Omnidirectional type inference for ML: principality any way

ANONYMOUS AUTHOR(S)

We propose a new concept of *omnidirectional* type inference: the ability to resolve ML-style typing constraints in disorder. In contrast, all known existing implementations typically infer the types of let-bound expressions before typechecking their use sites. Omnidirectional type inference relies on two technical devices: *suspended match constraints*, which suspend the resolution of some constraints until the context has more information about a type variable; and *incremental instantiation*, which allow taking instances of a partially solved type scheme containing suspended constraints, with a mechanism to incrementally update instances as the scheme is refined.

The benefits of omnidirectional type inference are most apparent for advanced ML extensions that rely on optional type annotations, where principality is *fragile*. We illustrate these advantages with OCaml's static overloading of record labels and datatype constructors and semi-explicit first-class polymorphism. By contrast, extensions that integrate seamlessly into the traditional ML framework—such as row polymorphism—already enjoy *robust* principality and do not gain from omnidirectionality.

1 Introduction

The Damas-Hindley-Milner (HM) [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 int $\to$ int and bool $\to$ bool.

This property ensures predictable and efficient inference. Local typing decisions are always optimal, yielding most general types without guessing or backtracking. As a result, inference of subexpressions can proceed in any order, with one exception: let-bound expressions are typically inferred before their use. Still, well-typedness is preserved under common program transformations such as let-contraction and argument reordering.

Principality, however, is fragile. Many extensions of ML preserve it straightforwardly—for example, extensible records with row-polymorphism [Garrigue 1998; Ohori 1995; Rémy 1989; Rémy and Vouillon 1997; Wand 1989] or higher-kinded types [Jones 1995b], preserve principality straightforwardly. Others, such as GADTs [Garrigue and Rémy 2013; Schrijvers, Jones, Sulzmann and Vytiniotis 2009], higher-rank polymorphism [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 naive typing rules. . 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.

A concrete example arises in OCaml with impredicative higher-rank polymorphism via *polymorphic object methods* [Garrigue and Rémy 1999]:

```
let self x = x#f x in self (object method f z = z end)
```

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). When typing `self` in the example above, one could *guess* the higher-rank type of `x` to be either $\langle \texttt{f} : \alpha.\alpha \to \alpha \rangle$ or $\langle \texttt{f} : \alpha.\alpha \to \alpha \to \alpha \rangle$ or—neither of which is strictly more general than the other, violating principality.

To restore principality, inference algorithms require[1] a minimal amount of *known* type information; in this example the binding of x should be annotated with the higher-rank type $\langle \mathsf{f} : \alpha.\, \alpha \to \alpha \rangle$. Yet, specifying *known type information* declaratively is difficult. As a result, specifications are often twisted with some direct or indirect algorithmic flavor in order to preserve principality and completeness.

Moreover, these (more or less) ad-hoc restrictions commonly reject examples whose type could easily be guessed. For instance, OCaml accepts or rejects the following expression, depending on the position of the annotation (green indicates typechecking success and red indicates failure):

```
let self' (x : ⟨f : α. α → α⟩) = if true then x#f x else x                OCaml
let self' x = if true then x#f x else (x : ⟨f : α. α → α⟩)                OCaml
```

Each fragile construct admits a *robust* counterpart where the type annotation is mandatory—for instance, $(e \# m : \sigma)$ is the robust form of $e \# m$ for polymorphic method instantiation. While robust constructs do not break principality, they are significantly more cumbersome to use, as they always require explicit type annotations. Fragile forms relieve this burden, but can only be elaborated into their robust counterpart if sufficient type information is already available from the context.

The challenge lies in finding a specification for the propagation of contextual type information in a way that is sufficiently expressive, principled, and intuitive for programmers, while still admitting a complete and principal inference algorithm. 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 from explicit type annotations.

The two dominant approaches thus far are *bidirectional* type inference [Pierce and Turner 1998] and π-*directional* 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, like the one above, to be rejected.

We propose *omnidirectional* type inference, which 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 intend to eventually apply our work to OCaml, so the fragile features we target—static overloading, polymorphic methods, and others—are those present in OCaml. In particular, while the idea of suspending constraints is not new (see §7), we show how to combine suspended constraints and Hindley-Damas-Milner *local let-generalization*; it is indispensable in OCaml[2] but 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 two novel devices—*suspended match constraints* and *incremental instantiation*—and show how they are applied to these fragile features. The subsequent sections present technical contributions:

(§3) A novel constraint language for omnidirectional inference, equipped with a semantics for suspended constraints and a new declarative characterization of *known* type information.

[1] Otherwise, they treat f monomorphically and fail on this example.
[2] In contrast, Haskell only supports top-level let-generalization.

- (§4) 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, a 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. Here, we detail our treatment for the interaction of let-generalization with suspended constraints via *partial type schemes*.
- (§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.

Finally, §7 compares related work. Section §8 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 higher-rank polymorphism available within OCaml.

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 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 $\texttt{ex}_1$, the type of `r` is unconstrained, so disambiguation fails.[3] In $\texttt{ex}_2$, the annotation fixes the type of `r`, thus `r`'s type is *known* and resolves `r.x` unambiguously. In $\texttt{ex}_3$, the type of `r` can only be `point`: considering the second porjection first,

[3]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 two for r. To check all

$$
\begin{array}{rcl@{\qquad}r}
e & ::= & [e : \sigma] \mid \langle e \rangle \mid (e : \tau) \mid \ldots & \text{Terms} \\
\tau & ::= & [\sigma]^{\varepsilon} \mid \ldots & \text{Types} \\
\sigma & ::= & \tau \mid \forall \alpha.\, \sigma \mid \forall \varepsilon.\, \sigma & \text{Type schemes} \\
\varepsilon & & & \text{Annotation variables}
\end{array}
$$

$$
\textsc{PolyML-Poly}\ \frac{\Gamma \vdash e : \sigma_1 \qquad (\sigma_1 : \sigma : \sigma_2)}{\Gamma \vdash [e : \sigma] : [\sigma_2]^{\varepsilon}}
\qquad
\textsc{PolyML-Inst}\ \frac{\Gamma \vdash e : \forall \varepsilon.\, [\sigma]^{\varepsilon}}{\Gamma \vdash \langle e \rangle : \sigma}
\qquad
\textsc{PolyML-Annot}\ \frac{\Gamma \vdash e : \tau_1 \qquad (\tau_1 : \tau : \tau_2)}{\Gamma \vdash (e : \tau) : \tau_2}
$$

Fig. 1. Syntax and typing rules for polytypes from PolyML [Garrigue and Rémy 1999].

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).

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 (§2.1):

```
let getx 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. We do not treat this behavior as principal; accordingly, we provide no formalization of such "default" rules, though their implementation is discussed further in §8. This fallback mechanism highlights the directionality of OCaml inference. Once the compiler selects a type, it commits to it—even if that choice causes errors downstream. Consider:

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

Here, OCaml defaults to `gray_point` for `r` when typing `r.x`, 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 some System-F-like expressiveness to OCaml by allowing 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 boxed type scheme $[\sigma]^{\varepsilon}$ that can be explicitly unboxed at use sites. The purpose of the annotation variable ε will be explained shortly. Boxed polytypes are considered to be (mono)types, enabling impredicativity. We write $[e : \sigma]$ to box a term e with the scheme σ, and $\langle e \rangle$ to unbox a polytype, instantiating it.

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`$\rangle$.

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.

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 2013].

Semi-explicit first-class polymorphism. For the remainder of this work, we will focus on polytypes—also called *semi-explicit first-class polymorphism* [Garrigue and Rémy 1999]—as originally formulated in the PolyML calculus. Polytypes expose the tricky interaction with principality of interest. We now review the typing rules of PolyML collected in Figure 1.

Annotation variables record the origins of polytypes and may themselves be generalized, yielding type schemes such as $\forall \varepsilon.\ [\sigma]^{\varepsilon}$. When ε is generalized, the polytype is considered *known*, rather than still being inferred—this distinction is precisely the purpose of annotation variables.

The introduction form (PolyML-Poly) for polytypes is a boxing operator $[e : \sigma]$ with an explicit polytype annotation σ. The resulting expression has type $[\sigma_2]^{\varepsilon}$ where ε is an arbitrary (typically fresh) annotation variable and σ_2 is a *freshened copy* of σ *i.e.*, a variant of σ with only its annotation variables renamed (see PolyML-Annot below for details). 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 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 of a not-yet-known polytype, and therefore $\langle e \rangle$ is ill-typed. This restriction enforces principality, preventing instantiation on *guessed* polytypes.

For example, the expression $\lambda x.\ \langle x \rangle$ is not typable. Indeed, the λ-bound variable x is assigned a monotype. The only admissible type for x is $x : [\sigma]^{\varepsilon}$ for some σ and ε. Since ε is bound in the surrounding context at the point of typing $\langle x \rangle$, it cannot be generalized prior to unboxing, rendering the term ill-typed.

PolyML-Annot can be used to freshen annotation variables. The auxiliary relation $(\sigma_1 : \sigma : \sigma_2)$ (also used in PolyML-Poly) holds if there exists renamings η_1, η_2 on annotation variables (leaving ordinary type variables unchanged) such that $\sigma_1 = \eta_1(\sigma)$ and $\sigma_2 = \eta_2(\sigma)$. Intuitively, $(\sigma_1 : \sigma : \sigma_2)$ produces two *fresh copies* of σ, preventing unwanted sharing of annotation variables that could otherwise block generalization. We usually omit annotation variables in annotations, since we can implicitly introduce fresh ones in their place.

For example, $\lambda x : [\sigma].\ \langle x \rangle$, which is syntactic sugar for $\lambda x.\ \mathsf{let}\ x = (x : [\sigma])\ \mathsf{in}\ \langle x \rangle$, is well-typed because the explicit annotation introduces a fresh variable annotation ε_1, which can then be generalized, yielding $\forall \varepsilon_1.\ [\sigma]^{\varepsilon_1}$.

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 enforce a fixed order when typechecking let-bindings $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$: first typecheck the definition e_1, then the body e_2. π-directionality leverages this ordering to resolve overloaded or ambiguous constructs in a *principal* way: the (parts of) types with polymorphic annotation variables (*e.g.* $\forall \varepsilon.\ [\sigma]^{\varepsilon}$) are treated as *known* and may guide disambiguation, whereas the (parts of) types with monomorphic annotation variables are considered not-yet-known and cannot be relied on for disambiguation.

We call this π-directional (read as "**pi**-directional") type inference, to mean that **p**olymorphic expressions must be typed before their **i**nstances. π-directionality is subtle, but it aligns with the implicit inference order already present in most ML-like typecheckers, making it straightforward to retrofit into existing implementations. For OCaml, the mechanism for annotation variables even comes *for free*, as a byproduct of the extensive optimizations in its inference algorithm using levels.

This mechanism was originally proposed by Garrigue and Rémy [1999] for semi-explicit first-class polymorphism, and later used by Le Botlan and Rémy [2009] for empowering ML^{F}. It has since been adopted in OCaml for features such as polymorphic object methods and the overloading of record fields and variant constructors. More generally, OCaml uses π-directionality whenever the typechecker disambiguates on type information.

To illustrate π-directionality, consider:

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

At first glance, $\mathtt{ex}_5$ and $\mathtt{ex}_6$ appear equivalent: both simply instantiate the polytype bound to p. Yet, PolyML accepts $\mathtt{ex}_5$ and rejects $\mathtt{ex}_6$. This is because the **let**-binding in $\mathtt{ex}_5$ allows p to type scheme $\forall \varepsilon.\ [\forall \alpha.\, \alpha \to \alpha]^{\varepsilon}$, and thus its type is considered *known*–enabling unboxing (PolyML-Inst). In $\mathtt{ex}_6$, 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 PolyML's inference algorithm, consider two equivalent versions of $\mathtt{ex}_6$:[4]

```
let ex₆₂ = app (fun p → ⟨p⟩) pid            PolyML  OmniML
let ex₆₃ = rev_app pid (fun p → ⟨p⟩)        PolyML  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, PolyML infers all subexpressions *simultaneously*, until they are **let**-bound. Consequently, PolyML does not make any difference between $\mathtt{ex}_6$, $\mathtt{ex}_{62}$, and $\mathtt{ex}_{63}$.

