P on constraints is *match-closed* if, for all constraints $\bar{C}, \bar{C}'$, contexts $\mathscr{C}$, matches match τ with $\bar{\chi}$ and shapes ς,

$$P(\bar{C}, \mathscr{C}[\text{match } \tau \text{ with } \bar{\chi}], \bar{C}') \Longrightarrow P(\bar{C}, \mathscr{C}[\text{match } \tau := \varsigma \text{ with } \bar{\chi}], \bar{C}')$$

Lemma C.7 (Determines is match-closed). *C determines $\bar{\beta}$ is match-closed. Similarly, $\vdash C$ determines $\bar{\beta}$ is matched closed.*

Proof. Follows from the definitions of C determines $\bar{\beta}$, $\vdash C$ determines $\bar{\beta}$, and Lemma B.1. □

Lemma C.8 (Simple context equivalence). *For any two simple contexts $\mathscr{C}_1, \mathscr{C}_2$ and a match-closed guard P. If the two contexts $\mathscr{C}_1$ and $\mathscr{C}_2$ are equivalent under any simple constraints satisfying P, then $\mathscr{C}_1 \equiv^P_\square \mathscr{C}_2$.*

Proof. Let us assume that (†) holds:

$$\forall \mathscr{C}, \bar{C} \text{ simple}. P(\bar{C}) \implies \mathscr{C}[\mathscr{C}_1[\bar{C}]] \equiv \mathscr{C}[\mathscr{C}_2[\bar{C}]]$$

We proceed by induction on the number of suspended match constraints n with the statement $Q(n) := \forall \bar{C}, \mathscr{C}. \#\text{match } \mathscr{C} + \#\text{match } \bar{C} = n \implies P(\bar{C}) \implies \mathscr{C}[\mathscr{C}_1[\bar{C}]] \equiv \mathscr{C}[\mathscr{C}_2[\bar{C}]]$.

Case *n is 0.*

$\mathscr{C}, \bar{C}$ simple	Premise (n is 0)
☞ $P(\bar{C}) \implies \mathscr{C}[\mathscr{C}_1][\bar{C}] \equiv \mathscr{C}[\mathscr{C}_2][\bar{C}]$	†

Case *n is $k+1$.*

Subcase $\implies$.

$P(\bar{C})$	Premise
$\phi \vdash \mathscr{C}[\mathscr{C}_1][\bar{C}]$	Premise
$\phi \Vdash \mathscr{C}[\mathscr{C}_1][\bar{C}]$	Theorem B.5
$\phi \Vdash \mathscr{C}'[\text{match } \tau := \varsigma \text{ with } \bar{\chi}]$	Inversion of Can-Match-Ctx
$\mathscr{C}'[\tau \,!\, \varsigma]$	″
$\mathscr{C}[\mathscr{C}_1][\bar{C}] = \mathscr{C}'[\text{match } \tau \text{ with } \bar{\chi}]$	″

Cases on $\mathscr{C}, \bar{C}$.

Subsubcase *$\mathscr{C}$ contains $\mathscr{C}'$'s hole.*

$\mathscr{C}[\mathscr{C}_1][\bar{C}] = \mathscr{C}_3[\text{match } \tau \text{ with } \bar{\chi}, \mathscr{C}_1[\bar{C}]]$	For some 2-hole context $\mathscr{C}_3$
$\phi \Vdash \mathscr{C}_3[\text{match } \tau := \varsigma \text{ with } \bar{\chi}, \mathscr{C}_1[\bar{C}]]$	
$k = \#\text{match } \mathscr{C}_3[\text{match } \tau := \varsigma \text{ with } \bar{\chi}, \mathscr{C}_1[\bar{C}]]$	
$\phi \vdash \mathscr{C}_3[\text{match } \tau := \varsigma \text{ with } \bar{\chi}, \mathscr{C}_2[\bar{C}]]$	By *i.h.*

For all $\phi', \mathfrak{g}$

$\phi' \vdash \lfloor \mathscr{C}_3[\tau = \mathfrak{g}, \mathscr{C}_2[\bar{C}]] \rfloor$	Premise
$\phi' \vdash \lfloor \mathscr{C}_3[\tau = \mathfrak{g}, \mathscr{C}_1[\bar{C}]] \rfloor$	†
$\text{shape}(\mathfrak{g}) = \varsigma$	$\implies$ E on $\mathscr{C}'[\tau \,!\, \varsigma]$
$\mathscr{C}_3[\square, \mathscr{C}_2[\bar{C}]][\tau \,!\, \varsigma]$	Above
☞ $\phi \vdash \mathscr{C}_3[\text{match } \tau \text{ with } \bar{\chi}, \mathscr{C}_2[\bar{C}]]$	By Match-Ctx

Subsubcase *C_i contains $\mathscr{C}'$'s hole.*

Similar argument to the above case, but relies on the match-closure of P.

Subcase $\impliedby$.

Symmetric argument.

□

Lemma C.9 (Simple let equivalence). *Given simple constraints C_1, C_2 and a simple context $\mathscr{C}$. Suppose that*

$$\forall \phi, \phi', \bar{C} \text{ simple}.\ \phi'(x) = \phi(\lambda\alpha[\bar{\alpha}].\mathscr{C}[\bar{C}]) \implies \phi' \vdash C_1 \iff \phi' \vdash C_2$$

Then, for any context $\mathscr{C}'$ that does not re-bind x, we have:

$$\text{let } x\ \alpha\ [\bar{\alpha}] = \mathscr{C}[\bar{\square}] \text{ in } \mathscr{C}'[C_1] \equiv^P_\square \text{let } x\ \alpha\ [\bar{\alpha}] = \mathscr{C}[\bar{\square}] \text{ in } \mathscr{C}'[C_2]$$

for any match-closed guard P on the holes.

Proof. Let us pose $\mathfrak{R} = \phi(\lambda\alpha[\bar{\alpha}].\mathscr{C}[\bar{C}])$ and let us assume (†):

$$\forall \phi, \phi', \bar{C}.\ \phi'(x) = \mathfrak{R} \implies \phi' \vdash C_1 \iff \phi' \vdash C_2$$

We proceed by induction on the number of suspended match constraints in $\mathscr{C}'', \mathscr{C}', \bar{C}$ with the statement $P(n) := \forall \mathscr{C}'', \mathscr{C}', \bar{C}.\ \#\text{match}\ \mathscr{C}'', \mathscr{C}', \bar{C} = n \implies \mathscr{C}''[\text{let } x\ \alpha\ [\bar{\alpha}] = \mathscr{C}[\bar{C}] \text{ in } \mathscr{C}'[C_1]] \equiv \mathscr{C}''[\text{let } x\ \alpha\ [\bar{\alpha}] = \mathscr{C}[\bar{C}] \text{ in } \mathscr{C}'[C_2]]$.

Case *n is 0.*

Thus $\mathscr{C}'', \mathscr{C}', \bar{C}$ are simple. It suffices to show the equivalence on the let-constraint directly and use congruence of equivalence for simple constraints (Lemma C.3) to establish the result.
We proceed by induction on the structure of $\mathscr{C}'$ with the statement (‡):

$$\forall \phi, \phi'.\ \phi'(x) = \mathfrak{R} \implies \phi' \vdash \mathscr{C}'[C_1] \iff \phi' \vdash \mathscr{C}'[C_2]$$

This holds due to the compositionality of simple equivalence using † as a base case.

Subcase $\implies$.

$\phi \vdash \text{let } x\ \alpha\ [\bar{\alpha}] = \mathscr{C}[\bar{C}] \text{ in } \mathscr{C}'[C_1]$	Premise
$\phi[x := \mathfrak{R}] \vdash \mathscr{C}'[C_1]$	″
$\phi[x := \mathfrak{R}] \vdash \mathscr{C}'[C_2]$	‡
$\phi \vdash \text{let } x\ \alpha\ [\bar{\alpha}] = \mathscr{C}[\bar{C}] \text{ in } \mathscr{C}'[C_2]$	By LetR

Subcase $\impliedby$.

Symmetric argument.

Case *n is $k + 1$.*

Analogous to the inductive step in Lemma C.8.

□

Lemma C.10. *If* $\vdash \mathscr{C}[\tau\,!\,\varsigma]$, *then* $\mathscr{C}[\tau\,!\,\varsigma]$.

Proof. **Case**

$$\frac{\tau \notin \mathcal{V}}{\vdash \mathscr{C}[\tau\,!\ \text{shape}(\tau)]}\ \text{S-Uni-Type}$$

$\tau \notin \mathcal{V}$ Premise

$\tau = \text{shape}(\tau)\ \bar{\tau}$	For some $\bar{\tau}$
For all $\phi, \mathfrak{g}$	Defn. of $\mathscr{C}[\tau\,!\ \text{shape}(\tau)]$
$\phi \vdash \lfloor \mathscr{C}[\tau = \mathfrak{g}] \rfloor$	Premise
$\phi_1 \vdash \tau = \mathfrak{g}$	Inversion of $\lfloor \mathscr{C}_1 \rfloor$
$\mathfrak{g} = \phi_1(\tau)$	Simple inversion
$= \text{shape}(\tau)\ \phi_1(\bar{\tau})$	″
☞ $\text{shape}(\mathfrak{g}) = \text{shape}(\tau)$	Applying shape to both sides

Case

$$\dfrac{\alpha \mathbin{\#} \mathrm{bv}(\mathscr{C}_2)}{\vdash \mathscr{C}_1[\alpha = \tau = \epsilon \land \mathscr{C}_2[-]][\alpha \,!\; \mathrm{shape}(\tau)]}\ \textsc{S-Uni-Var}$$

$\alpha \mathbin{\#} \mathrm{bv}(\mathscr{C}_2)$	Premise
$\tau = \mathrm{shape}(\tau)\,\bar{\tau}$	For some $\bar{\tau}$
For all $\phi, \mathfrak{g}$	Defn. of $\mathscr{C}[\alpha \,!\; \mathrm{shape}(\tau)]$
$\phi \vdash \lfloor \mathscr{C}_1[\alpha = \mathrm{shape}(\tau)\,\bar{\tau} = \epsilon \land \mathscr{C}_2[\alpha = \mathfrak{g}]] \rfloor$	Premise
$\phi_1 \vdash \alpha = \mathrm{shape}(\tau)\,\bar{\tau} = \epsilon$	Inversion of $\lfloor \mathscr{C}_1 \rfloor$
$\phi_2 \vdash \alpha = \mathfrak{g}$	Inversion of $\lfloor \mathscr{C}_2 \rfloor$
$\mathfrak{g} = \phi_2(\alpha)$	Simple inversion
$= \phi_1(\alpha)$	$\alpha \mathbin{\#} \mathrm{bv}(\mathscr{C}_2)$
$= \mathrm{shape}(\tau)\,\phi_1(\bar{\tau})$	Simple inversion
☞ $\mathrm{shape}(\mathfrak{g}) = \mathrm{shape}(\tau)$	Applying shape to both sides