Treating both examples uniformly is in one sense a strength of π-directionality, but it also reveals a limitation: annotatability is fragile, in that well-typedness depends on the *precise* placement of annotations, often forcing the programmer to introduce annotations that would otherwise be unnecessary. For instance, the following two terms differ only in the position of the annotation, yet only the one on the left-hand side is well-typed.

$$\lambda f.\, \langle (f : [\forall \alpha.\, \alpha \to \alpha]) \rangle\ f \qquad\qquad \lambda f.\, \langle f \rangle\ (f : [\forall \alpha.\, \alpha \to \alpha])$$

Bidirectional type inference. Bidirectional type inference 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 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 $\tau \to \tau'$, and then *checking* e_2 against τ (Syn-App); but the opposite, mode-correct

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

choice (Chk-App) is also possible:

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

Within the bidirectional framework, 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 ($\Leftarrow \tau$), or (3) produced by a synthesizing premise ($\Rightarrow \tau$). Using this discipline, we can recast polytypes and eliminate two artifacts needed in our π-directional presentation: explicit annotations on boxing and annotation variables. Since $(e : \tau)$ already propagates known information, $[e]$ requires no attached annotation; and because "knownness" now follows from inference modes rather than polymorphism, annotation variables are unnecessary. The resulting syntax and typing rules are as follows:

$$\begin{array}{lcll} e & ::= & [e] \mid \langle e \rangle \mid (e : \tau) \mid \ldots & \text{Terms} \\ \tau & ::= & [\sigma] \mid \ldots & \text{Types} \\ \sigma & ::= & \tau \mid \forall \alpha.\, \sigma & \text{Type schemes} \end{array} \qquad \textsc{Chk-Poly}\ \frac{\Gamma \vdash e \Leftarrow \sigma}{\Gamma \vdash [e] \Leftarrow [\sigma]} \qquad \textsc{Syn-Inst}\ \frac{\Gamma \vdash e \Rightarrow [\sigma]}{\Gamma \vdash \langle e \rangle \Rightarrow \sigma} \qquad \textsc{Syn-Annot}\ \frac{\Gamma \vdash e \Leftarrow \tau}{\Gamma \vdash (e : \tau) \Rightarrow \tau}$$

However, there is usually 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 often exist, but only with respect to a specification that made non-principal choices to begin with. For instance, $\mathtt{ex}_6$ would be ill-typed using the above rules for polytypes (with Syn-App).

Recent work on *contextual typing* [Xue and d. S. Oliveira 2024] addresses this difficulty by deferring the commitment between Syn-App and Chk-App. Typing multiple arguments from right to left (Chk-App) when sufficient contextual information is available, and thus successfully typechecks $\mathtt{ex}_6$. Nevertheless, it still enforces a fixed order of propagation, so some well-typed programs are rejected as ill-typed (*e.g.* $\mathtt{ex}_{62}$, $\mathtt{ex}_{63}$).

Limitations of directional type inference. Bidirectional type inference 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, relying on polymorphism—the essence of ML. 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 = let g (f : point → int) = f one in g (fun r → r.x)        OCaml  OmniML
```

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

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. As a result, we are able to propagate more type information and type more programs due to the limitations of (fixed) directionality discussed in §2.3.

The mechanism for suspension in our framework is our novel *suspended match constraints*. A match constraint (match τ with $\overline{\rho \to C}$) pairs a (typically unknown) matchee type τ with a finite series of shape-pattern branches $\overline{\rho \to C}$. These 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 top-level constructors (*e.g.* a nominal record type $\mathsf{t}\ \bar{\tau}$) and shape patterns ρ as type 'destructors' that may bind type information (*e.g.* the name of a nominal record type t) to meta-variables used in $\bar{C}$. This will be made precise in §3.1.

Suspended constraints in action. 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 $[\![e : \alpha]\!]$, which, given a term e and expected type α, produces a constraint C which is satisfiable if and only if e has the type α.

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 records, in the case of an ambiguous record projection $e.\ell$, we generate the typing constraint:

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

This constraint suspends resolution of the return type α until the record type β of e is *known*, say some type τ_0. The branch then matches τ_0 against the nominal type pattern $t\ \_$, binding the type constructor t to the pattern variable t when $\tau_0 = \mathsf{t}\ \bar{\tau}$, and failing otherwise. Using this, the globally unique projection type (scheme) $\forall \bar{\alpha}.\ \mathsf{t}\ \bar{\alpha} \to \tau$ for the qualified label $\mathsf{t}.\ell$ is instantiated, *e.g.* to $\mathsf{t}\ \bar{\tau}' \to \tau[\bar{\alpha} := \bar{\tau}']$ for some $\bar{\tau}'$. Finally, the resulting constraints are imposed on the domain (*i.e.*, $\tau_0 = \mathsf{t}\ \bar{\tau}'$) and codomain of the instantiated projection type (*i.e.*, $\alpha = \tau[\bar{\alpha} := \bar{\tau}']$).

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:

$$[\![\langle e \rangle : \alpha]\!] \triangleq \exists \beta.\ [\![e : \beta]\!] \wedge \mathsf{match}\ \beta\ \mathsf{with}\ ([s] \to s \leq \alpha)$$

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

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 $\mathcal{J}$, one of 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.[5] However, this state of affairs is not a necessity.

To efficiently achieve omnidirectional type inference for fragile ML extensions we work with *partial types schemes*, *i.e.*, with the ability to *incrementally instantiate* type schemes. That is, to instantiate type schemes that are not yet fully determined 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.

Plan. These two technical devices are introduced once and for all—in a general framework of constraint-based type inference. Each fragile ML construct can then be implemented by suspended constraints that expand to its robust counterpart once the annotation has been inferred. This generality comes at a cost, which is that everything is hard:

- (§3) Giving an adequate semantics for suspended constraints is hard, as we must capture declaratively the intuition that some type information must be *known* rather than *guessed*.
- (§5) Implementing incremental instantiation efficiently is equally hard, as it requires triggering re-instantiations upon refinements to the scheme, while avoiding redundant constraint solving across instantiations.

In return, the techniques we developed for the semantics also help provide declarative typing rules (§4) for each fragile construct, for which the generated constraints are sound and complete.

3 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 soundness, completeness, and principality proofs of inference. Without it, the meta-theory of our approach cannot be stated precisely.

Notation for collections. We write $\overline{X}$ for a (possible empty) set of elements $\{X_1, ..., X_n\}$ and a (possibly empty) sequence $X_1, ..., X_n$. 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 types[6] $\mathsf{t}\ \bar{\tau}$, and polytypes $[\sigma]$. 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. We write $\mathcal{V}$ for the set of type variables. We write $\bar{\alpha} \# \tau$ as a short-hand for $\bar{\alpha} \# \mathrm{fv}(\tau)$, where $\mathrm{fv}(-)$ computes the set of free variables of a given type.

Constraints. Building atop the constraint-based type inference framework of Pottier and Rémy [2005], we adopt a constraint language (Figure 2) that includes both term and type variables. Its semantics is given by a satisfiability judgment $\phi \vdash C$ (Figure 2). 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 $\mathfrak{G} \subseteq \mathcal{G}$ (the instances of a type scheme bound to x). We write $\phi[\alpha := \mathfrak{g}]$ and $\phi[x := \mathfrak{G}]$ for the extension of ϕ with a new binding. For a type τ, we write $\phi(\tau)$ for the ground type obtained by substitution.

[5] See §7 for a closer comparison with [Pottier and Rémy 2005].

[6] Type constructors are prefixed, except in OCaml code, where they are postfixed.

$$
\begin{array}{rcll}
\alpha, \beta, \gamma & \in & \mathcal{V} & \text{Type variables} \\
\tau & ::= & \alpha \mid 1 \mid \tau_1 \to \tau_2 \mid \Pi_{i=1}^{n} \tau_i \mid \mathsf{t}\ \bar{\tau} \mid [\sigma] & \text{Types} \\
\sigma & ::= & \tau \mid \forall \alpha.\, \sigma & \text{Type schemes} \\
C & ::= & \mathsf{true} \mid \mathsf{false} \mid C_1 \wedge C_2 \mid \tau_1 = \tau_2 \mid \exists \alpha.\, C \mid \forall \alpha.\, C & \text{Constraints} \\
 & \mid & \mathsf{let}\ x = \lambda \alpha.\, C_1\ \mathsf{in}\ C_2 \mid x\ \tau \mid \mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi} & \\
\chi & ::= & \rho \to C & \text{Branches} \\
\rho & ::= & \_ \mid t\ \_ \mid [s] & \text{Patterns} \\
\zeta & ::= & \nu \bar{\gamma}.\, \tau & \text{Shapes} \\
\varsigma & & & \text{Canonical principal shapes} \\
\mathscr{C} & ::= & \square \mid \mathscr{C} \wedge C \mid C \wedge \mathscr{C} \mid \exists \alpha.\, \mathscr{C} \mid \forall \alpha.\, \mathscr{C} & \text{Constraint contexts} \\
 & \mid & \mathsf{let}\ x = \lambda \alpha.\, \mathscr{C}\ \mathsf{in}\ C \mid \mathsf{let}\ x = \lambda \alpha.\, C\ \mathsf{in}\ \mathscr{C} & \\
\phi & ::= & \emptyset \mid \phi[\alpha := \mathfrak{g}] \mid \phi[x := \mathfrak{G}] & \text{Semantic environments} \\
\mathfrak{g} & \in & \mathcal{G} & \text{Ground types} \\
\mathfrak{G} & \subseteq & \mathcal{G} & \text{Sets of ground types}
\end{array}
$$

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

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

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

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

$$
\begin{array}{rcl}
\_\ \mathsf{matches}\ \varsigma\ \bar{\alpha} & \triangleq & \emptyset \\
t\ \_\ \mathsf{matches}\ (\nu \bar{\gamma}.\, \mathsf{t}\ \bar{\gamma})\ \bar{\alpha} & \triangleq & [t := \mathsf{t}] \\
[s]\ \mathsf{matches}\ (\nu \bar{\gamma}.\, [\sigma])\ \bar{\alpha} & \triangleq & [s := \sigma[\bar{\gamma} := \bar{\alpha}]]
\end{array}
$$

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

Fig. 2. Selected syntax and semantics of constraints.

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 syntactic sugar for the instantiation constraint $\exists \bar{\alpha}.\, \tau' = \tau$. OmniML-specific constraints, such as record label instantiation $t.\ell \leq \tau_1 \to \tau_2$, are defined using this instantiation form.

Two constructs deal with the introduction and elimination of constraint abstractions. Intuitively, a constraint abstraction $\lambda \alpha.\, C$ is a function which when applied to some type τ returns $C[\alpha := \tau]$. Abstractions are introduced by a let construct ($\mathsf{let}\ x = \lambda \alpha.\, C_1\ \mathsf{in}\ C_2$), which binds the constraint abstraction $\lambda \alpha.\, C_1$ to the term variable x in C_2; semantically, the abstraction $\lambda \alpha.\, C$ is interpreted as a set of ground types that satisfies C, and we require that this set be non-empty *i.e.*, there is at least one instantiation of α that satisfies C. Applications $x\ \tau$ eliminate abstractions, applying the 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. Concretely, if $\phi(x) = \phi'(\lambda\alpha.\,C)$, where ϕ' is the environment at the binding site of x, then $\phi(\tau) \in \phi(x)$ holds iff $\phi'[\alpha := \phi(\tau)] \vdash C$, which corresponds to the intuition that the application $(\lambda\alpha.\,C)\ \tau$ should be equivalent to $C[\alpha := \tau]$.

Finally, we introduce *suspended match constraints* (match τ with $\bar{\chi}$), which consist of: (1) A matchee τ. The constraint remains suspended while τ is a type variable, that is, until the *shape* of τ is determined. (2) A list of branches $\bar{\chi}$ of the form $\rho \to C$, where ρ is a shape pattern. For example, the pattern t _ matches record types, binding its name (*e.g.* the type constructor t) to pattern variable t. The constraint C is then solved in the extended context. 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, so we defer a full explanation until we define *shapes*.

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 is simply a constraint with a *hole*, analogous to evaluation contexts $\mathscr{E}$ used extensively in operational semantics. 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 $\mathrm{bv}(\mathscr{C})$ for the set of variables bound at the hole in $\mathscr{C}$.

3.1 Shapes

We introduce *shapes* as a generalization of type constructors for suspended match constraints. 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 up to α-conversion. When τ is a ground type, we omit the binder and write simply τ. We write $\bot$ for the shape $\nu\gamma.\,\gamma$, which we call the *trivial* shape. We write $\mathcal{S}$ 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 write that the pair $(\zeta, \bar{\tau})$ is a decomposition of τ.

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

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 non-trivial 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. We write ς for canonical principal shapes. Each non-variable type τ has a unique canonical principal shape, which we write $\mathsf{shape}(\tau)$. For example, $\mathsf{shape}(\mathsf{t}\ \bar{\tau})$ is $(\nu\bar{\gamma}.\,\mathsf{t}\ \bar{\gamma})$.

Polytypes are particularly interesting in this setting because they can be decomposed into shapes and treated analogously to type constructors. For instance, the polytype $[\forall\alpha.\,([\forall\beta.\,(\beta \to \mathsf{int\ list}) * \beta]) \to \alpha \to \alpha]$ has the principal shape $\varsigma = \nu\gamma.\,[\forall\alpha.\,([\forall\beta.\,(\beta \to \gamma) * \beta]) \to \alpha \to \alpha]$. The original polytype can thus be represented as the shape application $\varsigma\ (\mathsf{int\ list})$.

3.2 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. Our semantics is declarative but, unlike the rules of Pottier and Rémy [2005], it is not syntax-directed. This departure complicates reasoning and proofs. On the upside, our semantics directly suggest declarative typing rules for the surface language (§4).

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

Definition 3.3 (Discharged match constraint). Given a suspended constraint $(\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi})$ and a canonical shape ς, we introduce the syntactic sugar $(\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi})$ for the *discharged match constraint* that selects the branch in $\bar{\chi}$ that matches ς:

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

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 $\rho_i \to C_i$ is uniquely determined; but it may not exist as branches need not be exhaustive, in which case the discharged constraint is false.

The partial function $(\rho\ \text{matches}\ \varsigma\ \bar{\gamma})$ is defined in Figure 2: it matches a pattern ρ against a canonical principal shape ς opened with fresh shape variables $\bar{\gamma}$ (of the same arity as ς), which either fails or returns a substitution θ from pattern variables (*e.g.* nominal type variables t) to shape components (*e.g.* nominal type names t), which may themselves mention $\bar{\gamma}$. For example, the wildcard pattern _ matches any shape, yielding the empty substitution.

A natural attempt. To provide semantics for our suspended constraints, a first idea is to propose the following rule—henceforth referred to as the *natural semantics* of suspended constraints.

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

This rule states that a suspended constraint holds whenever the corresponding discharged constraint holds for the canonical shape ς of $\phi(\tau)$ in the semantic environment ϕ. Although simple and declarative, this semantics is too permissive. For example, $\exists\alpha.\,\mathsf{match}\ \alpha\ \mathsf{with}\ \_ \to \alpha = \mathsf{int}$ is satisfiable under the natural semantics:

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

The natural semantics can *guess* a type of α (*e.g.* int) in order to discharge the match constraint, rather than requiring α's type to be *known* from the surrounding context. This *ex nihilo* ("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 guessing or backtracking—cannot be complete with respect to this semantics; (2) this breaks the existence of principal solutions. Consider the function `fun r → r.x`, which projects the field `x` from the record `r`. The natural semantics lets us guess any record type containing the field `x` for r (*e.g.* `point`, `gray_point`). As a result, `r` has no most general type.

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 2), is the only non-syntax-directed rule in our semantics.

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

In this rule, a suspended match 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 $\mathscr{C}[\tau \,!\, \varsigma]$ (defined below) ensures that ς is uniquely determined by the context $\mathscr{C}$, capturing precisely what is means for the shape of a type to be *known*.

Definition 3.4 (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 3.5 (Simple constraints). We say that C is *simple* if it contains no suspended match constraints. We write $\phi \vdash_{\mathsf{simple}} C$ for a derivation of $\phi \vdash C$ that uses the rules listed in Figure 2, without using Match-Ctx. This judgment coincides with $\phi \vdash C$ on simple constraints.

Definition 3.6 (Unicity). We define the unicity condition $\mathscr{C}[\tau \,!\, \varsigma]$, which states that τ has a unique canonical shape ς within the context $\mathscr{C}$ as: $\forall \phi, \mathfrak{g}.\ \phi \vdash_{\mathsf{simple}} \lfloor \mathscr{C}[\tau = \mathfrak{g}] \rfloor \implies \mathsf{shape}(\mathfrak{g}) = \varsigma$.

The use of erasure $\lfloor \mathscr{C}[\tau = \mathfrak{g}] \rfloor$ in the definition of $\mathscr{C}[\tau \,!\, \varsigma]$ ensures that the unicity of ς is determined only by the constraints that have already been discharged in $\mathscr{C}$; it excludes suspended match constraints, which may be discharged in the future. This induces a partial order among the suspended match constraints within a constraint, corresponding to the order in which a solver may discharge them: a match constraint may only be discharged once all of its dependencies have been discharged.

The erasure of $\lfloor \mathscr{C}[\tau = \mathfrak{g}] \rfloor$ is a simple constraint, so the use of $\vdash_{\mathsf{simple}}$ avoids well-foundedness issues that would arise from a negative occurrence of $(\vdash)$ in a premise of Match-Ctx. Note that, when τ is not a variable, then $\square[\tau \,!\, \varsigma]$ holds trivially for $\varsigma = \mathsf{shape}(\tau)$. Likewise, when $\mathscr{C}$ is unsatisfiable, then $\mathscr{C}[\alpha \,!\, \varsigma]$ holds vacuously for any ς. The nontrivial cases arise when τ is a type variable and $\mathscr{C}$ is satisfiable.

Example 3.7. 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 with no contextual information: $\mathscr{C} := \exists \alpha. \square$. So for any ground type $\mathfrak{g}$, $\mathscr{C}[\alpha = \mathfrak{g}]$ is satisfiable, allowing $\mathfrak{g}$ to have an arbitrary shape (*e.g.* int, bool, *etc.*). As a result, the uniqueness condition $\mathscr{C}[\alpha \,!\, \varsigma]$ never holds making Match-Ctx inapplicable. The constraint is unsatisfiable as intended.

Example 3.8. Consider the satisfiable constraint:

$$\exists\alpha.\, \alpha = \mathsf{int} \wedge \mathsf{match}\ \alpha\ \mathsf{with}\ \_ \to \mathsf{true}$$

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

Example 3.9. Consider the more intricate example:

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

Suppose we attempt to apply Match-Ctx to the match on β first. We want to show $\mathscr{C}[\beta\,!\,\mathsf{bool}]$ for the context $\mathscr{C}$ equal to $(\mathsf{match}\ \alpha\ \mathsf{with}\ \_ \to \beta = \mathsf{bool}) \wedge \square \wedge \alpha = \mathsf{int}$. Its erasure $\lfloor\mathscr{C}\rfloor$ is $\mathsf{true} \wedge \square \wedge \alpha = \mathsf{int}$, which imposes no constraints on β. Thus both $\lfloor\mathscr{C}[\beta = \mathsf{int}]\rfloor$ and $\lfloor\mathscr{C}[\beta = \mathsf{bool}]\rfloor$ are 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 $\square \wedge (\mathsf{match}\ \beta\ \mathsf{with}\ \_ \to \mathsf{true}) \wedge \alpha = \mathsf{int}$. Its erasure $\lfloor\mathscr{C}\rfloor$ equal to $\square \wedge \mathsf{true} \wedge \alpha = \mathsf{int}$ does constraint α, giving $\mathscr{C}[\alpha\,!\,\mathsf{int}]$. We may therefore discharge the match on α, rewriting it as $(\mathsf{match}\ \alpha := \mathsf{int}\ \mathsf{with}\ \_ \to \beta = \mathsf{bool})$ *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.*, $\alpha = \mathsf{int} \wedge \beta = \mathsf{bool} \wedge (\mathsf{match}\ \beta\ \mathsf{with}\ \_ \to \mathsf{true}) \wedge \alpha = \mathsf{int}$. 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 match constraints must be resolved in a dependency-respecting order: attempting to resolve a match constraint too early may result in unsatisfiability.

Example 3.10. Let us consider a constraint with a cyclic dependency between match constraints:

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

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 $\mathscr{C} := \square \wedge \mathsf{match}\ \beta\ \mathsf{with}\ \_ \to \alpha = \mathsf{int}$. 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 The OmniML calculus

To prove correctness of constraint generation, we must define a surface language and its type system. Surprisingly, identifying an appropriate declarative type system to use as a specification is itself an interesting problem! In particular, naive specifications for fragile features often fail to preserve principality.

Consider polytypes. We can ask the user to provide a type scheme σ when unboxing, via an annotated syntax $\langle e : \exists\bar{\alpha}.\, \sigma\rangle$, which has a simple typing rule (Use-X). On the other hand, the natural typing rule for the fragile unboxing construct $\langle e\rangle$ breaks principality (Use-I-Nat). For example, term $\lambda x.\, \langle x\rangle\ x$ admits infinitely many typings for x, as explained in §1. This is precisely the difficulty also faced in the natural semantics of suspended constraints: σ must be *known*, not *guessed*. Our solution is the same in both cases, impose a unicity condition and introduce a contextual rule (Use-I) that transforms the fragile, implicit construct into the robust, explicit counterpart $\langle e : \exists\bar{\alpha}.\, \sigma\rangle$.

Use-I-Nat
$$\frac{\Gamma \vdash e : [\sigma]}{\Gamma \vdash \langle e\rangle : \sigma}$$

$$
\begin{array}{rcll}
e & ::= & x \mid () \mid \lambda x.\, e \mid e_1\ e_2 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid (e : \exists \bar{\alpha}.\, \tau) & \text{Terms} \\
& \mid & [e] \mid [e : \exists \bar{\alpha}.\, \sigma] \mid \langle e \rangle \mid \langle e : \exists \bar{\alpha}.\, \sigma \rangle & \\
& \mid & \{\overline{\ell = e}\} \mid e.\ell \mid \mathsf{t}.\{\overline{\ell = e}\} \mid e.\mathsf{t}.\ell & \\
\Gamma & ::= & \emptyset \mid \Gamma, x : \sigma & \text{Contexts} \\
\Omega & ::= & \emptyset \mid \Omega, \ell : \forall \bar{\alpha}.\, \mathsf{t}\ \bar{\alpha} \to \tau & \text{Label contexts}
\end{array}
$$

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

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

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

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

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

Fig. 3. Syntax and explicit, robust typing rules of OmniML.

4.1 Syntax

OmniML (Figure 3) extends ML with two fragile constructs: polytypes and nominal records. Its terms 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 $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$, and the following extensions: (1) 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. (2) 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: $\mathsf{t}.\{\ell_1 = e_1; \ldots; \ell_n = e_n\}$ and $e.\mathsf{t}.\ell$, where the nominal type annotation t indicates that the labels correspond to the label definitions in t (thereby disambiguating overloading). Variant constructors are not treated formally in OmniML, but behave analogously in practice.

We write e^i for fragile, implicit forms and e^x for their explicit counterparts. Typing rules for explicit terms are mostly standard; nominal records are 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 §4.3.

4.2 Typing rules for robust, explicit constructs

As usual, the main typing judgment $\Gamma \vdash e : \sigma$ (Figure 3) states that in context Γ, expression e has 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 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 $[\alpha := \mathsf{int}]$.

Explicit polytypes. Rule 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 $\forall \bar{\alpha}.\, \mathsf{t}\ \bar{\alpha} \to \tau$. A label ℓ may belong to multiple record types, but is unique within each type t. For a given t, we write $\Omega(\mathsf{t})$ for the restriction of Ω to its labels: $\Omega(\mathsf{t}) \triangleq \{\ell : \forall \bar{\alpha}.\, \mathsf{t}\ \bar{\alpha} \to \tau \in \Omega\}$. Thus, $\mathrm{dom}\ (\Omega(\mathsf{t}))$ is the set of labels belonging to t, and $(\Omega(\mathsf{t}))(\ell)$, abbreviated as $\Omega(\mathsf{t}.\ell)$, is the unique scheme $\forall \bar{\alpha}.\, \mathsf{t}\ \bar{\alpha} \to \tau$ associated with ℓ in t (if defined).

Label instantiations are typed by an auxiliary judgment $\mathsf{t}.\ell \leq \tau_1 \to \tau_2$ (Lab-Inst). Explicit field projections (Rcd-Proj-X) require that $e.\mathsf{t}.\ell$ projects from a record e of type τ_1 to τ_2, provided that $\tau_1 \to \tau_2$ is an instance of the projection scheme $\Omega(\mathsf{t}.\ell)$. Explicit records (Rcd-X) are typed similarly, checking that each field has the appropriate type. In addition, the premise asserts $\mathrm{dom}\ (\Omega(\mathsf{t})) = \bar{\ell}$ so that the declared fields exactly match the labels of t.

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 $\{\ell_1 = e_1; \ldots; \ell_n = e_n\}$, if the set of labels $\ell_1, \ldots, \ell_n$ uniquely identifies a record type t in the context Ω, then the record has the type of $\mathsf{t}.\{\ell_1 = e_1; \ldots; \ell_n = e_n\}$ (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 $e.\mathsf{t}.\ell$ (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 $\{x, y\}$ in ex_8 uniquely identifies `point`, even though the individual labels x and y also appear in `cpoint`. We formalize this *closed world uniqueness* using the predicates $\ell \triangleright \mathsf{t}$ (a label uniquely identifies t) and $\bar{\ell} \blacktriangleright \mathsf{t}$ (a closed label set uniquely identifies t):

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

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.

4.3 Typing rules for fragile, implicit constructs

We now turn to the typing of fragile implicit constructs (Figure 4). As with the satisfiability of suspended constraints, their typing rules rely on contextual information and are inherently non-compositional. All rules instantiate a common framework—the *omnidirectional recipe*—which

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

$$\textsc{Magic}\ \frac{(\Gamma \vdash e_i : \tau_i)_{i=1}^{n}}{\Gamma \vdash \{\bar{e}\} : \tau'} \qquad \textsc{Proj-I}\ \frac{\mathscr{E}[e \triangleright \nu\bar{\gamma}.\,\Pi_{i=1}^{n}\bar{\gamma}] \quad \Gamma \vdash \mathscr{E}[e.j/n] : \tau}{\Gamma \vdash \mathscr{E}[e.j] : \tau} \qquad \textsc{Use-I}\ \frac{\mathscr{E}[e \triangleright \nu\bar{\gamma}.\,[\sigma]] \quad \Gamma \vdash \mathscr{E}[\langle e : \exists\bar{\gamma}.\,\sigma\rangle] : \tau}{\Gamma \vdash \mathscr{E}[\langle e\rangle] : \tau} \qquad \textsc{Poly-I}\ \frac{\mathscr{E}[\square \triangleleft \nu\bar{\gamma}.\,[\sigma] \mid e] \quad \Gamma \vdash \mathscr{E}[[e : \exists\bar{\gamma}.\,\sigma]] : \tau}{\Gamma \vdash \mathscr{E}[[e]] : \tau}$$

$$\textsc{Rcd-I}\ \frac{\mathscr{E}[\square \triangleleft \nu\bar{\gamma}.\,\mathrm{t}\ \bar{\gamma} \mid \bar{e}] \qquad \Gamma \vdash \mathscr{E}[\mathrm{t}.\{\overline{\ell = e}\}] : \tau}{\Gamma \vdash \mathscr{E}[\{\overline{\ell = e}\}] : \tau} \qquad \textsc{Rcd-Proj-I}\ \frac{\mathscr{E}[e \triangleright \nu\bar{\gamma}.\,\mathrm{t}\ \bar{\gamma}] \qquad \Gamma \vdash \mathscr{E}[e.\mathrm{t}.\ell] : \tau}{\Gamma \vdash \mathscr{E}[e.\ell] : \tau}$$

Fig. 4. Typing rules for fragile, implicitly typed extensions.

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 $\{\overline{\ell = e}\}$). These predicates are analogous to the unicity condition $\mathscr{C}[\tau \,!\, \varsigma]$ for constraints, though the analogy is not exact. Our framework employs two variants depending on whether they infer a unique shape for a particular subexpression $\mathscr{E}[e \triangleright \varsigma \mid \bar{e}]$ or for the expected type of the context's hole $\mathscr{E}[\square \triangleleft \varsigma \mid \bar{e}]$.

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): $\mathscr{E}[e \triangleright \varsigma \mid \bar{e}]$.

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.

Unicity. In order to define the unicity conditions, we use *typed holes* $\{\bar{e}\}$. The terms $\bar{e}$ are required to be well-typed in the current environment (Magic), but their types are independent of the type of the hole: the hole itself may be assigned an arbitrary type.

We introduce an erasure function $\lfloor e \rfloor$, the term counterpart of constraint erasure $\lfloor C \rfloor$, which erases all not-yet-elaborated implicit constructs e^i in e with a typed hole around their subterms. This ensures the subterms—such as type annotations—remain present, so that any constraints they introduce can still contribute to unicity. For example, $\lfloor e.\ell \rfloor$ is $\{\lfloor e \rfloor\}$. The full definition is given in Appendix §A.

Finally, we define a restricted typing judgment $\Gamma \vdash_{\mathsf{robust}} e : \sigma$, which derives $\Gamma \vdash e : \sigma$ without using any fragile, implicit typing rules (*-I). This plays the same role for typing as $(\vdash_{\mathsf{simple}})$ does for constraint satisfiability, ensuring unicity is well-founded. We can now formalize the two unicity conditions as:

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

Implicit polytypes. Unboxing a polytype $\langle e \rangle$ is an *elimination form*. Following the omnidirectional recipe, Use-I requires that e (the principal term) has the unique shape $\nu\bar{\gamma}.\,[\sigma]$ in the context $\mathscr{E}$ (*Step 2*). Following *Step 3*, we 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 nominal type t: the projection $e.\ell$ is typed as $e.\mathsf{t}.\ell$ (*Step 3*) provided the shape of the type of expression e in context $\mathscr{E}$ is a nominal record $\nu\bar{\gamma}.\,\mathsf{t}\ \bar{\gamma}$ (*Step 2*). For record construction, $\{\overline{\ell = e}\}$ is an *introduction form*. In Rcd-I, we type an overloaded record $\{\overline{\ell = e}\}$ as $\mathsf{t}.\{\overline{\ell = e}\}$ (*Step 3*), provided the context $\mathscr{E}$ with subterms $\bar{e}$ expects a nominal record type of shape $\nu\bar{\gamma}.\,\mathsf{t}\ \bar{\gamma}$ (*Step 2*).

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

Example 4.1. Consider the term $\mathtt{ex}_9 \triangleq \lambda r.\,\{\mathtt{x} = r.\mathtt{x}; \mathtt{y} = r.\mathtt{color}\}$. In $\mathtt{ex}_9$, r can only be of type `gray_color`. Indeed, considering the second projection first, we should learn that r is of type `gray_color` (using Rcd-Proj-Closed) and since it is λ-bound, this should then make the first projection unambiguous.

Formally, we derive:

$$\frac{\mathscr{E}[r \triangleright \mathtt{gray\_point}] \qquad \emptyset \vdash \mathscr{E}[r.\mathtt{gray\_point}.\mathtt{x}] : \mathtt{gray\_point} \longrightarrow \mathtt{point}}{\emptyset \vdash \mathscr{E}[r.\mathtt{x}] : \mathtt{gray\_point} \longrightarrow \mathtt{point}}\ \textsc{Rcd-Proj-I}$$

where the context $\mathscr{E}$ is $\lambda r.\,\{\mathtt{x} = \square; \mathtt{y} = r.\mathtt{color}\}$. We have $\emptyset \vdash \mathscr{E}[r.\mathtt{gray\_point}.\mathtt{x}] : \mathtt{gray\_point} \longrightarrow \mathtt{point}$, indeed. Therefore, it remains to show that $\mathscr{E}[r \triangleright \mathtt{gray\_point}]$ (**1**). Assume $\emptyset \vdash \mathscr{E}[\{(r : \mathfrak{g})\}] : \tau$. Since $r.\mathtt{color}$ requires r to have the type `gray_point` (due to Rcd-Proj-Closed), it follows that there is no other choice but to take $\mathfrak{g} = \mathtt{gray\_point}$, which proves (1)

Example 4.2. 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:

$$\frac{\mathscr{E}[r \triangleright \nu\bar{\gamma}.\,\mathsf{t}\ \bar{\gamma}] \qquad \emptyset \vdash \mathscr{E}[r.\mathsf{t}.\mathtt{x}] : \tau}{\emptyset \vdash \mathscr{E}[r.\mathtt{x}] : \tau}\ \textsc{Rcd-Proj-I}$$

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

Example 4.3. Our final example illustrates a limitation of our approach: 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
```