Case

$$\dfrac{\vdash \mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = \mathscr{C}_1[\mathsf{true}]\ \mathsf{in}\ \mathscr{C}_2[i^x(\alpha') \rightsquigarrow \gamma \land -][\gamma \,!\, \varsigma] \qquad \alpha' \in \alpha, \bar{\alpha} \qquad x \mathbin{\#} \mathrm{bv}(\mathscr{C}_2) \qquad \alpha' \mathbin{\#} \mathrm{bv}(\mathscr{C}_1)}{\vdash \mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = \mathscr{C}_1[-]\ \mathsf{in}\ \mathscr{C}_2[i^x(\alpha') \rightsquigarrow \gamma][\alpha' \,!\, \varsigma]}\ \textsc{S-Uni-BackProp}$$

$\alpha' \in \alpha, \bar{\alpha}$	Premise
$x \mathbin{\#} \mathrm{bv}(\mathscr{C}_2)$	″
$\alpha' \mathbin{\#} \mathrm{bv}(\mathscr{C}_1)$	″
$\vdash \mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = \mathscr{C}_1[\mathsf{true}]\ \mathsf{in}\ \mathscr{C}_2[i^x(\alpha') \rightsquigarrow \gamma \land -][\gamma \,!\, \varsigma]$	″
$\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = \mathscr{C}_1[\mathsf{true}]\ \mathsf{in}\ \mathscr{C}_2[i^x(\alpha') \rightsquigarrow \gamma \land -][\gamma \,!\, \varsigma]$	By *i.h.*

For all $\phi, \mathfrak{g}$	Defn. of $\ldots[\alpha \,!\; \mathrm{shape}(\tau)]$
$\phi \vdash \lfloor \mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = \mathscr{C}_1[\alpha' = \mathfrak{g}]\ \mathsf{in}\ \mathscr{C}_2[i^x(\alpha') \rightsquigarrow \gamma] \rfloor$	Premise
Let $\mathfrak{R} = \phi(\lambda\alpha[\bar{\alpha}].\, \lfloor \mathscr{C}_1[\alpha' = \mathfrak{g}] \rfloor)$.	
Let $\phi_1 = \phi[x := \mathfrak{R}]$.	
$\phi'(\alpha') = \mathfrak{g}$	For any $(\_, \phi') \in \phi_1(x)$
$\phi_2 \vdash i^x(\alpha') \rightsquigarrow \gamma$	Inversion of $\lfloor \mathscr{C}_2 \rfloor$
$\phi_2(i^x)(\alpha') = \phi_2(\gamma)$	Simple inversion
$\phi_2(i^x) \in \phi_2(x)$	Since $\exists i^x.\, \in \mathscr{C}_2$, ϕ_2 extends ϕ_1
$\phi_2(i^x)(\alpha') = \mathfrak{g}$	Above
$= \phi_2(\gamma)$	″

$\phi_1 \vdash \lfloor \mathscr{C}_2[i^x(\alpha') \rightsquigarrow \gamma \land \gamma = \mathfrak{g}] \rfloor$	Entailment for $\lfloor \mathscr{C}_2 \rfloor$
$\phi \vdash \lfloor \mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = \mathscr{C}_1[\alpha' = \mathfrak{g}]\ \mathsf{in}\ \mathscr{C}_2[i^x(\alpha') \rightsquigarrow \gamma \land \gamma = \mathfrak{g}] \rfloor$	By LetR
$\phi \vdash \mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = \mathscr{C}_1[\mathsf{true}]\ \mathsf{in}\ \mathscr{C}_2[i^x(\alpha') \rightsquigarrow \gamma \land \gamma = \mathfrak{g}]$	Simple congruence
☞ $\mathrm{shape}(\mathfrak{g}) = \varsigma$	$\Longrightarrow E$ on $\ldots[\gamma \,!\, \varsigma]$

□

Lemma C.11. *If $\mathscr{C}$ is normalized, then $\mathscr{C}[\tau \,!\, \varsigma]$ if and only $\vdash \mathscr{C}[\tau \,!\, \varsigma]$.*

Proof.

Case $\Longrightarrow$.

Let us assume $\mathscr{C}[\tau \,!\, \varsigma]$ and $\mathscr{C}$ is normalized.

Given $\mathscr{C}$ is normalized, every constraint in $\mathscr{C}$ is of the form:

$$R \quad ::= \quad \bar{\hat{\epsilon}} \wedge \overline{\mathsf{match}\ \alpha\ \mathsf{with}\ \bar{\chi}} \wedge \exists \overline{i^x}.\, \overline{i^x(\beta) \rightsquigarrow \gamma} \wedge \overline{\mathsf{let}\ x\ \delta\ [\bar{\delta}] = R_1\ \mathsf{in}\ R_2}$$

By assumptions, we have $\forall \phi, \mathfrak{g}.\, \phi \vdash \lfloor \mathscr{C} \rfloor[\alpha = \mathfrak{g}] \implies \mathsf{shape}(\mathfrak{g}) = \varsigma$. Hence $\lfloor \mathscr{C} \rfloor$ contains $\lfloor R \rfloor$ where:

$$\lfloor R \rfloor \quad ::= \quad \bar{\hat{\epsilon}} \wedge \exists \overline{i^x}.\, \overline{i^x(\beta) \rightsquigarrow \gamma} \wedge \overline{\mathsf{let}\ x\ \delta\ [\bar{\delta}] = \lfloor R_1 \rfloor\ \mathsf{in}\ \lfloor R_2 \rfloor}$$

w.l.o.g. all constraints that may determine the shape of α are located with the regional binder (following the S-Exists-Lower and S-Let-ConjLeft rules). There are two cases:

Subcase $\alpha = \tau = \epsilon \in \bar{\hat{\epsilon}}$. Apply S-Uni-Var.

Subcase *Otherwise.*

Since $\mathscr{C}$ is normalized, it must be that case that no equality constraint determines the shape of α. Since any such equality would normalize to $\alpha = \tau = \epsilon$, contradicting our assumption that $\mathscr{C}$ is normalized.

By elimination on the structure of R, the only constraints that could determine the shape of α are incremental instantiation constraints that copy α. So there exists a partial instantiation constraint $i^x(\alpha) \rightsquigarrow \gamma$ such that $\mathscr{C}'[i^x(\alpha) \rightsquigarrow \gamma] = \mathscr{C}[\mathsf{true}]$ and $\mathscr{C}'[\gamma \,!\, \varsigma]$.

By induction, we have $\vdash \mathscr{C}'[\gamma \,!\, \varsigma]$. From S-Uni-BackProp, we have $\vdash \mathscr{C}[\alpha \,!\, \varsigma]$.

Case $\Longleftarrow$. Follows from Lemma C.10.

□

Lemma C.12 (Unification preservation). *If* $U_1 \longrightarrow U_2$*, then* $U_1 \equiv U_2$

Proof. By induction on the given derivation $U_1 \longrightarrow U_2$. See Pottier and Rémy [2005] for more details. □

Theorem 5.9 (Preservation). *If* $C_1 \longrightarrow C_2$*, then* $C_1 \equiv C_2$*.*

Proof. We proceed by induction on the given derivation. It suffices to show that for each individual rule R ($C_1 \longrightarrow_R C_2$), that $C_1 \equiv_{\mathrm{ctx}} C_2$.

Case

$$\frac{U_1 \qquad U_1 \longrightarrow U_2}{U_2} \text{ S-Unif}$$

$U_1 \longrightarrow U_2$	Premise
$U_1 \equiv U_2$	Lemma C.12
U_1, U_2 simple	Lemma C.4
☞ $U_1 \equiv_{\mathrm{ctx}} U_2$	Lemma C.3

Case

$$\frac{(\exists \alpha.\, C_1) \wedge C_2 \qquad \alpha \,\#\, C_2}{\exists \alpha.\, C_1 \wedge C_2} \text{ S-Exists-Conj}$$

$\alpha \,\#\, C_2$ Premise

Sufficient to show equivalence for simple constraints.	Lemma C.8
Suppose C_1, C_2 simple.	Premise

Subcase $\Longrightarrow$.

For all ϕ

$\phi \vdash (\exists \alpha. C_1) \wedge C_2$	Premise
$\phi[\alpha := \mathfrak{g}] \vdash C_1$	Simple inversion
$\phi \vdash C_2$	Simple inversion
$\phi[\alpha := \mathfrak{g}] \vdash C_2$	$\alpha \# C_2$
$\phi[\alpha := \mathfrak{g}] \vdash C_1 \wedge C_2$	By Conj
☞ $\phi \vdash \exists \alpha. C_1 \wedge C_2$	By Exists

Subcase $\Longleftarrow$.

Symmetric argument.

Case *S-Let*, *S-True*, *S-False*, *S-Let-ExistsRight*, *S-Let-ConjLeft*, *S-Let-ConjRight*, *S-Inst-Name*, *S-Exists-Exists-Inst*, *S-Exists-Inst-Conj*, *S-Exists-Inst-Let*, *S-Exists-Inst-Solve*, *S-All-Conj*.

Similar argument to the S-Exists-Conj case.

Case

$$\dfrac{\text{let } x\ \alpha\ [\bar{\alpha}] = \exists \beta. C_1 \text{ in } C_2 \qquad \beta \# \alpha, \bar{\alpha}, C_2}{\text{let } x\ \alpha\ [\bar{\alpha}, \beta] = C_1 \text{ in } C_2}\ \text{S-Let-ExistsLeft}$$

If we have $\phi \vdash \text{let } x\ \alpha\ [\bar{\alpha}] = \exists \beta. C_1 \text{ in } C_2$, then for each instance $(i^x := \phi'_i) \in \phi$ we have $\phi'_i \vdash \exists \beta. C_1$. By simple inversion, we have $\phi'_i \vdash \exists \beta. C_1$ if and only if $\phi'_i[\beta := \mathfrak{g}_i] \vdash C_1$ for some $\mathfrak{g}_i$, and so our derivation

$$\phi[\overline{i^x := \phi'_i}] \vdash \text{let } x\ \alpha\ [\bar{\alpha}] = \exists \beta. C_1 \text{ in } C_2$$

corresponds to exactly one derivation

$$\phi[\overline{i^x := \phi'_i[\beta := \mathfrak{g}_i]}] \vdash \text{let } x\ \alpha\ [\bar{\alpha}, \beta] = C_1 \text{ in } C_2$$

Case

$$\dfrac{\mathscr{C}[\text{match } \tau \text{ with } \bar{\chi}] \qquad \vdash \mathscr{C}[\tau \,!\, \varsigma]}{\mathscr{C}[\text{match } \tau := \varsigma \text{ with } \bar{\chi}]}\ \text{S-Match-Ctx}$$

$\vdash \mathscr{C}[\tau \,!\, \varsigma]$	Premise
$\mathscr{C}[\tau \,!\, \varsigma]$	Lemma C.10
Sufficient to show equivalences between constraints.	Lemma B.2

Subcase $\Longrightarrow$.