Here, neither field projections 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: $\lambda r.\,r.\mathtt{cie\_point}.\mathtt{y}.\mathtt{cie\_color}.\mathtt{x}$.

Our framework nonetheless rejects $\mathtt{ex}_{10}$. This is intentional: implicit terms 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].

Formally, to type $\texttt{ex}_{10}$ one must eliminate the final implicit projection in a context of the form $\mathscr{E}[e.\ell]$. Two cases arise:

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

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

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

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

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

However, $\mathscr{E}[r \triangleright \texttt{cie\_point}]$ does not hold either. The following judgement $\emptyset \vdash \lfloor\mathscr{E}[(\{r\} : \mathfrak{g})]\rfloor : \tau'$ holds for any $\mathfrak{g}$.

4.4 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 $[\![e : \alpha]\!]$, where e is the term to be translated and α is the expected type of e.

Pattern constraints. Thus far, our formal presentation of constraint patterns has remained abstract, deliberately leaving the syntax and semantics of patterns unspecified to accommodate a range of language features. We now concretize this by specifying the patterns used in OmniML (in Figure 6), and introducing the corresponding constraints for the variables they bind. Patterns include: (1) Tuple patterns $\Pi\ \alpha_j$, matching a tuple type $\Pi_{i=1}^{n}\bar{\tau}$ of arity $n \geq j$, and binding the j-th component to α. (2) Nominal patterns $t\ \_$, binding the name of a nominal type $\mathrm{t}\ \bar{\tau}$ to the nominal variable t. (3) Polytype patterns $[s]$ matching a polytype $[\sigma]$ and binding the resulting scheme to the variable s.

Each new constraint has an unsubstituted form ($s \leq \tau, x \leq s$ *etc.*), whose semantics is defined via substitution into a sugared form ($\sigma \leq \tau, x \leq \sigma$, *etc.*). Semantic environments ϕ are extended to interpret nominal variables t as names t and scheme variables s as ground type schemes $\mathfrak{s}$, that is type schemes with no unbound variables (*i.e.*, $\forall \mathrm{fv}(\tau).\,\tau$).

The function $[\![- : =]\!]$ is defined in Figure 5. All generated type variables are fresh with respect to the expected type α, ensuring capture-avoidance. Unsurprisingly, variables generate an instantiation constraint. Unit () requires the type α to be 1. A function generates a constraint that binds two fresh flexible type variables for the argument and return types. We use a let constraint to bind the argument in the constraint generated for the body of the function. The let constraint is monomorphic since β' is fully constrained by type variables defined outside the abstraction's scope and therefore cannot be generalized. Applications introduce two fresh flexible, one for the argument type and one for the type of the function, typing each subterm with these, ensuring α 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 variables and enforce the equality of the annotated type τ and the expected type α. For polytypes, boxing asserts that e has the polymorphic type σ (using universal quantification) and that the expected type is the polytype $[\sigma]$. Conversely, explicit unboxing

$$
\begin{array}{lcl}
\llbracket x : \alpha \rrbracket & \triangleq & x\ \alpha \\
\llbracket () : \alpha \rrbracket & \triangleq & \alpha = 1 \\
\llbracket \lambda x.\, e : \alpha \rrbracket & \triangleq & \exists \beta, \gamma.\, \alpha = \beta \to \gamma \wedge \mathsf{let}\ x = \lambda \beta'.\, \beta' = \beta\ \mathsf{in}\ \llbracket e : \gamma \rrbracket \\
\llbracket e_1\ e_2 : \alpha \rrbracket & \triangleq & \exists \beta \gamma.\, \gamma = \beta \to \alpha \wedge \llbracket e_1 : \gamma \rrbracket \wedge \llbracket e_2 : \beta \rrbracket \\
\llbracket \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \alpha \rrbracket & \triangleq & \mathsf{let}\ x = \lambda \beta.\, \llbracket e_1 : \beta \rrbracket\ \mathsf{in}\ \llbracket e_2 : \alpha \rrbracket \\
\llbracket (e : \exists \bar{\alpha}.\, \tau) : \alpha \rrbracket & \triangleq & \exists \bar{\alpha}.\, \alpha = \tau \wedge \llbracket e : \alpha \rrbracket \\
\llbracket [e : \exists \bar{\alpha}.\, \sigma] : \alpha \rrbracket & \triangleq & \exists \bar{\alpha}.\, \llbracket e : \sigma \rrbracket \wedge \alpha = [\sigma] \\
\llbracket \langle e : \exists \bar{\alpha}.\, \sigma \rangle : \alpha \rrbracket & \triangleq & \exists \bar{\alpha}, \beta.\, \llbracket e : \beta \rrbracket \wedge \beta = [\sigma] \wedge \sigma \leq \alpha \\
\llbracket \langle e \rangle : \alpha \rrbracket & \triangleq & \exists \beta.\, \llbracket e : \beta \rrbracket \wedge \mathsf{match}\ \beta\ \mathsf{with}\ [s] \to s \leq \alpha \\
\llbracket [e] : \alpha \rrbracket & \triangleq & \mathsf{let}\ x = \lambda \beta.\, \llbracket e : \beta \rrbracket\ \mathsf{in}\ \mathsf{match}\ \alpha\ \mathsf{with}\ [s] \to x \leq s \\
\llbracket e.\ell : \alpha \rrbracket & \triangleq & \begin{cases} \llbracket e.\mathrm{t}.\ell : \alpha \rrbracket & \text{if } \ell \triangleright \mathrm{t} \\ \exists \beta.\, \llbracket e : \beta \rrbracket \wedge \mathsf{match}\ \beta\ \mathsf{with}\ t\ \_ \to t.\ell \leq \beta \to \alpha & \text{otherwise} \end{cases} \\
\llbracket e.\mathrm{t}.\ell : \alpha \rrbracket & \triangleq & \exists \beta.\, \llbracket e : \beta \rrbracket \wedge \mathrm{t}.\ell \leq \beta \to \alpha \\
\llbracket \{\overline{\ell = e}\} : \alpha \rrbracket & \triangleq & \begin{cases} \llbracket \mathrm{t}.\{\overline{\ell = e}\} : \alpha \rrbracket & \text{if } \bar{\ell} \blacktriangleright \mathrm{t} \\ \exists \bar{\beta}.\ \bigwedge_{i=1}^{n} \llbracket e_i : \beta_i \rrbracket & \text{otherwise} \\ \quad \wedge \mathsf{match}\ \alpha\ \mathsf{with}\ t\ \_ \to \mathsf{dom}\ t = \bar{\ell} \wedge \bigwedge_{i=1}^{n} t.\ell_i \leq \alpha \to \beta_i & \end{cases} \\
\llbracket \mathrm{t}.\{\overline{\ell = e}\} : \alpha \rrbracket & \triangleq & \exists \bar{\beta}.\ \bigwedge_{i=1}^{n} \llbracket e_i : \beta_i \rrbracket \wedge \mathsf{dom}\ \mathrm{t} = \bar{\ell} \wedge \bigwedge_{i=1}^{n} \mathrm{t}.\ell_i \leq \alpha \to \beta_i \\
\llbracket \{\bar{e}\} : \alpha \rrbracket & \triangleq & \exists \bar{\beta}.\ \bigwedge_{i=1}^{n} \llbracket e_i : \beta_i \rrbracket \\
\\
\llbracket e : \tau \rrbracket & \triangleq & \exists \alpha.\, \alpha = \tau \wedge \llbracket e : \alpha \rrbracket \\
\llbracket e : \forall \bar{\alpha}.\, \tau \rrbracket & \triangleq & \forall \bar{\alpha}.\, \llbracket e : \tau \rrbracket \\
\\
\llbracket \emptyset \vdash e : \tau \rrbracket & \triangleq & \llbracket e : \tau \rrbracket \\
\llbracket x : \sigma, \Gamma \vdash e : \tau \rrbracket & \triangleq & \mathsf{let}\ x = \lambda \alpha.\, \sigma \leq \alpha\ \mathsf{in}\ \llbracket \Gamma \vdash e : \tau \rrbracket
\end{array}
$$

Fig. 5. The constraint generation translation for OmniML.

requires that α be an instance of σ. By contrast, implicit unboxing suspends until the inferred type of e is known to be a polytype, captured by the pattern $[s]$, at which point we require α to be an instance of s. Implicit boxing infers the principal type for e using a let constraint and suspends until the expected type of the entire term is known to be a polytype, bound to s. We then assert that the principal type of e is at least as general as s, via the constraint $x \leq s$.

Record projections generate a fresh variable for the nominal record type and constrain e to this type, suspending until the type β of e is known to be a nominal record type t. Once resolved, the type of the projected label is retrieved from the global label context Ω and instantiated. For record expressions, we generate a fresh variable β_i for each field assignment to capture the type of each e_i as β_i. The rest of the constraint is deferred until the context determines the type of the whole record type to be t. Once known, the labels are instantiated to match the projection types $\alpha \to \beta_i$, and we additionally check that the domain of t is exactly $\bar{\ell}$, ensuring that every label is defined. Explicit records and projections, along with closed-world disambiguated terms, bypass suspension and directly instantiate the appropriate labels.

Example 4.4. Considering the example from §2.1:

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

$$
\begin{array}{rcll}
\rho & ::= & \Pi\ \alpha_j \mid t\ \_ \mid [s] & \text{Patterns} \\
C & ::= & \ldots \mid \Omega(t.\ell) \leq \tau_1 \to \tau_2 \mid \Omega(\mathsf{t}.\ell) \leq \tau_1 \to \tau_2 & \text{Constraints} \\
 & \mid & s \leq \tau \mid \sigma \leq \tau & \\
 & \mid & x \leq s \mid x \leq \sigma &
\end{array}
$$

$$
\begin{array}{rcll}
\Pi\ \alpha_j \text{ matches } (\nu\bar{\gamma}.\, \Pi_{i=1}^{n}\bar{\gamma})\ \bar{\beta} & \triangleq & [\alpha := \beta_j] & \text{if } n \geq j \\
t\ \_ \text{ matches } (\nu\bar{\gamma}.\, \mathsf{t})\ \bar{\beta} & \triangleq & [t := \mathsf{t}] & \\
[s] \text{ matches } (\nu\bar{\gamma}.\, [\sigma])\ \bar{\beta} & \triangleq & [s := \sigma[\bar{\gamma} := \bar{\beta}]] &
\end{array}
$$

$$
\textsc{Lab-Inst}\ \frac{\phi \vdash \Omega(\phi(t).\ell) \leq \tau_1 \to \tau_2}{\phi \vdash \Omega(t.\ell) \leq \tau_1 \to \tau_2}
\qquad
\textsc{Lab-Dom}\ \frac{\phi \vdash \mathsf{dom}\ (\phi(t)) = \bar{\ell}}{\phi \vdash \mathsf{dom}\ t = \bar{\ell}}
\qquad
\textsc{Scm-Inst}\ \frac{\phi \vdash \phi(s) \leq \tau}{\phi \vdash s \leq \tau}
\qquad
\textsc{Abs-Inst}\ \frac{\phi \vdash x \leq \phi(s)}{\phi \vdash x \leq s}
$$

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

Fig. 6. Patterns for OmniML.

The typing constraint generated for $\mathtt{ex}_9$ contains the following, where α stands for the type of `r`:

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

The suspended constraint can be discharged under our contextual semantics. We apply the Match-Ctx rule with context $\mathscr{C}$ equal to $\mathsf{let}\ x = \lambda\beta.\square\ \mathsf{in}\ x\ \mathtt{int} \wedge \alpha = \mathtt{point}$. 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 will further show in our next example.

Example 4.5. 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 in the **let**-definition alone. In the previous example, this type was unified to a known shape in the **let**-body. Here, this is more subtle: an *instance* of `getx`'s type scheme is taken, which only satisfies the application (`getx one`) if `r` has a variable type or a type of the form `point`. However, the projection `r.x` would be ill-typed if `r` had a variable type (unicity would fail), so `point` is the unique solution. We call this flow of information from instances back to definitions *backpropagation*.

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

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

With the context $\mathscr{C}$ equal to $\mathsf{let}\ \mathit{getx} = \lambda\delta.\, \exists\beta,\gamma.\, \delta = \beta \to \gamma \wedge \square\ \mathsf{in}\ \mathit{getx}\ (\mathtt{point} \to \alpha)$, we can show the unicity predicate $\mathscr{C}[\beta\,!\,\varsigma]$ for the shape ς equal to $\mathtt{point}$. For any $\mathfrak{g}$, the erasure $\lfloor\mathscr{C}[\beta = \mathfrak{g}]\rfloor$ is $\mathsf{let}\ \mathit{getx} = \lambda\delta.\, \exists\beta,\gamma.\, \delta = \beta \to \gamma \wedge \beta = \mathfrak{g}\ \mathsf{in}\ \mathit{getx}\ (\mathtt{point} \to \alpha)$. 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 $\mathfrak{g} = \texttt{point}$. This proves unicity, hence the generated constraint for $\texttt{ex}_{11}$ is satisfiable.

4.5 Metatheory

Constraint generation is sound and complete with respect to the typing judgment. That is to say, the term e is typable with τ if and only if $[\![e : \alpha]\!]$ is satisfiable when α is τ.

Theorem 4.6 (Constraint generation is sound and complete). *Given a closed term e and type τ. Then for any $\alpha \# \tau$, $\vdash e : \tau$ iff $\alpha = \tau \vDash [\![e : \alpha]\!]$.*

Theorem 4.7 (Principal types). *For any well-typed closed 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 Solving constraints

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 in solved form—from which we can read off a most general solution—or the solver becomes stuck, indicating that C is unsatisfiable.

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 7a) 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.

The unification rules are listed in Figure 7b. Rewriting proceeds under an arbitrary context $\mathscr{U}$, modulo α-equivalence and associativity/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 mutli-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 variable 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[7]. This is formalized by the relation $\alpha \prec_U \beta$ indicating that α occurs in a type assigned to β in U. A unification problem is cyclic, written cyclic (U), if $\alpha \prec_U^* \alpha$ for some α.

[7] We discuss relaxing this constraint in ??.

$$
\begin{array}{llll}
U & ::= & \mathsf{true} \mid \mathsf{false} \mid U_1 \wedge U_2 \mid \exists \alpha.\, U \mid \epsilon & \text{Unification problems} \\
\epsilon & ::= & \emptyset \mid \tau = \epsilon & \text{Multi-equations} \\
C & ::= & \ldots \mid \epsilon & \text{Constraints} \\
\mathcal{U} & ::= & \mathcal{U} \wedge U_2 \mid U_1 \wedge \mathcal{U} \mid \exists \alpha.\, \mathcal{U} & \text{Unification context}
\end{array}
\qquad
\textsc{Multi-Unif}\ \frac{\forall \tau \in \epsilon.\ \phi(\tau) = \mathfrak{g}}{\phi \vdash \epsilon}
$$

(a) Syntax and semantics of unification problems.