For all ϕ

$\phi \vdash \mathscr{C}[\text{match } \alpha \text{ with } \bar{\chi}]$	Premise
☞ $\phi \vdash \mathscr{C}[\text{match } \alpha := \text{shape}(\tau) \text{ with } \bar{\chi}]$	Lemma B.6

Subcase $\Longleftarrow$.

For all ϕ

$\phi \vdash \mathscr{C}[\text{match } \alpha := \text{shape}(\tau) \text{ with } \bar{\chi}]$	Premise
☞ $\phi \vdash \mathscr{C}[\text{match } \alpha \text{ with } \bar{\chi}]$	By Match-Ctx

Case

$$\dfrac{\text{let } x\ \alpha\ [\bar{\alpha}] = C \text{ in } \mathscr{C}[x\ \tau] \qquad \gamma \# \tau \qquad x \# \text{bv}(\mathscr{C})}{\text{let } x\ \alpha\ [\bar{\alpha}] = C \text{ in } \mathscr{C}[\exists \gamma, i^x. \gamma = \tau \wedge i(\alpha) \rightsquigarrow \gamma]}\ \text{S-Let-AppR}$$

$\gamma \# \tau$	Premise
$x \# \text{bv}(\mathscr{C})$	Premise

Sufficient to show a simple equivalence between $x\ \tau$ and $\exists \gamma, i^x. \gamma = \tau \wedge i(\alpha) \rightsquigarrow \gamma$. Lemma C.9
Suppose $\phi'(x) = \phi(\lambda\alpha[\bar{\alpha}].C)$. Premise

Subcase $\Longrightarrow$.

$\phi' \vdash x\ \tau$	Premise
$(\phi'(\tau), \phi_1) \in \phi'(x)$	Simple inversion
$\phi'(\tau) = \phi_1(\alpha)$	Defn. of $\phi(\lambda\alpha[\bar{\alpha}].C)$
$\phi'[\gamma := \phi'(\tau), i := \phi_1] \vdash i(\alpha) \rightsquigarrow \gamma$	By Incr-Inst
$\phi'[\gamma := \phi'(\tau), i := \phi_1] \vdash \gamma = \tau$	By Unif
☞ $\phi' \vdash \exists \gamma, i^x. \gamma = \tau \wedge i(\alpha) \rightsquigarrow \gamma$	By Exists, Exists-Inst and Conj

Subcase $\Longleftarrow$.

Symmetric argument.

Case

$$\frac{\text{let } x\ \alpha\ [\bar{\alpha}] = C \text{ in } \mathscr{C}[i^x(\alpha') \rightsquigarrow \gamma] \quad C = C' \wedge \alpha' = \varsigma\,\bar{\beta} = \epsilon \quad \alpha' \in \alpha, \bar{\alpha} \quad \neg\text{cyclic}(C) \quad \bar{\beta}' \# \alpha', \gamma, \bar{\beta} \quad x \# \text{bv}(\mathscr{C})}{\text{let } x\ \alpha\ [\bar{\alpha}] = C \text{ in } \mathscr{C}[\exists \bar{\beta}'. \gamma = \varsigma\,\bar{\beta}' \wedge i^x(\bar{\beta}) \rightsquigarrow \bar{\beta}']}\ \text{S-Inst-Copy}$$

$x \# \text{bv}(\mathscr{C})$	Premise
$\bar{\beta}' \# \alpha', \gamma, \bar{\beta}$	Premise

Sufficient to show equivalence between $i^x(\alpha') \rightsquigarrow \gamma$ and $\exists \bar{\beta}'. \gamma = \varsigma\,\bar{\beta}' \wedge i^x(\bar{\beta}) \rightsquigarrow \bar{\beta}'$. Lemma C.9
Suppose $\phi'(x) = \phi(\lambda\alpha[\bar{\alpha}].C)$. Premise

Subcase $\Longrightarrow$.

$\phi' \vdash i^x(\alpha') \rightsquigarrow \gamma$	Premise
$(\mathfrak{g}, \phi_1) \in \phi'(x)$	$\exists i^x. \in \mathscr{C}$
$\phi'(i) = \phi_1$	"
$\phi'(\gamma) = \phi(i)(\alpha')$	Simple inversion
$= \phi_1(\alpha')$	Above
$\phi_1 \vdash C' \wedge \alpha' = \varsigma\,\bar{\beta} = \epsilon$	Above
$\phi_1 \vdash \alpha' = \varsigma\,\bar{\beta} = \epsilon$	Simple inversion
$\phi_1(\alpha') = \varsigma\,\phi_1(\bar{\beta})$	"
$\phi'(\gamma) = \varsigma\,\phi_1(\bar{\beta})$	Above
$\phi'[\bar{\beta}' := \phi_1(\bar{\beta})] \vdash \gamma = \varsigma\,\bar{\beta}'$	By Unif
$\phi'[\bar{\beta}' := \phi_1(\bar{\beta})] \vdash i^x(\bar{\beta}) \rightsquigarrow \bar{\beta}'$	By Incr-Inst
☞ $\phi' \vdash \exists \bar{\beta}'. \gamma = \varsigma\,\bar{\beta}' \wedge i^x(\bar{\beta}) \rightsquigarrow \bar{\beta}'$	By Exists and Conj

Subcase $\Longleftarrow$.

Symmetric argument.

Case

$$\frac{i(\alpha) \rightsquigarrow \gamma_1 \wedge i(\alpha) \rightsquigarrow \gamma_2}{i(\alpha) \rightsquigarrow \gamma_1 \wedge \gamma_1 = \gamma_2}\ \text{S-Inst-Unify}$$

Sufficient to show equivalence between $i(\alpha) \rightsquigarrow \gamma_1 \wedge i(\alpha) \rightsquigarrow \gamma_2$ and $i(\alpha) \rightsquigarrow \gamma_1 \wedge \gamma_1 = \gamma_2$. Lemma C.8

Subcase $\Longrightarrow$.

	$\phi \vdash i(\alpha) \rightsquigarrow \gamma_1 \wedge i(\alpha) \rightsquigarrow \gamma_2$	Premise
	$\phi \vdash i(\alpha) \rightsquigarrow \gamma_1$	Simple inversion
	$\phi \vdash i(\alpha) \rightsquigarrow \gamma_2$	″
	$\phi(\gamma_1) = \phi(i)(\alpha)$	″
	$\phi(\gamma_2) = \phi(i)(\alpha)$	″
	$\phi(\gamma_1) = \phi(\gamma_2)$	Above
	$\phi \vdash \gamma_1 = \gamma_2$	By Unif
☞	$\phi \vdash i(\alpha) \rightsquigarrow \gamma_1 \wedge \gamma_1 = \gamma_2$	By Conj

Subcase $\Longleftarrow$.

Symmetric argument.

Case

$$\frac{\text{let } x\ \alpha\ [\bar\alpha] = \bar\epsilon \wedge C \text{ in } \mathscr{C}[i^x(\beta) \rightsquigarrow \gamma] \qquad \forall\beta.\exists\bar\beta\setminus\beta.\bar\epsilon \equiv \text{true} \qquad \beta \in \bar\beta \subseteq \alpha,\bar\alpha \qquad \bar\beta \# C \qquad i(\beta) \# \text{insts}(\mathscr{C}) \qquad x \# \text{bv}(\mathscr{C})}{\text{let } x\ \alpha\ [\bar\alpha] = \bar\epsilon \wedge C \text{ in } \mathscr{C}[\text{true}]} \longrightarrow \textsc{S-Inst-Poly}$$

$\forall\beta.\exists\bar\beta\setminus\beta.\bar\epsilon \equiv \text{true}$	Premise
$\bar\beta \# C$	Premise
$i(\beta) \# \text{insts}(\mathscr{C})$	Premise
$x \# \text{bv}(\mathscr{C})$	Premise

Sufficient to show equivalence between $i^x(\beta) \rightsquigarrow \gamma$ and true.	Lemma C.9
Suppose $\phi'(x) = \phi(\lambda\alpha[\bar\alpha].\bar\epsilon \wedge C)$.	Premise

Subcase $\Longrightarrow$.

	$\phi' \vdash i^x(\beta) \rightsquigarrow \gamma$	Premise
☞	$\phi' \vdash \text{true}$	By True

Subcase $\Longleftarrow$.

$\phi' \vdash \text{true}$	Premise
$(\mathfrak{g}, \phi_1) \in \phi'(x)$	$\mathscr{C} = \mathscr{C}_1[\exists i^x.\mathscr{C}_2]$
$\phi'(i) = \phi_1$	″

By cases on $\phi_1(\beta)$.

Subsubcase $\phi_1(\beta) = \phi'(\gamma)$.

	$\phi_1(\beta) = \phi'(\gamma)$	Premise
☞	$\phi' \vdash i^x(\beta) \rightsquigarrow \gamma$	By Incr-Inst

Subsubcase $\phi_1(\beta) \neq \phi'(\gamma)$.

Let $\phi_2' = \phi_1[\beta := \phi'(\gamma)]$.	
$\phi_2' \vdash \exists \bar{\beta} \setminus \beta. \bar{\epsilon}$	$\forall \beta. \exists \bar{\beta} \setminus \beta. \bar{\epsilon} \equiv \mathsf{true}$
Let $\phi_2 = \phi_1[\beta := \phi'(\gamma), \bar{\beta} \setminus \beta := \bar{\mathfrak{g}}]$.	
$\phi_2 \vdash \bar{\epsilon}$	Simple inversion
$\phi_1 \vdash \bar{\epsilon} \wedge C$	By definition
$\phi_1 \vdash C$	Simple inversion
$\phi_2 \vdash C$	$\bar{\beta} \# C$
$\phi_2 \vdash \bar{\epsilon} \wedge C$	By Conj
$(\mathfrak{g}', \phi_2) \in \phi(x)$	By definition
Suppose $\phi_3 \vdash \mathscr{C}_2[\mathsf{true}]$.	Considering entailment on $\exists i^x$.
$\phi_3(i) = \phi_1$	"
$\phi_3[i := \phi_2] \vdash \mathscr{C}_2[\mathsf{true}]$	$i(\beta) \# \mathsf{insts}(\mathscr{C}_2)$
$\mathscr{D} :: \phi_3 \vdash \mathscr{C}_2[\mathsf{true}]$	By Exists-Inst

$\mathscr{D}$ is a derivation that satisfies $\phi_1(\beta) = \phi'(\gamma)$.