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

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

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

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

$$
\textsc{S-Unif}\ \frac{U_1 \quad U_1 \longrightarrow U_2}{U_2}
\qquad
\textsc{S-False}\ \frac{\mathscr{C}[\mathsf{false}] \quad \mathscr{C} \neq \square}{\mathsf{false}}
\qquad
\textsc{S-Let}\ \frac{\mathsf{let}\ x = \lambda \alpha.\, C_1\ \mathsf{in}\ C_2}{\mathsf{let}\ x\ \alpha\ [\emptyset] = C_1\ \mathsf{in}\ C_2}
\qquad
\textsc{S-Exists-Conj}\ \frac{(\exists \alpha.\, C_1) \wedge C_2 \quad \alpha \# C_2}{\exists \alpha.\, C_1 \wedge C_2}
$$

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

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

(c) Basic rewriting rules $C_1 \longrightarrow C_2$.

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

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

$$
\textsc{LetR}\ \frac{\phi \vdash \exists \alpha, \bar{\alpha}.\, C_1 \qquad \phi[x := \phi(\lambda \alpha[\bar{\alpha}].\, C_1)] \vdash C_2}{\phi \vdash \mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = C_1\ \mathsf{in}\ C_2}
\qquad
\textsc{AppR}\ \frac{\alpha[\phi'] \in \phi(x) \qquad \phi(\tau) = \phi'(\alpha)}{\phi \vdash x\ \tau}
\qquad
\textsc{Exists-Inst}\ \frac{\alpha[\phi'] \in \phi(x) \qquad \phi[i := \phi'] \vdash C}{\phi \vdash \exists i^x.\, C}
\qquad
\textsc{Incr-Inst}\ \frac{\phi(i)(\alpha) = \phi(\tau)}{\phi \vdash i[\alpha \rightsquigarrow \tau]}
$$

(d) Syntax and semantics of region-based let and incremental instantiation constraints.

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

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

$$\textsc{S-Inst-Unify}\quad \frac{i[\beta \rightsquigarrow \gamma_1] \wedge i[\beta \rightsquigarrow \gamma_2]}{i[\beta \rightsquigarrow \gamma_1] \wedge \gamma_1 = \gamma_2}$$

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

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

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

(e) Solving rules for let-bindings and applications.

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

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

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

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

(f) Rewriting rules for suspended match constraints.

Definition 5.1 (Solved form $\hat{U}$). A solved form is a constraint $\hat{U}$ of the form $\exists \bar{\alpha}.\ \bigwedge_{i=1}^{n} \epsilon_i$, where: (1) each ϵ_i contains at most one non-variable type; (2) each variable may occur as a term in at most one e_i (3) the constraint is acyclic.

5.2 Solving rules

We now gradually introduce the rules of the constraint solver itself (Figures 7c, 7e and 7f). 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.

Basic rules. S-Unif (Figure 7c) invokes the unification algorithm to 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.

In general, existential quantifiers $\exists \alpha.\, C$ are lifted to the nearest enclosing let, if one exists, or otherwise to the top of the constraint. The resulting existential prefix $\exists \bar{\alpha}$ is called a *region*. To make regions explicit, we introduce the syntax $\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = C_1\ \mathsf{in}\ C_2$, 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 7d. The semantics of an abstraction with a region, written $\phi(\lambda\alpha[\bar{\alpha}].\, C)$, is a set of *ground regions* that satisfy C. A ground region is a satisfying interpretation for the region ϕ' with a designated *root* variable α, written $\alpha[\phi']$. Regional let-constraints strictly generalize ordinary constraint abstractions, as captured by the equivalence:

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

In Figure 7c, 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 application, let-binding, or suspended constraint.

OmniML-specific constraints, such as the label and polytype instantiation constraints ($t.\ell \le \tau_1 \to \tau_2$, $s \le \tau$, *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.

Let constraints. Application constraints could be solved by copying constraints:

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

This is rule, due to Pottier and Rémy [2005], resembles β-reduction, except that the original abstraction is retained. While correct for *simple* constraints, it may duplicate solving work across applications of the same abstraction. A more efficient approach first solves the abstraction once—*e.g.* reducing it to $\lambda\alpha[\bar\alpha].\,\bar\epsilon$, where $\bar\alpha$ are generalizable variables—and then reuses the result at each application site by only copying the solved constraint ϵ. This mirrors the ML generalization and instantiation: $\lambda\alpha[\bar\alpha].\,\bar\epsilon$ 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.

However, this approach *does not* extend to suspended constraints. To illustrate this, let us reexamine $\mathtt{ex}_4$ from §2.1:

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

The generated typing constraint contains:

$$\exists\alpha.\ \mathsf{let}\ x = \lambda\beta.\ \mathsf{match}\ \beta\ \mathsf{with}\ (t\ \_) \to \mathscr{C}[t, \alpha, \beta]\ \mathsf{in}\ x\ \mathsf{int} \wedge \alpha = \mathsf{point}$$

where $\mathscr{C}[t, \alpha, \beta]$ is $t.\ell \le \alpha \to \beta$. Here, α stands for `r`'s type. The constraint remains suspended until `(r : point).y` forces `r`'s type to be `point`. Crucially, the variable β (introduced inside the abstraction for the type of `x`) is captured by the suspended match constraint that is not yet resolved at the point of solving the let constraint that binds `x`.

Nonetheless, to continue solving the let-body, we must assign a scheme to `x`. We naively pick $\forall\beta.\,\beta$. This appears unsound, since β will later unify with int once the match constraint is discharged. But it would be incomplete to lower β as a monomorphic variable. This motivates *partial type schemes*, our second novel mechanism for omnidirectional inference. Partial type schemes are type schemes that delay commitment to certain quantifications (*e.g.* β). Such *partially generalized* variables are treated as generalized, but can be incrementally refined in future as suspended constraints are discharged.

To support this, we extend the constraint language with *incremental instantiation constraints* (Figure 7d). Instead of duplicating an abstraction at each application site, we introduce: (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.

Incremental instantiations enables efficient handling of constraint applications: solved parts are reused immediately, while suspended constraints can be solved later, further refining the abstraction and propagating additional equations to the application sites.

The semantics of the existential constraint $\exists i^x.\, C$ (Exists-Inst), given in Figure 7d, introduces the fresh instantiation i by "guessing" a region ϕ' that satisfies the regional constraint abstraction bound to x. Incremental instantiations (Incr-Inst) equate the copy of α in i with τ. The domain of incremental instantiation constraints must lie within the closure of the abstraction or among the regional variables of x. Consequently, the variables $\alpha, \bar{\alpha}$ bound by the let-constraint $\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = C_1$ in C_2 are bound not only in the body of the abstraction C_1, but also in the constraint C_2, where they may appear in incremental instantiations of x in the domain of renamings—and only there. Hence, they cannot appear in C_2 when the corresponding variable x does not itself appear in C_2.

Incremental instantiation constraints are reduced using the following rules, summarized in Figure 7e:

(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-Unify unifies two instantiations if they both refer to the source variable β at same instantiation site i.

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

(1) A nullary shape is copied and no further instantiations are needed (S-Inst-Copy).
(2) The copied variable β is polymorphic, and thus the instantiation constraint imposes no restriction (S-Inst-Poly), provided no other instantiations of β remain (if not, then apply S-Inst-Unify).
(3) The copy is monomorphic and in scope, so we unify it directly (S-Inst-Mono).

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 partial instantiation constraints, written as the substitution $[x.\beta := \gamma]$. We view this as an analogous copy rule for variables.

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.

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 5.2. 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.

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

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, $\exists \beta.\, \alpha = \beta \to \gamma$ determines γ, as γ is free.

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: (1) $\gamma = \epsilon'$ where $\gamma \# \bar{\alpha}, \bar{\beta}$ and $\bar{\alpha} \subseteq \mathsf{fv}(\epsilon')$, or (2) $\bar{\alpha} = \tau = \epsilon'$ 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. By reducing the number of variables that need to be copied, lowering directly reduces instantiation overhead.

S-Let-AppR rewrites an application constraint $x\ \tau$ into 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.

Suspended match constraints. S-Match-Ctx (Figure 7f) solves suspended match constraints when the surrounding context $\mathscr{C}$ *proves* that the scrutinee τ has the unique shape ς, denoted $\vdash \mathscr{C}[\tau\,!\,\varsigma]$. Uni-Type handles the case 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 4.5. 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 a incremental instantiation $i^x[\alpha' \rightsquigarrow \gamma]$ together with a proof that the copy of γ has the unique shape ς. Then α' must also have shape ς, as any other shape would render the instantiation unsatisfiable. This final case (Uni-BackProp) may raise concerns about the well-foundedness of $\vdash \mathscr{C}[\tau\,!\,\varsigma]$. However, well-foundedness follows directly from the fact that in Uni-BackProp the regional depth of the hole strictly decreases. For a solved context $\hat{\mathscr{C}}$, this relation is moreover sound and complete with respect to the semantic definition of unicity (Definition 3.6).

5.3 Metatheory

We establish the correctness of our solver. Correctness follows from three standard metatheoretic properties: *progress*, *preservation*, and *termination*. Together, they ensure that every satisfiable constraint eventually reduces to an equivalent solved form.

Definition 5.3. A constraint C is term-variable-closed if all its term variables x are bound *i.e.*, $\mathsf{fv}(C) \subseteq \mathcal{V}$.

Lemma 5.4 (Scope preservation). *If* $C_1 \longrightarrow C_2$, *then* $\mathsf{fv}(C_1) \supseteq \mathsf{fv}(C_2)$.

Theorem 5.5 (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 $\hat{\mathscr{C}}[\mathsf{match}\ \alpha\ \mathsf{with}\ \bar{\chi}]$ *in* C, $\hat{\mathscr{C}}[\alpha\,!\,\varsigma]$ *does not hold for any* ς.

Theorem 5.6 (Termination). *The constraint solver terminates on all inputs.*

Theorem 5.7 (Preservation). *If* $C_1 \longrightarrow C_2$, *then* $C_1 \equiv C_2$.

Corollary 5.8 (Correctness). *For the term-variable-closed constraint* C, C *is satisfiable if and only if* $C \longrightarrow^* \hat{C}$ *and* $\hat{C}$ *is a solved form equivalent to* C.

6 Implementation

We have two working prototypes 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. One closely follows the constraint-based presentation described here[8]. It is public and open-source[9]. Its implementation is inspired by previous work such as Inferno [Pottier 2014, 2018]. It uses state-of-the-art 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. Many standard ML implementations, for example Inferno, represent the solver state as a linear *stack* of inference regions, from the outermost variable scope to the current region. Unification associates an integer *rank* (or *level*) for each variable, that indexes the region in the stack to which it belongs. This approach does not work for partial generalization. If generalization at some region is suspended by a match constraint, the region must remain alive while we continue inference in other regions. However, later parts of the constraint may introduce a new let-region at the same rank that is unrelated to the suspended one—neither its ancestor nor its descendant—breaking the linear assumption of ranks.

Our implementation must instead use a *tree* of nested let-regions. Under this scheme, ranks no longer uniquely determine a variable's region. Instead, we interpret a rank 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 rank (along this path) suffices to determine the lowered region: we keep the efficient integer comparisons of generalization.

Partial generalization. Partial generalization arises when a region cannot be fully generalized due to suspended constraints that may still 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; treated polymorphically.*
- **(PG)** Variables that are partially generalizable. *Generalizable variables mentioned by suspended match constraint or partial instantiations.*
- **(PI)** Variables that were previously partially generalized but have since been updated. *Awaiting re-generalization. Introduced by the unification of partial generics.*

At generalization time, we conservatively approximate whether a variable may be updated in the future using *guards*. A guard is a mark on a variable that indicates the variable is captured by some suspended constraint that has not yet been solved. Guarded variables are generalized as partial generics (**PG**); unguarded ones are fully generalized (**G**).

When an instance is taken from a partial generic, we retain a forward reference from the partial generic (**PG**) to the instance. This enables the generic to notify the instance that it has been updated, propagating the updated type structure to all instances. This mirrors, in reverse, the way our formalized solver uses incremental instantiation constraints to track copies. In addition, the instance remains guarded by the partial generic 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 partial generics to become fully generalizable. Conversely, if a partially generalized variable is lowered (*e.g.* by S-Lower-Exists), it must be unified with all its instances.

[8] The other prototype is a direct implementation of type inference based on semi-unification. We mention it here only it indicate that we have explored multiple implementation strategies leading to the same results.
[9] Link omitted for anonymity.

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.

7 Related work

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* [**?**]. But they are insufficient when we require a unique resolution of the constraint across all instances—as in *static overloading*.

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, 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.*

Suspended constraints. Suspending constraints that cannot be solved yet is not a novel idea: it is a standard approach to implement unification 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 unsuspended 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, 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 argues [Vytiniotis, Jones, Schrijvers and Sulzmann 2011] 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 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.

Higher-rank Polymorphism. Polytypes are not *higher-rank* in the usual sense; our interest in them stems from their role in OCaml's inference of polymorphic methods. Many systems for higher-rank polymorphism exist; here we highlight a few in the context of ML. $\mathsf{ML^F}$ is an extension of ML that supports first-class polymorphism that goes beyond the power of System F, while retaining type inference. It is a generalization of Garrigue and Rémy [1999]'s polytypes, relying on π-directionality, but it remains unclear how to effectively scale $\mathsf{ML^F}$ to the rest of OCaml's features. FreezeML 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 polytypes. The essential difference is that generalization of higher-rank types is implicit, inferring the most-general type (if one exists). QuickLook, Haskell's latest approach at impredicative higher-rank polymorphism, uses bidirectional propagation to take a "quick look" at the spine of an application to guide instantiation of higher-rank functions. However, this approach inherits the limitations of fixed directionality discussed in §2.3.

8 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.

Future work. We aim to extend our framework to support more advanced features. One direction is generalized *static overloading*; another is *higher-rank 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.

References

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

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

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

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

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

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.

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.

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. 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

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

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

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. 2013. Semi-explicit polymorphic parameters. presented at the ML Family Workshop.

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 \rightarrow \tau_2 \mid \Pi_{i=1}^{n} \tau_i \mid \mathsf{t}\ \bar{\tau} \mid [\sigma] & \text{Types} \\
\sigma & ::= & \tau \mid \forall \alpha.\, \sigma & \text{Type schemes} \\
\mathfrak{g} & & & \text{Ground types} \\
\mathfrak{s} & & & \text{Ground type schemes} \\
\mathfrak{r} & ::= & \alpha[\phi] & \text{Ground region} \\
\mathfrak{G} \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 & \Omega(t.\ell) \leq \tau_1 \rightarrow \tau_2 \mid \mathsf{dom}\ t = \bar{\ell} \mid \Omega(\mathsf{t}.\ell) \leq \tau_1 \rightarrow \tau_2 \mid \mathsf{dom}\ \mathsf{t} = \bar{\ell} & \\
 & \mid & s \leq \tau \mid \sigma \leq \tau \mid x \leq s \mid x \leq \sigma & \\
\chi & ::= & \rho \rightarrow C & \text{Branches} \\
\rho & ::= & \_ \mid \Pi\ \alpha_j \mid t\ \_ \mid [s] & \text{Patterns} \\
\phi & ::= & \emptyset \mid \phi[\alpha := \mathfrak{g}] \mid \phi[x := \mathfrak{G}] \mid \phi[x := \mathfrak{R}] \mid \phi[i := \phi'] & \text{Semantic environment} \\
 & \mid & \phi[t := \mathsf{t}] \mid \phi[s := \mathfrak{s}] & \\
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 \mathsf{t}.\{\overline{\ell = e}\} \mid e.\mathsf{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 & \{\bar{e}\} & \\
\mathscr{E} & ::= & \square \mid \mathscr{E}\ e \mid e\ \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 & \mathsf{t}.\{\ell_1 = e_1\ \ldots\ \ell_i = \mathscr{E}\ \ldots\ \ell_n = e_n\} \mid \mathscr{E}.\mathsf{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 & \{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{t}\ \bar{\alpha} \rightarrow \tau & \text{Label environment}
\end{array}
$$

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

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

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

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

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

$$\textsc{Exists-Inst}\ \frac{\alpha[\phi'] \in \phi(x) \qquad \phi[i := \phi'] \vdash C}{\phi \vdash \exists i^x.\, C} \qquad \textsc{Incr-Inst}\ \frac{\phi(i)(\alpha) = \phi(\tau)}{\phi \vdash i[\alpha \rightsquigarrow \tau]} \qquad \textsc{Lab-Inst}\ \frac{\phi \vdash \Omega(\ell/\phi(t)) \leq \tau_1 \to \tau_2}{\phi \vdash \Omega(t.\ell) \leq \tau_1 \to \tau_2}$$

$$\textsc{Lab-Dom}\ \frac{\phi \vdash \mathsf{dom}\ \phi(t) = \bar\ell}{\phi \vdash \mathsf{dom}\ t = \bar\ell} \qquad \textsc{Scm-Inst}\ \frac{\phi \vdash \phi(s) \leq \tau}{\phi \vdash s \leq \tau} \qquad \textsc{Abs-Inst}\ \frac{\phi \vdash x \leq \phi(s)}{\phi \vdash x \leq s}$$

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

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

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

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

Note: in most definitions, we ignore the additional OmniML constraints, as they are not particularly interesting.

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

$$\textsc{Inst-Shape}\ \frac{\bar\gamma_2 \# \nu\bar\gamma_1.\, \tau}{\nu\bar\gamma_1.\, \tau \preceq \nu\bar\gamma_2.\, \tau[\bar\gamma_1 := \bar\tau_1]}$$

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. $\mathsf{shape}(\tau)$ is the canonical principal shape of τ.

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

$$
\begin{array}{rcll}
\Pi\ \beta_j \text{ matches } (\nu\bar{\gamma}.\,\Pi_{i=1}^{n}\bar{\gamma})\ \bar{\alpha} & \triangleq & [\beta := \alpha_j] & \text{if } n \geq j \\
t\ \_ \text{ matches } (\nu\bar{\gamma}.\,\mathsf{t})\ \bar{\alpha} & \triangleq & [t := \mathsf{t}] & \\
[s] \text{ matches } (\nu\bar{\gamma}.\,[\sigma])\ \bar{\alpha} & \triangleq & [s := \sigma[\bar{\gamma} := \bar{\alpha}]] &
\end{array}
$$

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

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

$$
\textsc{Simple-Unif}\ \frac{}{\tau_1 = \tau_2 \text{ simple}}
\qquad
\textsc{Simple-Let}\ \frac{C_1 \text{ simple} \qquad C_2 \text{ simple}}{\mathsf{let}\ x = \lambda\alpha.\, C_1 \ \mathsf{in}\ C_2 \text{ simple}}
\qquad
\textsc{Simple-App}\ \frac{}{x\ \tau \text{ simple}}
\qquad
\textsc{Simple-LetR}\ \frac{C_1 \text{ simple} \qquad C_2 \text{ simple}}{\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = C_1 \ \mathsf{in}\ C_2 \text{ simple}}
$$

$$
\textsc{Simple-Exists-Inst}\ \frac{C \text{ simple}}{\exists i^{x}.\, C \text{ simple}}
\qquad
\textsc{Simple-Incr-Inst}\ \frac{}{i[\alpha \rightsquigarrow \tau] \text{ simple}}
$$

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

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

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

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

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

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

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

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

$\boxed{\mathsf{t}.\ell \leq \tau_1 \to \tau_2}$ The label ℓ in the nominal record type t has the field type τ_2 and record type τ_1.

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

$\boxed{\ell \triangleright \mathsf{t}}$ The label ℓ infers the unique nominal record type t.

$\boxed{\bar{\ell} \blacktriangleright \mathsf{t}}$ The *closed* set of labels $\bar{\ell}$ infer the unique nominal record type t.

$$
\textsc{Lab-Uni}\quad \frac{\ell \in \mathsf{dom}\ (\mathsf{t}.\Omega) \qquad \forall \mathsf{t}',\ \ell \in \mathsf{dom}\ (\mathsf{t}'.\Omega) \implies \mathsf{t} = \mathsf{t}'}{\ell \triangleright \mathsf{t}}
$$

$$
\textsc{Labs-Uni}\quad \frac{\mathsf{dom}\ (\mathsf{t}.\Omega) = \bar{\ell} \qquad \forall \mathsf{t}',\ \mathsf{dom}\ (\mathsf{t}'.\Omega) = \bar{\ell} \implies \mathsf{t} = \mathsf{t}'}{\bar{\ell} \blacktriangleright \mathsf{t}}
$$

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

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

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

$$\textsc{Let}\ \frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau} \qquad \textsc{Tuple}\ \frac{(\Gamma \vdash e_i : \tau_i)_{i=1}^n}{\Gamma \vdash (e_1, \dots, e_n) : \Pi_{i=1}^n \tau_i} \qquad \textsc{Proj-X}\ \frac{\Gamma \vdash e : \Pi_{i=1}^n \tau_i \qquad 1 \le j \le n}{\Gamma \vdash e.j/n : \tau_j}$$

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

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

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

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

$$\textsc{Rcd-I}\ \frac{\mathscr{E}[\square \triangleleft \nu\bar\gamma.\, \mathrm{t}\ \bar\gamma \mid \bar e] \qquad \Gamma \vdash \mathscr{E}[\mathrm{t}.\{\overline{\ell = e}\}] : \tau}{\Gamma \vdash \mathscr{E}[\{\overline{\ell = e}\}] : \tau} \qquad \textsc{Rcd-Proj-X}\ \frac{\Gamma \vdash e : \tau_1 \qquad \mathrm{t}.\ell \le \tau_1 \to \tau_2}{\Gamma \vdash e.\mathrm{t}.\ell : \tau_2}$$

$$\textsc{Rcd-Proj-Closed}\ \frac{\Gamma \vdash e.\mathrm{t}.\ell : \tau \qquad \ell \triangleright \mathrm{t}}{\Gamma \vdash e.\ell : \tau} \qquad \textsc{Rcd-Proj-I}\ \frac{\mathscr{E}[e \triangleright \nu\bar\gamma.\, \mathrm{t}\ \bar\gamma] \qquad \Gamma \vdash \mathscr{E}[e.\mathrm{t}.\ell] : \tau}{\Gamma \vdash \mathscr{E}[e.\ell] : \tau} \qquad \textsc{Magic}\ \frac{(\Gamma \vdash e_i : \tau_i)_{i=1}^n}{\Gamma \vdash \{\bar e\} : \tau'}$$

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

$\boxed{[\![\Gamma \vdash e : \tau]\!]}$ $[\![\Gamma \vdash e : \tau]\!]$ 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.

$\boxed{[\![e : \alpha]\!]}$ $[\![e : \alpha]\!]$ is satisfiable iff e has the expected type α.