☞ So this case degenerates to the former case.

Case

$$\frac{\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = C\ \mathsf{in}\ \mathscr{C}[i^x(\beta) \rightsquigarrow \gamma] \qquad \beta \notin \alpha, \bar{\alpha} \qquad x, \beta \# \mathsf{bv}(\mathscr{C})}{\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = C\ \mathsf{in}\ \mathscr{C}[\beta = \gamma]} \longrightarrow \textsc{S-Inst-Mono}$$

$\beta \# \alpha, \bar{\alpha}$	Premise
$x, \beta \# \mathsf{bv}(\mathscr{C})$	Premise

Sufficient to show equivalence between $i^x(\beta) \rightsquigarrow \gamma$ and $\beta = \gamma$.	Lemma C.9
Suppose $\phi'(x) = \phi(\lambda\alpha[\bar{\alpha}].C)$.	Premise

Subcase $\Longrightarrow$.

$\phi' \vdash i^x(\beta) \rightsquigarrow \gamma$	Premise
$(\_, \phi_1) \in \phi(C)$	$\exists i^x. \in \mathscr{C}$
$\phi'(i) = \phi_1$	"
$\phi'(\gamma) = \phi_1(\beta)$	Simple inversion
$\phi_1(\beta) = \phi(\beta)$	$\beta \# \alpha, \bar{\alpha}$
$\phi'(\beta) = \phi(\beta)$	$\beta \# \mathsf{bv}(\mathscr{C})$
$\phi'(\gamma) = \phi'(\beta)$	Above
$\phi' \vdash \gamma = \beta$	By Unif

Subcase $\Longleftarrow$.

Symmetric argument.

Case

$$\frac{\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = \bar{\epsilon}\ \mathsf{in}\ C \qquad x \# C \qquad \exists \alpha, \bar{\alpha}. \bar{\epsilon} \equiv \mathsf{true}}{C} \longrightarrow \textsc{S-Let-Solve}$$

$x \# C$	Premise
$\exists \alpha, \bar{\alpha}. \bar{\epsilon} \equiv \mathsf{true}$	

Sufficient to show equivalence for simple constraints.	Lemma C.8
Suppose C simple.	Premise

Subcase $\Longrightarrow$.

For all ϕ

	$\phi \vdash \text{let } x\ \alpha\ [\bar{\alpha}] = \bar{\epsilon} \text{ in } C$	Premise
	$\phi \vdash \exists \alpha, \bar{\alpha}. \bar{\epsilon}$	Simple inversion
	$\phi[x := \phi(\lambda\alpha[\bar{\alpha}].\bar{\epsilon})] \vdash C$	"
☞	$\phi \vdash C$	$x \# C$

Subcase $\Longleftarrow$.

For all ϕ

	$\phi \vdash C$	Premise
	$\phi[x := \phi(\lambda\alpha[\bar{\alpha}].\bar{\epsilon})] \vdash C$	$x \# C$
	$\phi \vdash \exists \alpha, \bar{\alpha}. \bar{\epsilon}$	
☞	$\phi \vdash \text{let } x\ \alpha\ [\bar{\alpha}] = \bar{\epsilon} \text{ in } C$	By LetR

Case

$$\frac{\text{let } x\ \alpha\ [\bar{\alpha}, \bar{\beta}] = C_1 \text{ in } C_2 \qquad \exists \alpha, \bar{\alpha}. C_1 \text{ determines } \bar{\beta}}{\exists \bar{\beta}. \text{let } x\ \alpha\ [\bar{\alpha}] = C_1 \text{ in } C_2} \longrightarrow \text{S-Exists-Lower}$$

$\exists \alpha, \bar{\alpha}. C_1$ determines $\bar{\beta}$	Premise
Sufficient to show equivalence for simple constraints.	Lemma C.8 and Lemma C.7
Suppose C_1, C_2 simple.	Premise

Subcase $\Longrightarrow$.

$\phi \vdash \text{let } x\ \alpha\ [\bar{\alpha}, \bar{\beta}] = C_1 \text{ in } C_2$	Premise
$\phi \vdash \exists \alpha, \bar{\alpha}, \bar{\beta}. C_1$	Simple inversion
$\phi[x := \phi(\lambda\alpha[\bar{\alpha}, \bar{\beta}]. C_1)] \vdash C_2$	"
$\phi[\alpha := \mathfrak{g}, \bar{\alpha} := \bar{\mathfrak{g}}, \bar{\beta} := \bar{\mathfrak{g}}'] \vdash C_1$	"
$\phi[\bar{\beta} := \bar{\mathfrak{g}}'] \vdash \exists \alpha, \bar{\alpha}. C_1$	By Exists

Sufficient to show $\phi[x := \phi(\lambda\alpha[\bar{\alpha}, \bar{\beta}]. C_1)] = \phi[\bar{\beta} := \bar{\mathfrak{g}}'](\lambda\alpha[\bar{\alpha}]. C_1)$.

Subsubcase $\Longrightarrow$.

	$\phi[\alpha := \mathfrak{g}_1, \bar{\alpha} := \bar{\mathfrak{g}}_1, \bar{\beta} := \bar{\mathfrak{g}}_2] \vdash C_1$	Premise
	$\phi[\bar{\beta} := \bar{\mathfrak{g}}_2] \vdash \exists \alpha, \bar{\alpha}. C_1$	By Exists
	$\bar{\mathfrak{g}}_2 = \bar{\mathfrak{g}}'$	By definition of determines
☞	$\phi[\bar{\beta} := \bar{\mathfrak{g}}', \alpha := \mathfrak{g}_1, \bar{\alpha} := \bar{\mathfrak{g}}_1] \vdash C_1$	Above

Subsubcase $\Longleftarrow$.

Symmetric argument.

Subcase $\Longleftarrow$.

Symmetric argument.

Case *S-Compress, S-Gc, S-Exists-All, S-All-Escape, S-All-Rigid, S-All-Solve.*

Similar argument. Use Lemma C.8.

The simple equivalences are standard, see Pottier and Rémy [2005].

□

C.2 Progress

Lemma C.13 (Unification progress). *If unification problem U cannot take a step $U \longrightarrow U'$, then either:*

(i) U *is solved.*
(ii) U *is* false.

Proof. This is a standard result. See Pottier and Rémy [2005]. □

Theorem C.14 (Progress). *If constraint* C *cannot take a step* $C \longrightarrow C'$*, then either:*

(1) C *is solved.*
(2) C *is stuck, it is either: (a)* false*; (b)* $\hat{\mathscr{C}}[x\ \tau]$ *where* $x \# \hat{\mathscr{C}}$*; (c)* $\hat{\mathscr{C}}[i^x(\alpha) \rightsquigarrow \gamma]$ *where* $x \# \hat{\mathscr{C}}$ *and* $i(\alpha) \# \mathsf{insts}(\hat{\mathscr{C}})$*; (d) for every match constraint* $C = \hat{\mathscr{C}}[\mathsf{match}\ \alpha\ \mathsf{with}\ \bar{\chi}]$, $\hat{\mathscr{C}}[\alpha\,!\,\varsigma]$ *does not hold for any* ς*. Here,* $\hat{\mathscr{C}}$ *is a normalized context (Definition A.4).*

Proof. We proceed by induction on the structure of C. We focus on suspended match constraints, conjunctions, and let rules.

Case $\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}$. We have two cases:

Subcase *τ is a non-variable type.* Apply S-Match-Ctx using S-Uni-Type

Subcase *τ is a type variable α.*
We have $\square[\alpha\,\text{⚹}]$. It suffices that every match constraint in a context-reachable position $\hat{\mathscr{C}}[\mathsf{match}\ \alpha'\ \mathsf{with}\ \bar{\chi}]$ satisfies $\hat{\mathscr{C}}[\alpha'\,\text{⚹}]$. By the definition of constraint contexts, there is only one such $\hat{\mathscr{C}}$, namely $\square$, for which we already have $\square[\alpha\,\text{⚹}]$. Hence $\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}$ is stuck.

Case $C_1 \wedge C_2$. We begin by inducting on C_1 and C_2. Then we consider cases:

Subcase *C_1 (or C_2) take a step.* Apply congruence rewriting rule.

Subcase *C_1 (or C_2) is* true. Apply S-True.

Subcase *C_1 (or C_2) is* false. Apply S-False.

Subcase *C_1 (or C_2) begins with $\exists$.* Apply S-Exists-Conj.

Subcase *C_1, C_2 are solved.*
We either apply the above $\exists$ case, or both C_1 and C_2 are solved multi-equations $\bar{\epsilon}_1, \bar{\epsilon}_2$. We perform cases on this:

Subsubcase *$\bar{\epsilon}_1$ and $\bar{\epsilon}_2$ are mergable.* Apply U-Merge.

Subsubcase $\mathsf{cyclic}(\bar{\epsilon}_1, \bar{\epsilon}_2)$. Apply U-Cycle.

Subsubcase *Otherwise.* The conjunction $\bar{\epsilon}_1 \wedge \bar{\epsilon}_2$ is solved.

Subcase *C_1 and C_2 are stuck (and not* false*).*
w.l.o.g., consider cases C_1.

Subsubcase $\hat{\mathscr{C}}_1[x\ \tau]$. We have $x \# \mathsf{bv}(\hat{\mathscr{C}}_1)$.
$\hat{\mathscr{C}}_1[x\ \tau] \wedge C_2$ is stuck as we do not bind x in $\hat{\mathscr{C}}_1 \wedge C_2$.

Subsubcase $\hat{\mathscr{C}}_1[i^x(\alpha) \rightsquigarrow \gamma]$. We have $x \# \mathsf{bv}(\hat{\mathscr{C}}_1)$ and $i(\alpha) \# \mathsf{insts}(\hat{\mathscr{C}}_1)$.
If $i(\alpha) \in \mathsf{insts}(C_2)$ and $i \# \mathsf{bv}(\hat{C}_1)$, then apply S-Inst-Unify. It must be the case that we can apply S-Inst-Unify, otherwise, we could lift these instantiation constraints using S-Exists-Lower and S-Let-ConjLeft, contradicting that $\hat{\mathscr{C}}_1$ is stuck.
Otherwise, $x \# \mathsf{bv}(\hat{\mathscr{C}}_1 \wedge C_2)$, thus $\hat{\mathscr{C}}_1[i^x(\alpha) \rightsquigarrow \gamma]$ is stuck.

Subsubcase $\hat{\mathscr{C}}_1[\mathsf{match}\ \alpha'\ \mathsf{with}\ \bar{\chi}]$. We have $\mathscr{C}_1[\alpha'\,\text{⚹}]$.
Consider a match constraint $\mathsf{match}\ \alpha'\ \mathsf{with}\ \bar{\chi}$ in C_1.
If $\vdash [\hat{\mathscr{C}}_1[-] \wedge C_2][\alpha'\,!\,\varsigma]$. Then we can apply S-Match-Ctx.
Otherwise $\nvdash [\hat{\mathscr{C}}_1[-] \wedge C_2][\alpha'\,!\,\varsigma]$. We have Lemma C.11, so we are stuck and $(\mathscr{C}_1 \wedge C_2)[\alpha'\,\text{⚹}]$.

Case $\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = C_1\ \mathsf{in}\ C_2$. We begin by inducting on C_1 and C_2. Then we consider cases:

Subcase *C_1 (or C_2) take a step.* Apply congruence rewriting rule.

Subcase *C_1 (or C_2) is* false. Apply S-False.

Subcase *C_1 begins with $\exists$.* Apply S-Let-ExistsLeft

Subcase C_2 *begins with* $\exists$. Apply S-Let-ExistsRight
Subcase C_2 *begins with* $\wedge$ *with* $x \# $ *from conjunct*. Apply S-Let-ConjRight.
Subcase C_1 *begins with* $\wedge$ *with* $\alpha, \bar{\alpha} \#$ *from conjunct* . Try apply S-Let-ConjLeft
Subcase C_2 *begins with* $\exists i^{x'}.\,, x \neq x'$. Apply S-Exists-Inst-Let
Subcase $\alpha' \in \bar{\alpha}$ *is determined by* C_1. Apply S-Exists-Lower
Subcase C_2 *is solved*.
Thus C_2 must be true (due to above cases).
Subsubcase C_1 *is solved*. Thus C_1 must be $\bar{\epsilon}$.
There are two cases:

- $\exists \alpha, \bar{\alpha}.\, \bar{\epsilon} \equiv \mathsf{true}$. Apply S-Let-Solve.
- $\exists \alpha, \bar{\alpha}.\, \bar{\epsilon} \not\equiv \mathsf{true}$. It must be the case there is some β that dominates a α' in $\alpha, \bar{\alpha}$ in $\bar{\epsilon}$. Hence $\exists \alpha, \bar{\alpha} \setminus \alpha'.\, \bar{\epsilon}$ determines α'. So we can apply S-Exists-Lower.

Subsubcase C_1 *is stuck*.
The constraint $\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = C_1\ \mathsf{in}\ C_2$ remains stuck, since no additional term variable bindings occur for the scope of C_1, ruling out the instantiation cases. Additionally, we cannot apply backpropagation since C_2 is true.
Subcase C_2 *is stuck*.
Subsubcase $\hat{\mathscr{C}}[x\ \tau]$. We have $x \# \mathrm{bv}(\hat{\mathscr{C}})$.
Apply S-Let-AppR.
Subsubcase $\hat{\mathscr{C}}[i^x(\alpha') \rightsquigarrow \gamma]$. We have $x \# \mathrm{bv}(\hat{\mathscr{C}})$ or $i(\alpha') \# \mathrm{insts}(\hat{\mathscr{C}})$.

- $\alpha' \in \alpha, \bar{\alpha}$.
  We can either apply S-Inst-Copy or S-Compress if a multi-equation involving α' occurs in C_1.
  Otherwise, we consider cases where C_1 is solved or stuck.
  If C_1 is solved, then it must be of the form $\bar{\epsilon}$. There are two cases:
  - $\exists \alpha, \bar{\alpha}.\, \bar{\epsilon} \equiv \mathsf{true}$. As α' does not appear in the head position of any multi-equation in $\bar{\epsilon}$, it must be polymorphic. Thus $\forall \alpha'.\, \exists \alpha, \bar{\alpha} \setminus \alpha'.\, \bar{\epsilon} \equiv \mathsf{true}$. So we can apply S-Inst-Poly.
  - $\exists \alpha, \bar{\alpha}.\, \bar{\epsilon} \not\equiv \mathsf{true}$. Apply S-Exists-Lower (using the same logic as above).

  If C_1 is stuck, then neither stuck case regarding instantiations in C_1 is fixed, so in these cases the constraint remains stuck. If C_1 is stuck with $\hat{\mathscr{C}}'[\mathsf{match}\ \beta\ \mathsf{with}\ \bar{\chi}']$. Then either backpropagation (via S-Uni-BackProp and S-Match-Ctx) applies with an equation in $\hat{\mathscr{C}}$, or the entire constraint is stuck (by Lemma C.11).
- $\alpha' \notin \alpha, \bar{\alpha}$. Apply S-Inst-Mono.

Subsubcase *For any* $\hat{\mathscr{C}}[\mathsf{match}\ \alpha'\ \mathsf{with}\ \bar{\chi}]$. We have $\hat{\mathscr{C}}[\alpha' \divideontimes]$.
Either $\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = C_1\ \mathsf{in}\ C_2$ can progress with an instantiation constraint (in the above case) to discharge the match constraint or $\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = C_1\ \mathsf{in}\ C_2$ is stuck.

□

C.3 Termination

This section presents a proof of termination for our solver. Most rewrite rules, in both unification and constraint solving, are *destructive*—that is, they eliminate or modify the structure of a constraint in a way that prevents the rule from begin applied again. Consequently, to establish termination, it suffices to consider only those rules that are not inherently destructive.

Lemma C.15 (Unification termination). *The unifier terminates on all inputs.*

Proof. Let every shape ς have an integer *weight* defined by $\mathrm{sw}(\varsigma) \triangleq 4 + 2 \times |\varsigma|$, where $|\varsigma|$ is the arity of the shape ς. The weight of a type $\mathrm{tw}(\tau)$ is defined by:

$$\begin{array}{rcl} \mathrm{tw}(\alpha) & \triangleq & 1 \\ \mathrm{tw}(\varsigma\,\bar{\tau}) & \triangleq & \mathrm{iw}(\varsigma\,\bar{\tau}) - 2 \\ \mathrm{iw}(\alpha) & \triangleq & 0 \\ \mathrm{iw}(\varsigma\,\bar{\tau}) & \triangleq & \mathrm{sw}(\varsigma) + \mathrm{iw}(\bar{\tau}) \\ \mathrm{iw}(\bar{\tau}) & \triangleq & \sum_{i=1}^{n} \mathrm{iw}(\tau_i) \end{array}$$

The helper $\mathrm{iw}(\tau)$ computes the "internal" weight of τ; in the common case of shallow types it is just the weight of its head shape.

We define the weight of a multi-equation as the sum of the weights of its members. The weight of a unification problem $\mathrm{uw}(U)$ is defined as the sum of the weights of its multi-equations.

In $U \longrightarrow U'$, the rules U-Decomp and U-Name are not obviously destructive, as they may introduce new constraints that are structurally larger than the constraint being rewritten.

However, we show that this is not problematic: in both cases, the unification weight $\mathrm{uw}(U)$ strictly decreases. The remaining rules are obviously destructive and either maintain or decrease the unification weight.

Case

$$\frac{\varsigma\,\bar{\alpha} = \varsigma\,\bar{\beta} = \epsilon}{\varsigma\,\bar{\alpha} = \epsilon \land \bar{\alpha} = \bar{\beta}}\ \text{U-Decomp}$$

We have:

$$\begin{array}{lrcl} (+) & \mathrm{uw}(\varsigma\,\bar{\alpha} = \varsigma\,\bar{\beta} = \epsilon) & = & \mathrm{tw}(\varsigma\,\bar{\alpha}) + \mathrm{tw}(\varsigma\,\bar{\beta}) + \mathrm{tw}(\epsilon) \\ (-) & \mathrm{uw}(\varsigma\,\bar{\alpha} = \epsilon \land \bar{\alpha} = \bar{\beta}) & = & \mathrm{tw}(\varsigma\,\bar{\alpha}) + \mathrm{tw}(\epsilon) + \mathrm{tw}(\bar{\alpha}) + \mathrm{tw}(\bar{\beta}) \\ \hline & & = & \mathrm{tw}(\varsigma\,\bar{\beta}) - \mathrm{tw}(\bar{\alpha}) - \mathrm{tw}(\bar{\beta}) \\ & & = & (\mathrm{sw}(\varsigma) + 0 - 2) - 2|\varsigma| \\ & & = & (2 + 2|\varsigma|) - 2|\varsigma| \ = \ \mathbf{2} \end{array}$$

Hence $\mathrm{uw}(\varsigma\,\bar{\alpha} = \varsigma\,\bar{\beta} = \epsilon) > \mathrm{uw}(\varsigma\,\bar{\alpha} = \epsilon \land \bar{\alpha} = \bar{\beta})$.

Case

$$\frac{\varsigma\,(\bar{\tau}, \tau_i, \bar{\tau}') = \epsilon \qquad \alpha \# \bar{\tau}, \bar{\tau}', \epsilon \qquad \tau_i \notin \mathcal{V}}{\exists \alpha.\, \varsigma\,(\bar{\tau}, \alpha, \bar{\tau}') = \epsilon \land \alpha = \tau_i}\ \text{U-Name}$$

Given $\tau_i \notin \mathcal{V}$, by Theorem 3.2, $\tau_i = \varsigma'\,\bar{\tau''}$ for some shape ς' and types $\bar{\tau''}$. So we have:

$$\begin{array}{lrcl} (+) & \mathrm{uw}(\varsigma\,(\bar{\tau}, \tau_i, \bar{\tau}') = \epsilon) & = & \mathrm{sw}(\varsigma) + \mathrm{iw}(\bar{\tau}) + \mathrm{iw}(\tau_i) + \mathrm{iw}(\bar{\tau}') - 2 + \mathrm{uw}(\epsilon) \\ (-) & \mathrm{uw}(\exists \alpha.\, \alpha = \tau_i \land \varsigma\,(\bar{\tau}, \alpha, \bar{\tau}') = \epsilon) & = & \mathrm{sw}(\varsigma) + \mathrm{iw}(\bar{\tau}) + 0 + \mathrm{iw}(\bar{\tau}') - 2 + \mathrm{uw}(\epsilon) + 1 + \mathrm{tw}(\tau_i) \\ \hline & & = & \mathrm{iw}(\tau_i) - \mathrm{iw}(\alpha) - \mathrm{tw}(\tau_i) - 1 \\ & & = & \mathrm{iw}(\tau_i) - 0 - (\mathrm{iw}(\tau_i) - 2) - 1 \\ & & = & \mathbf{1} \end{array}$$

Hence $\mathrm{uw}(\varsigma\,(\bar{\tau}, \tau_i, \bar{\tau}') = \epsilon) > \mathrm{uw}(\exists \alpha.\, \varsigma\,(\bar{\tau}, \alpha, \bar{\tau}') = \epsilon \land \alpha = \tau_i)$.

□

Theorem 5.8 (Termination). *The constraint solver terminates on all inputs.*

Proof. The difficulty for termination comes from the "suspended match discharge" rule S-Match-Ctx which can make arbitrary sub-constraints appear in the non-suspended part of the constraint; and from the instantiation rules that copy/duplicate existing structure in another part of the constraint, increasing its total size.

As we argued before, the other rewrite rules are *destructive*, they strictly simplify the constraint towards a normal form and can only be applied finitely many times when taken together. The fragment without discharge rules and incremental instantiation is also extremely similar to the constraint language of Pottier and Rémy [2005], so their termination proof applies directly.

Discharge rules. The discharge rules strictly decrease the number of occurrences of suspended match constraint (if we also count nested suspended constraints), and no rewriting rule introduces new suspended match constraints. So these discharge rules can only be applied finitely many times. To prove termination of constraint solving, it thus suffices to prove that rewriting sequences that do not contain one of the discharge rules (those that occur in-between two discharge rules) are always finite.

Starting instantiations. By a similar argument, the number of non-incremental instantiations $x\ \tau$ decreases strictly on S-Let-AppR when an incremental instantiation starts, and is preserved by other non-discharge rules. The rule S-Let-AppR can thus only occur finitely many times in non-discharging sequences, and it suffices to prove that all rewriting sequences that are non-discharging and do not contain S-Let-AppR are finite.

Other instantiation rules. Among other instantiation rules, the rule of concern is S-Inst-Copy, which is not destructive: it introduces new instantiation constraints and structurally increases the size of the constraint.

Intuitively, S-Inst-Copy should not endanger termination because the amount of copying it can perform for a given instantiation is bounded by the size of the types in the constraint C it is copying from. (C could have cyclic equations with infinite unfoldings, but S-Inst-Copy forbids copying in that case.) The difficulty is that rewrites to C can be interleaved with instantiation rules, so that the equations that are being copied can grow strictly during instantiation.

To control this, we perform a structural induction: to prove that (let $x\ \alpha\ [\bar{\alpha}] = C_1$ in C_2) does not contain infinite non-discharging non-instance-starting rewrite rules, we can assume that the result holds for the strictly smaller constraint C_1, and then prove termination of the incremental instantiations of x in C_2. (The notion of structural size used here is preserved by non-discharging rewrite rules, as they do not affect the let-structure of the constraint.)

Assuming that C_1 has no infinite rewriting sequence, it suffices to prove that only finitely many rewrites in the rest of the constraint (namely C_2) can occur between each rewrite of C_1.

We define a weight that captures the contribution of types within C_1 to the partial instances in C_2:

$$\mathrm{tw}\ (\varsigma\ \bar{\tau}) \triangleq 2 \times \mathrm{sw}\ (\varsigma) + \textstyle\sum_{i=1}^{n} \mathrm{tw}\ (\tau_i)$$

$$\mathrm{tw}\ (\alpha) \triangleq \begin{cases} \sup \{\mathrm{tw}\ (\tau) : \alpha = \tau \in C_1\} & \text{if } C_1 \text{ is acyclic} \\ 0 & \text{otherwise} \end{cases}$$

The weight of a incremental instantiation cw $(i^x(\alpha) \rightsquigarrow \tau)$ is defined as the sum of tw (τ) and tw (α). The weight of other constraints is given using the measure uw defined in the the proof of Lemma C.15.

Case

$$\frac{\text{let } x\ \alpha\ [\bar{\alpha}] = C \text{ in } \mathscr{C}[i^x(\alpha') \rightsquigarrow \gamma] \qquad C = C' \wedge \alpha' = \varsigma\ \bar{\beta} = \epsilon \qquad \alpha' \in \alpha, \bar{\alpha} \qquad \neg\mathrm{cyclic}(C) \qquad \bar{\beta}' \# \alpha', \gamma, \bar{\beta} \qquad x \# \mathrm{bv}(\mathscr{C})}{\text{let } x\ \alpha\ [\bar{\alpha}] = C \text{ in } \mathscr{C}[\exists \bar{\beta}'.\, \gamma = \varsigma\ \bar{\beta}' \wedge i^x(\bar{\beta}) \rightsquigarrow \bar{\beta}']} \text{ S-Inst-Copy}$$

We aim to show that the weight of the rewritten constraint $\exists \bar{\beta}'.\, \gamma = \varsigma\ \bar{\beta}' \wedge i^x(\bar{\beta}) \rightsquigarrow \bar{\beta}'$ is strictly less than the original $i^x(\alpha') \rightsquigarrow \gamma$.

$$\begin{array}{rcl}
\mathsf{cw}\ (i^x(\alpha') \rightsquigarrow \gamma) & = & 1 + \mathsf{tw}\ (\alpha) \\
& \geq & 1 + 2 \times \mathsf{sw}\ (\varsigma) + \sum_{i=1}^{n} \mathsf{tw}\ (\beta_i) \\
\mathsf{cw}\ (\exists \bar{\beta}'. \gamma = \varsigma\, \bar{\beta}' \wedge i^x(\bar{\beta}) \rightsquigarrow \bar{\beta}') & = & 1 + \mathsf{sw}\ (\varsigma) + \sum_{i=1}^{n} \mathsf{tw}\ (\beta_i) + |\bar{\beta}'|
\end{array}$$

To ensure a strict decrease, it suffices to show that $\mathsf{sw}\ (\varsigma) > |\bar{\beta}'|$. Given that $|\bar{\beta}'| = |\varsigma|$, and by the definition of $\mathsf{sw}\ (\varsigma)$, this inequality holds. Therefore, the weight strictly decreases under S-Inst-Copy.

Thus the constraint solver terminates. □

C.4 Correctness

Lemma C.16. *Given non-simple* C *constraint. If every match constraint* $\mathscr{C}[\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}] = C$ *satisfies* $\mathscr{C}[\tau \not\chi]$*, then* C *is unsatisfiable.*

Proof. By contradiction, inverting on the canonical derivation of C. □

Lemma C.17 (Scope preservation). *For all* C_1, C_2*, if* $C_1 \longrightarrow C_2$*, then* $\mathsf{fv}(C_1) \supseteq \mathsf{fv}(C_2)$.

Proof. By induction on $C_1 \longrightarrow C_2$. □

Corollary C.18. *For the closed-term-variable constraint* C*,* C *is satisfiable if and only if* $C \longrightarrow^* \hat{C}$ *and* $\hat{C}$ *is a solved form equivalent to* C.

Proof. We show each direction individually:

Case $\Longrightarrow$.

By transfinite induction on the well-ordering of constraints whose existence is shown in Theorem 5.8.

We have C is satisfiable. By Theorem C.14, we have three cases:

Subcase C *is solved*. We have $C \longrightarrow^* C$ and $C \equiv C$ by reflexivity. So we are done.

Subcase C *is stuck*. Given C is a closed-term-variable constraint, it must be the case that either C is false *or* $\hat{\mathscr{C}}[\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}]$ and $\mathscr{C}[\tau \not\chi]$.

If C is false, this contradicts our assumption that C is satisfiable. Similarly, by Lemma C.16, if C is $\hat{\mathscr{C}}[\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}]$, then this also contradicts the satisfiability of C.

Subcase $C \longrightarrow C'$.

By Theorem 5.9, we have $C \equiv C'$, thus C' is satisfiable. Additionally, by Lemma C.17, we have $\mathsf{fv}(C') = \emptyset$. So by induction, we have $C' \longrightarrow^* \hat{C}$ and $\hat{C}$ is a solved form equivalent to C'. By transitivity of equivalence, we therefore have $\hat{C} \equiv C$, as required.

Case $\Longleftarrow$.

By induction on the rewriting $C \longrightarrow^* \hat{C}$.

Subcase

$$\frac{}{\hat{C} \longrightarrow^* \hat{C}}\ \textsc{Zero-Step}$$

We have $C = \hat{C}$ by inversion. All solved forms are satisfiable, thus C is satisfiable.

Subcase

$$\frac{C \longrightarrow C' \qquad C' \longrightarrow^* \hat{C}}{C \longrightarrow^* \hat{C}}\ \textsc{One-Step}$$

By induction, we have C' is satisfiable. By Theorem 5.9, $C \equiv C'$, hence C is satisfiable.

□

D Properties of OmniML

This section states and proves the two central metatheoretic properties of OmniML. The first is the *soundness and completeness* of the constraint generator $[\![e : \alpha]\!]$ with respect to the OmniML typing rules. The second is the existence of *principal types*, which follows as a consequence of soundness and completeness: every closed well-typed term e admits a most general type.

Throughout this section, we restrict our attention to *closed terms*. This is because the typing context Γ can contain bindings to terms whose type is "guessed". When we generate constraints for a term e under a context Γ, we encode the type schemes in Γ as part of the constraint itself using let-constraints. However, these schemes are treated as known within the constraint! As a result, we assume terms are closed from the outside to avoid Γ leaking any guessed type information.

D.1 Simple syntax-directed system

As a first step towards proving soundness and completeness of constraint generation, we first present a variant of the OmniML type system for *simple terms*. For this system, the syntax tree completely determines the derivation tree.

We use the standard technique of removing the Inst and Gen rules, and always apply instantiations in Var (Var-SD) and always generalize at let-bindings (Let-SD). We can show that this system is sound and complete with respect to the declarative rules.

Theorem D.1 (Soundness of the syntax directed rules). *Given the simple term e. If $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} e : \tau$ then we also have $\Gamma \vdash_{\mathsf{simple}} e : \tau$*

Proof. Induction on the given derivation. □

Theorem D.2 (Completeness of the syntax directed rules). *Given the simple term e. If $\Gamma \vdash_{\mathsf{simple}} e : \sigma$, then $\Gamma \vdash_{\mathsf{simple}} e : \tau$ for any instance τ of σ.*

Proof. Induction on the given derivation. □

Inversion. On a simple syntax-directed derivation $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} e : \tau$, we have the usual inversion principle:

Lemma D.3 (Simple inversion).

(i) *If $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} x : \tau$, then $x : \forall \bar{\alpha}. \tau' \in \Gamma$ and $\tau = \tau'[\bar{\alpha} := \bar{\tau}]$.*

(ii) *If $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} \lambda x. e : \tau$, then $\Gamma, x : \tau_1 \vdash^{\mathsf{sd}}_{\mathsf{simple}} e : \tau_2$ and $\tau = \tau_1 \to \tau_2$.*

(iii) *If $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} e_1\ e_2 : \tau$, then $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} e_1 : \tau' \to \tau$ and $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} e_2 : \tau'$.*

(iv) *If $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} () : \tau$, then $\tau = 1$.*

(v) *If $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau$, then $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} e_1 : \tau'$, $\bar{\alpha} \# \mathsf{fv}(\Gamma)$, and $\Gamma, x : \forall \bar{\alpha}. \tau' \vdash^{\mathsf{sd}}_{\mathsf{simple}} e_2 : \tau$.*

(vi) *If $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} (e : \exists \bar{\alpha}. \tau') : \tau$, then $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} e : \tau'[\bar{\alpha} := \bar{\tau}]$ and $\tau = \tau'[\bar{\alpha} := \bar{\tau}]$.*

(vii) *If $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} (e_1, \ldots, e_n) : \tau$, then $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} e_i : \tau_i$ for all $1 \leq i \leq n$ and $\tau = \Pi^n_{i=1} \tau_i$.*

(viii) *If $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} e.j^n : \tau$, then $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} e : \Pi^n_{i=1} \tau_i$ and $\tau = \tau_j$, with $n \geq j$.*

(ix) *If $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} [e : \exists \bar{\alpha}. \forall \bar{\beta}. \tau'] : \tau$, then $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} e : \tau[\bar{\alpha} := \bar{\tau}]$, $\bar{\beta} \# \Gamma$ and $\tau = [\forall \bar{\beta}. \tau'][\bar{\alpha} := \bar{\tau}]$.*

(x) *If $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} \langle e : \exists \bar{\alpha}. \sigma \rangle : \tau$, then $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} e : [\sigma][\bar{\alpha} := \bar{\tau}]$ and $\sigma[\bar{\alpha} := \bar{\tau}] \leq \tau$.*

(xi) *If $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} \square\ \mathsf{with}\ \bar{e} : \tau$, then $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} e_i : \tau'_i$ for all $1 \leq i \leq n$.*

(xii) *If $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} T.\{\overline{\ell = e}\} : \tau$, then $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} e_i : \tau_i$ and $T.\ell \leq \tau \to \tau_i$ for $1 \leq i \leq n$ and $\mathsf{dom}\ (T.\Omega) = \bar{\ell}$.*

(xiii) *If $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} \{\overline{\ell = e}\} : \tau$, then $\bar{\ell} \blacktriangleright T$ and $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} T.\{\overline{\ell = e}\} : \tau$.*

(xiv) *If* $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} e.T.\ell : \tau$, *then* $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} e : \tau', T.\ell \leq \tau' \to \tau$.

(xv) *If* $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} e.\ell : \tau$, *then* $\ell \triangleright T$ *and* $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} e.T.\ell : \tau$.

D.2 Canonicalization of typability

Our system satisfies a similar canonicalization theorem to constraint satisfiability.

Lemma D.4 (Composability of unicity).

(i) *If* $\mathscr{E}_1[\square \triangleleft \varsigma \mid \bar{e}]$, *then* $\mathscr{E}_2[\mathscr{E}_1][\square \triangleleft \varsigma \mid \bar{e}]$.

(ii) *If* $\mathscr{E}_1[e \triangleright \varsigma]$, *then* $\mathscr{E}_2[\mathscr{E}_1][e \triangleright \varsigma]$.

Proof. By induction on $\mathscr{E}_2$. □

Lemma D.5 (Decanonicalization). *If* $\Vdash e : \tau$, *then* $\emptyset \vdash e : \tau$.

Proof. By induction on the given derivation $\Vdash e : \tau$. □

Theorem D.6 (Canonicalization). *If* $\vdash e : \sigma$, *then* $\Vdash e : \tau$ *for any instance* τ *of* σ.

Proof. By induction on the following measure of e:

$$\|e\| \triangleq \langle \#\mathsf{implicit}\ e, |e| \rangle$$

where $\langle \ldots \rangle$ denotes a lexicographically ordered pair, and

(1) $\#\mathsf{implicit}\ e$ is the number of implicit constructs in e *i.e.*, overloaded tuple projections $e.j$, implicit non-unique field projections $e.\ell$, implicit non-unique records $\{\overline{\ell = e}\}$, polytype instantiations $\langle e \rangle$ and polytype boxing $[e]$.

(2) the last component $|e|$ is a structural measure of terms *i.e.*, a application $e_1\ e_2$ is larger than the two terms e_1, e_2.

This measure is analogous to the measure $\|C\|$ for constraints. □

D.3 Unifiers

A substitution ϑ is an idempotent function from type variables to types. The (finite) domain of ϑ is the set of type variables such that $\vartheta(\alpha) \neq \alpha$ for any $\alpha \in \mathsf{dom}\ \vartheta$, while the codomain consists of the free type variables of its range. We use the notation $[\bar{\alpha} := \bar{\tau}]$ for the substitution ϑ with domain $\bar{\alpha}$ and $\vartheta(\bar{\alpha}) = \bar{\tau}$.

The constraint induced by a substitution ϑ, written $\exists\vartheta$, is $\exists\bar{\beta}.\, \bar{\alpha} = \bar{\tau}$ where $\bar{\beta} = \mathsf{rng}\ \vartheta$, $\bar{\alpha} = \mathsf{dom}\ \vartheta$ and $\vartheta(\bar{\alpha}) = \bar{\tau}$.

Definition D.7 (Unifier). A substitution ϑ is a unifier of C if $\exists\vartheta$ entails C. A unifier ϑ of C is *most general* when $\exists\vartheta$ is equivalent to C.

Lemma D.8 (Simple inversion of unifiers).

- *If* ϑ *is a unifier of* $\tau_1 = \tau_2$, *then* $\vartheta(\tau_1) = \vartheta(\tau_2)$.
- *For simple* C_1, C_2, *if* ϑ *is a unifier of* $C_1 \wedge C_2$, *then* ϑ *is a unifier of* C_1 *and* C_2.
- *For simple* C, *if* ϑ *is a unifier of* $\exists\alpha.\, C$, *then* $\vartheta[\alpha := \tau]$ *is a unifier of* C *for some* τ.
- *For simple* C, *if* ϑ *is a unifier of* $\forall\alpha.\, C$, *then* ϑ *is a unifier of* C.

Proof. Follows by simple inversion. □

Lemma D.9. *If* ϑ *unifies* $\exists\alpha.\, C$, *then there exists a unifier* ϑ' *that extends* ϑ *with* α, *where* ϑ' *is most general unifier of* $\exists\vartheta \wedge C$.

Then $\lambda\alpha.\, C$ *is equivalent to* $\lambda\alpha.\, \sigma \leq \alpha$ *under* ϑ, *where* $\sigma = \forall\bar{\beta}.\, \vartheta'(\alpha)$ *and* $\bar{\beta} = \mathsf{fv}(\vartheta'(\alpha)) \setminus \mathsf{rng}\ \vartheta$. *We write this equivalent constraint abstraction as* $[\![\lambda\alpha.\, C]\!]_\vartheta$.

Proof. See Pottier and Rémy [2005]. □

Lemma D.10 (Let inversion of unifiers). *For simple* C_1, C_2. *If* ϑ *unifies* $\mathsf{let}\ x = \lambda\alpha.\, C_1\ \mathsf{in}\ C_2$, *then* ϑ *unifies* $\exists\alpha.\, C_1$ *and* ϑ *unifies* $\mathsf{let}\ x = [\![\lambda\alpha.\, C_1]\!]_\vartheta\ \mathsf{in}\ C_2$

Proof. Follows from Lemma D.9 and simple inversion. □

Lemma D.11. *For two substitutions* ϑ, ϑ'. *If* $\exists\vartheta \vDash \exists\vartheta'$, *there exists* ϑ'' *such that* $\vartheta = \vartheta'' \circ \vartheta'$.

Proof. Standard result, follows from definition of $\exists\vartheta$. □

D.4 Soundness and completeness of constraint generation

Lemma D.12. *For any term context* $\mathscr{E}$, *term* e, $[\![\mathscr{E}[\square : \tau] : \tau']\!][[\![e : \tau]\!]] = [\![\mathscr{E}[e] : \tau']\!]$.

Proof. By induction on the structure of $\mathscr{E}$. □

Lemma D.13. *For any term* e, $\lfloor [\![e : \tau]\!] \rfloor = [\![\lfloor e \rfloor : \tau]\!]$.

Proof. By induction on e. □

Lemma D.14 (Simple soundness and completeness). *For simple terms* e. $\vartheta(\Gamma) \vdash^{\mathsf{sd}}_{\mathsf{simple}} e : \vartheta(\tau)$ *if and only if* ϑ *is a unifier of* $[\![\Gamma \vdash e : \tau]\!]$.

Proof. By induction on e simple. □

Theorem D.15 (Soundness and completeness). $\Vdash e : \vartheta(\alpha)$ *if and only if* ϑ *is a unifier of* $[\![e : \alpha]\!]$

Proof. By induction on the number n of implicit terms in e.

Case n *is* 0.

			e simple	Premise
	$\emptyset \vdash^{\mathsf{sd}}_{\mathsf{simple}} e : \vartheta(\alpha)$	$\iff$	ϑ unifies $[\![e : \alpha]\!]$	Lemma D.14
	$\emptyset \vdash^{\mathsf{sd}}_{\mathsf{simple}} e : \vartheta(\alpha)$	$\iff$	$\Vdash e : \vartheta(\alpha)$	When e simple
☞	$\Vdash e : \vartheta(\alpha)$	$\iff$	ϑ unifies $[\![e : \alpha]\!]$	Above

Case n *is* $k + 1$.

Subcase $\implies$.

Subsubcase

$$\frac{\mathscr{E}[e \triangleright \nu\bar{\gamma}.\,\Pi_{i=1}^{n}\bar{\gamma}] \qquad \vartheta(\Gamma) \Vdash \mathscr{E}[e.j^n] : \vartheta(\alpha)}{\Vdash \mathscr{E}[e.j] : \vartheta(\alpha)}\ \textsc{Can-Proj-I}$$

$\vartheta(\Gamma) \Vdash \mathscr{E}[e.j^n] : \vartheta(\alpha)$	Premise
ϑ unifies $[\![\Gamma \vdash \mathscr{E}[e.j^n] : \alpha]\!]$	By *i.h.*
$[\![\Gamma \vdash \mathscr{E}[e.j^n] : \alpha]\!] = \text{let } \Gamma \text{ in } [\![\mathscr{E}[e.j^n] : \alpha]\!]$	By definition
$= \text{let } \Gamma \text{ in } [\![\mathscr{E}[\square : \beta] : \alpha]\!][[\![e.j^n : \beta]\!]]$	Lemma D.12
$[\![e.j^n : \beta]\!] \equiv \exists \alpha_1 \bar{\gamma}.\, [\![e : \alpha_1]\!] \wedge \alpha_1 = \Pi_{i=1}^{n}\bar{\gamma} \wedge \beta = \gamma_j$	By definition
$\equiv \exists \alpha_1.\, [\![e : \alpha_1]\!] \wedge \text{match } \alpha_1 := \nu\bar{\gamma}.\,\Pi_{i=1}^{n}\bar{\gamma} \text{ with } \Pi\ \gamma_j \Rightarrow \beta = \gamma$	″
ϑ unifies $\text{let } \Gamma \text{ in } [\![\mathscr{E}[\square : \beta] : \alpha]\!][\exists \alpha_1.\, [\![e : \alpha_1]\!] \wedge \ldots]$	Above
$\mathscr{E}[e \triangleright \nu\bar{\gamma}.\,\Pi_{i=1}^{n}\bar{\gamma}]$	Premise
Let $\mathscr{C} = \text{let } \Gamma \text{ in } [\![\mathscr{E}[\square : \beta] : \alpha]\!][\exists \alpha_1.\, [\![e : \alpha_1]\!] \wedge \square]$.	
$\phi \vdash \lfloor \mathscr{C}[\alpha_1 = \mathfrak{g}] \rfloor$	Premise
$\exists \alpha_1.\, [\![e : \alpha_1]\!] \wedge \alpha_1 = \mathfrak{g} = \exists \alpha_1.\, [\![(e : \mathfrak{g}) : \alpha_1]\!]$	By definition
$= [\![\square \text{ with } (e : \mathfrak{g}) : \beta]\!]$	″
$\lfloor \mathscr{C}[\alpha_1 = \mathfrak{g}] \rfloor = \lfloor \text{let } \Gamma \text{ in } [\![\mathscr{E}[\square : \beta] : \alpha]\!][[\![\square \text{ with } (e : \mathfrak{g}) : \beta]\!]] \rfloor$	″
$= \lfloor \text{let } \Gamma \text{ in } [\![\mathscr{E}[\square \text{ with } (e : \mathfrak{g})] : \alpha]\!] \rfloor$	Lemma D.12
$= \text{let } \Gamma \text{ in } \lfloor [\![\mathscr{E}[\square \text{ with } (e : \mathfrak{g})] : \alpha]\!] \rfloor$	By definition
$= \text{let } \Gamma \text{ in } [\![\lfloor \mathscr{E}[\square \text{ with } (e : \mathfrak{g})] \rfloor : \alpha]\!]$	Lemma D.13
ϕ unifies $\text{let } \Gamma \text{ in } [\![\lfloor \mathscr{E}[\square \text{ with } (e : \mathfrak{g})] \rfloor : \alpha]\!]$	Above
$\Vdash \lfloor \mathscr{E}[\square \text{ with } (e : \mathfrak{g})] \rfloor : \phi(\alpha)$	By *i.h.*
$\emptyset \vdash \lfloor \mathscr{E}[\square \text{ with } (e : \mathfrak{g})] \rfloor : \phi(\alpha)$	Lemma D.5

$\text{shape}(\mathfrak{g}) = \nu\bar{\gamma}.\,\Pi_{i=1}^{n}\bar{\gamma}$	$\Longrightarrow$ E
$\mathscr{C}[\alpha_1 \mathbin{!} \nu\bar{\gamma}.\,\Pi_{i=1}^{n}\bar{\gamma}]$	Above
ϑ unifies $\mathscr{C}[\text{match } \alpha_1 \text{ with } \Pi\ \gamma_j \Rightarrow \beta = \gamma]$	By Match-Ctx
$[\![e.j : \beta]\!] = \exists \alpha_1.\, [\![e : \alpha_1]\!] \wedge \text{match } \alpha_1 \text{ with } \ldots$	By definition
$\mathscr{C}[\text{match } \alpha_1 \text{ with } \ldots] = \text{let } \Gamma \text{ in } [\![\mathscr{E}[\square : \beta] : \alpha]\!][\exists \alpha_1.\, [\![e : \alpha_1]\!] \wedge \ldots]$	″
$= \text{let } \Gamma \text{ in } [\![\mathscr{E}[\square : \beta] : \alpha]\!][[\![e.j : \beta]\!]]$	Above
$= \text{let } \Gamma \text{ in } [\![\mathscr{E}[e.j] : \alpha]\!]$	Lemma D.12
$= [\![\mathscr{E}[e.j] : \alpha]\!]$	
☞ ϑ unifies $[\![\mathscr{E}[e.j] : \alpha]\!]$	

Subsubcase *Can-Poly-I, Can-Use-I, Can-Rcd-I, Can-Rcd-Proj-I.*

Similar arguments.

Subcase $\Longleftarrow$.

Subsubcase

$$\frac{\mathscr{C}[\alpha_1 \mathbin{!} \nu\bar{\gamma}.\,\Pi_{i=1}^{n}\bar{\gamma}] \qquad \vartheta \text{ unifies } \mathscr{C}[\text{match } \alpha_1 := \nu\bar{\gamma}.\,\Pi_{i=1}^{n}\bar{\gamma} \text{ with } \ldots]}{\vartheta \text{ unifies } \underbrace{\mathscr{C}[\text{match } \alpha_1 \text{ with } \Pi\ \gamma_j \Rightarrow \beta = \gamma]}_{[\![e:\alpha]\!]}}\ \textsc{Can-Match-Ctx}$$

$[\![e : \tau]\!] = \text{let } \Gamma \text{ in } [\![\mathscr{E}[e.j] : \alpha]\!]$ Premise

$\mathscr{C} = \mathsf{let}\ \Gamma\ \mathsf{in}\ [\![\mathscr{E}[\square : \beta] : \alpha]\!][\exists\alpha.\, [\![e : \alpha]\!] \wedge \square]$	Premise
ϑ unifies $\mathscr{C}[\mathsf{match}\ \alpha_1 := \nu\bar{\gamma}.\,\Pi_{i=1}^{n}\bar{\gamma}\ \mathsf{with}\ \ldots]$	Premise
ϑ unifies $[\![\mathscr{E}[e.j^n] : \alpha]\!]$	Above (See $\Longrightarrow$ direction)
$\Vdash \mathscr{E}[e.j^n] : \vartheta(\alpha)$	By *i.h.*
$\Gamma' \vdash \mathscr{E}[\square\ \mathsf{with}\ (e : \mathfrak{g})] : \tau'$	Premise
$\Gamma' = \emptyset$	$\mathscr{E}[\square\ \mathsf{with}\ (e : \mathfrak{g})]$ is closed
$\Vdash \mathscr{E}[\square\ \mathsf{with}\ (e : \mathfrak{g})] : \tau'$	Lemma D.5
$[\alpha := \tau']$ unifies $[\![\mathscr{E}[\square\ \mathsf{with}\ (e : \mathfrak{g})] : \alpha]\!]$	By *i.h.*
$\phi[\alpha := \phi(\tau')] \vdash [\![\mathscr{E}[\square\ \mathsf{with}\ (e : \mathfrak{g})] : \alpha]\!]$	By definition
$\mathscr{E}[\alpha_1\ !\ \nu\bar{\gamma}.\,\Pi_{i=1}^{n}\bar{\gamma}]$	Premise
$\mathsf{shape}(\mathfrak{g}) = \nu\bar{\gamma}.\,\Pi_{i=1}^{n}\bar{\gamma}$	$\Longrightarrow$ E
$\mathscr{E}[e \triangleright \nu\bar{\gamma}.\,\Pi_{i=1}^{n}\bar{\gamma}]$	Above
$\Vdash \mathscr{E}[e.j] : \vartheta(\alpha)$	By Can-Proj-I

Subsubcase $[e]$, $\langle e \rangle$, $\{\overline{\ell = e}\}$, $e.\ell$.

Similar arguments.

□

D.5 Principal types

Theorem 4.12 (Principal types). *For any well-typed closed OmniML term e, there exists a type τ such that: (i) $\vdash e : \tau$. (ii) For any other typing $\vdash e : \tau'$, then $\tau' = \theta(\tau)$ for some substitution θ.*

Proof. Let e be an arbitrary closed well-typed term; that is, there exists a type τ such that $\vdash e : \tau$. By Theorem D.15, the constraint $[\![e : \alpha]\!]$ is satisfiable (specifically under the unifier $\alpha = \tau$). By Corollary C.18, there exists a solved constraint $\hat{C}$ such that $\hat{C} \equiv [\![e : \alpha]\!]$. From $\hat{C}$, we extract a unifier ϑ. Since $\hat{C} \equiv \exists\vartheta$, it follows that ϑ is *most general*.

We claim that $\vartheta(\alpha)$ is the principal type of e. This amounts to showing:

(i) $\vdash e : \vartheta(\alpha)$

(ii) For any other typing $\vdash e : \tau'$, then $\tau' = \theta(\vartheta(\alpha))$ for some θ.

Since ϑ is a unifier of $[\![e : \alpha]\!]$, it follows immediately from Theorem D.15 that $\vdash e : \vartheta(\alpha)$, proving (i). For (ii), suppose $\vdash e : \tau'$ for some τ'. Then by Theorem D.15 again, there exists a unifier ϑ' of $[\![e : \alpha]\!]$ such that $\vartheta'(\alpha) = \tau'$. Since ϑ is most general, we have $\exists\vartheta' \vDash \exists\vartheta$, and by Lemma D.11, this implies the existence of a substitution ϑ'' such that $\vartheta' = \vartheta'' \circ \vartheta$. Hence, $\tau' = \vartheta'(\alpha) = \vartheta''(\vartheta(\alpha))$, witnessing that τ' is an instance of $\vartheta(\alpha)$, as required (ii). □

Contents