$$
\begin{array}{lcl}
[\![x : \alpha]\!] & \triangleq & x\ \alpha \\
[\![() : \alpha]\!] & \triangleq & \alpha = 1 \\
[\![\lambda x.\, e : \alpha]\!] & \triangleq & \exists \beta, \gamma.\, \alpha = \beta \to \gamma \wedge \mathsf{let}\ x = \lambda \beta'.\, \beta' = \beta\ \mathsf{in}\ [\![e : \gamma]\!] \\
[\![e_1\ e_2 : \alpha]\!] & \triangleq & \exists \beta \gamma.\, \gamma = \beta \to \alpha \wedge [\![e_1 : \gamma]\!] \wedge [\![e_2 : \beta]\!] \\
[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \alpha]\!] & \triangleq & \mathsf{let}\ x = \lambda \beta.\, [\![e_1 : \beta]\!]\ \mathsf{in}\ [\![e_2 : \alpha]\!] \\
[\![(e : \exists \bar{\alpha}.\, \tau) : \alpha]\!] & \triangleq & \exists \bar{\alpha}.\, \alpha = \tau \wedge [\![e : \alpha]\!] \\
[\![(e_1, \ldots, e_n) : \alpha]\!] & \triangleq & \exists \bar{\alpha}.\, \alpha = \Pi_{i=1}^{n} \bar{\alpha} \wedge \bigwedge_{i=1}^{n} [\![e_i : \alpha_i]\!] \\
[\![e.j/n : \alpha]\!] & \triangleq & \exists \beta, \bar{\beta}.\, [\![e : \beta]\!] \wedge \beta = \Pi_{i=1}^{n} \bar{\beta} \wedge \alpha = \beta_j \\
[\![e.j : \alpha]\!] & \triangleq & \exists \beta.\, [\![e : \beta]\!] \wedge \mathsf{match}\ \beta\ \mathsf{with}\ \Pi\ \gamma_j \to \alpha = \gamma \\
[\![[e : \exists \bar{\alpha}.\, \sigma] : \alpha]\!] & \triangleq & \exists \bar{\alpha}.\, [\![e : \sigma]\!] \wedge \alpha = [\sigma] \\
[\![\langle e : \exists \bar{\alpha}.\, \sigma \rangle : \alpha]\!] & \triangleq & \exists \bar{\alpha}, \beta.\, [\![e : \beta]\!] \wedge \beta = [\sigma] \wedge \sigma \le \alpha \\
[\![\langle e \rangle : \alpha]\!] & \triangleq & \exists \beta.\, [\![e : \beta]\!] \wedge \mathsf{match}\ \beta\ \mathsf{with}\ [s] \to s \le \alpha \\
[\![[e] : \alpha]\!] & \triangleq & \mathsf{let}\ x = \lambda \beta.\, [\![e : \beta]\!]\ \mathsf{in}\ \mathsf{match}\ \alpha\ \mathsf{with}\ [s] \to x \le s \\
[\![e.\ell : \alpha]\!] & \triangleq & \begin{cases} \exists \beta.\, [\![e : \beta]\!] \wedge \Omega(\mathrm{t}.\ell) \le \beta \to \alpha & \text{if } \ell \triangleright \mathrm{t} \\ \exists \beta.\, [\![e : \beta]\!] \wedge \mathsf{match}\ \beta\ \mathsf{with}\ t\ \_ \to \Omega(t.\ell) \le \beta \to \alpha & \text{otherwise} \end{cases} \\
[\![e.\mathrm{t}.\ell : \alpha]\!] & \triangleq & \exists \beta.\, [\![e : \beta]\!] \wedge \Omega(\mathrm{t}.\ell) \le \beta \to \alpha \\
[\![\{\overline{\ell = e}\} : \alpha]\!] & \triangleq & \begin{cases} \exists \bar{\beta}.\ \bigwedge_{i=1}^{n} [\![e_i : \beta_i]\!] \wedge \bigwedge_{i=1}^{n} \Omega(\mathrm{t}.\ell_i) \le \alpha \to \beta_i & \text{if } \bar{\ell} \blacktriangleright \mathrm{t} \\ \exists \bar{\beta}.\ \bigwedge_{i=1}^{n} [\![e_i : \beta_i]\!] & \text{otherwise} \\ \quad \wedge \mathsf{match}\ \alpha\ \mathsf{with}\ t\ \_ \to \mathsf{dom}\ t = \bar{\ell} \wedge \bigwedge_{i=1}^{n} \Omega(t.\ell_i) \le \alpha \to \beta_i & \end{cases} \\
[\![\mathrm{t}.\{\overline{\ell = e}\} : \alpha]\!] & \triangleq & \exists \bar{\beta}.\ \bigwedge_{i=1}^{n} [\![e_i : \beta_i]\!] \wedge \mathsf{dom}\ \mathrm{t} = \bar{\ell} \wedge \bigwedge_{i=1}^{n} \Omega(\mathrm{t}.\ell_i) \le \alpha \to \beta_i \\
[\![\{\bar{e}\} : \alpha]\!] & \triangleq & \exists \bar{\beta}.\ \bigwedge_{i=1}^{n} [\![e_i : \beta_i]\!]
\end{array}
$$

$$
\begin{array}{lcl}
[\![e : \tau]\!] & \triangleq & \exists \alpha.\, \alpha = \tau \wedge [\![e : \alpha]\!] \\
[\![e : \forall \bar{\alpha}.\, \tau]\!] & \triangleq & \forall \bar{\alpha}.\, [\![e : \tau]\!]
\end{array}
$$

$$
\begin{array}{lcl}
[\![\emptyset \vdash e : \tau]\!] & \triangleq & [\![e : \tau]\!] \\
[\![x : \sigma, \Gamma \vdash e : \tau]\!] & \triangleq & \mathsf{let}\ x = \lambda \alpha.\, \sigma \le \alpha\ \mathsf{in}\ [\![\Gamma \vdash e : \tau]\!]
\end{array}
$$

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

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

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

$$\textsc{Simple-Poly-X}\ \frac{e \text{ simple}}{[e : \exists\bar{\alpha}.\,\sigma] \text{ simple}} \qquad \textsc{Simple-Use-X}\ \frac{e \text{ simple}}{\langle e : \exists\bar{\alpha}.\,\sigma\rangle \text{ simple}} \qquad \textsc{Simple-Rcd-X}\ \frac{(e_i \text{ simple})_{i=1}^{n}}{\mathsf{t}.\{\ell_1 = e_1 \ \ldots\ \ell_n = e_n\}} \qquad \textsc{Simple-Rcd-Closed}\ \frac{(e_i \text{ simple})_{i=1}^{n} \qquad \bar{\ell} \blacktriangleright \mathsf{t}}{\{\ell_1 = e_1 \ \ldots\ \ell_n = e_n\}}$$

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

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

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

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

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

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

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

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

$$\textsc{Can-Lab-I}\ \frac{\mathscr{L}[\ell\,!\,\mathsf{t}] \qquad \Vdash \mathscr{L}[\mathsf{t}.\ell] : \tau}{\Vdash \mathscr{L}[\ell] : \tau} \qquad \textsc{Can-Rcd-I}\ \frac{\mathscr{E}[\Box \triangleleft \nu\bar{\gamma}.\, \mathsf{t}\ \bar{\gamma} \mid \bar{e}] \qquad \Vdash \mathscr{E}[\mathsf{t}.\{\ell_1 = e_1; \ldots; \ell_n = e_n\}] : \tau}{\Vdash \mathscr{E}[\{\ell_1 = e_1; \ldots; \ell_n = e_n\}] : \tau}$$

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

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

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

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

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

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

$$\textsc{S-Unif}\ \frac{U_1 \qquad U_1 \longrightarrow U_2}{U_2} \qquad \textsc{S-True}\ \frac{C \wedge \mathsf{true}}{C} \qquad \textsc{S-False}\ \frac{\mathscr{C}[\mathsf{false}] \qquad \mathscr{C} \neq \Box}{\mathsf{false}} \qquad \textsc{S-Let}\ \frac{\mathsf{let}\ x = \lambda\alpha.\, C_1\ \mathsf{in}\ C_2}{\mathsf{let}\ x\ \alpha\ [\emptyset] = C_1\ \mathsf{in}\ C_2}$$

$$\textsc{S-Exists-Conj}\ \frac{(\exists\alpha.\, C_1) \wedge C_2 \qquad \alpha \# C_2}{\exists\alpha.\, C_1 \wedge C_2} \qquad \textsc{S-Let-ExistsLeft}\ \frac{\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = \exists\beta.\, C_1\ \mathsf{in}\ C_2 \qquad \beta \# \alpha, \bar{\alpha}, C_2}{\mathsf{let}\ x\ \alpha\ [\bar{\alpha}, \beta] = C_1\ \mathsf{in}\ C_2}$$

$$\textsc{S-Let-ExistsRight}\ \frac{\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = C_1\ \mathsf{in}\ \exists\beta.\, C_2 \qquad \beta \# \alpha, \bar{\alpha}, C_1}{\exists\beta.\, \mathsf{let}\ x = \lambda\bar{\alpha}.\, C_1\ \mathsf{in}\ C_2} \qquad \textsc{S-Let-ConjLeft}\ \frac{\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = C_1 \wedge C_2\ \mathsf{in}\ C_3 \qquad C_1 \# \alpha, \bar{\alpha}}{C_1 \wedge \mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = C_2\ \mathsf{in}\ C_3}$$

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

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

S-Inst-Name
$$\frac{i[\alpha \rightsquigarrow \tau] \qquad \tau \notin \mathcal{V}}{\exists \gamma.\, \gamma = \tau \land i[\alpha \rightsquigarrow \gamma]}$$

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

S-Inst-Copy
$$\frac{\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = C\ \mathsf{in}\ \mathscr{C}[i^x[\alpha' \rightsquigarrow \gamma]] \qquad C = C' \land \alpha' = \varsigma\, \bar{\beta} = \epsilon \qquad \alpha' \in \alpha, \bar{\alpha} \qquad \neg\mathrm{cyclic}\,(C) \qquad \bar{\beta}' \# \alpha', \gamma, \bar{\beta} \qquad x \# \mathrm{bv}(\mathscr{C})}{\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = C\ \mathsf{in}\ \mathscr{C}[\exists \bar{\beta}'.\, \gamma = \varsigma\, \bar{\beta}' \land i^x[\bar{\beta} \rightsquigarrow \bar{\beta}']]}$$

S-Inst-Unify
$$\frac{i[\alpha \rightsquigarrow \gamma_1] \land i[\alpha \rightsquigarrow \gamma_2]}{i[\alpha \rightsquigarrow \gamma_1] \land \gamma_1 = \gamma_2}$$

S-Inst-Poly
$$\frac{\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = \bar{\epsilon} \land C\ \mathsf{in}\ \mathscr{C}[i^x[\alpha' \rightsquigarrow \gamma]] \qquad \forall \alpha'.\, \exists \alpha.\, \bar{\epsilon} \equiv \mathsf{true} \qquad \alpha' \in \alpha, \bar{\alpha} \qquad \alpha' \# C \qquad i.\alpha' \# \mathrm{insts}(\mathscr{C}) \qquad x \# \mathrm{bv}(\mathscr{C})}{\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = \bar{\epsilon} \land C\ \mathsf{in}\ \mathscr{C}[\mathsf{true}]}$$

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

S-Compress
$$\frac{\mathsf{let}\ x\ \alpha\ [\bar{\alpha}, \beta] = C_1 \land \beta = \gamma = \epsilon\ \mathsf{in}\ C_2 \qquad \beta \neq \gamma}{\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = C_1[\beta := \gamma] \land \gamma = \epsilon[\beta := \gamma]\ \mathsf{in}\ C_2[x.\beta := \gamma]}$$

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

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

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

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

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

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

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

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

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

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

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

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

$$\textsc{S-Uni-Var}\quad \dfrac{\alpha \mathrel{\#} \mathsf{bv}(\mathscr{C}_2)}{\vdash \mathscr{C}_1[\alpha = \tau = \epsilon \wedge \mathscr{C}_2[-]][\alpha \,!\, \mathsf{shape}(\tau)]} \qquad \textsc{S-Uni-Type}\quad \dfrac{\tau \notin \mathcal{V}}{\vdash \mathscr{C}[\tau \,!\, \mathsf{shape}(\tau)]}$$

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

Definition A.2. 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.

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

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

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

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

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

Definition A.3 (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.

B Properties of the constraint language

This appendix establishes key properties of the constraint language. The first is the principality of shapes Theorem B.1: 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 B.1 (Principal shapes). *Any non-variable type τ has a non-trivial 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 n-ary nominal 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 3.4) 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, $C_1 \triangleq (\mathsf{match}\ \alpha\ \mathsf{with}\ \_ \to \mathsf{true})$ 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.2 (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 $\mathscr{C}[\tau \mathbin{!} \varsigma]$, 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.3 (Composability of unicity). *If $\mathscr{C}_1[\tau \mathbin{!} \varsigma]$, then $\mathscr{C}_2[\mathscr{C}_1][\tau \mathbin{!} \varsigma]$.*

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 $(\mathscr{C}_3[\mathscr{C}_1] \wedge C)[\tau\,!\,\varsigma]$
	$\phi \vdash \lfloor \mathscr{C}_3[\mathscr{C}_1][\tau = \mathfrak{g}] \rfloor \wedge \lfloor C \rfloor$	$\implies$ I
	$\phi \vdash \lfloor \mathscr{C}_3[\mathscr{C}_1][\tau = \mathfrak{g}] \rfloor$	Simple inversion
$\mathrm{shape}(\mathfrak{g}) = \varsigma$		$\implies$ 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 $(\exists\alpha.\,\mathscr{C}_3[\mathscr{C}_1])[\tau\,!\,\varsigma]$
	$\phi \vdash \exists\alpha.\,\lfloor \mathscr{C}_3[\mathscr{C}_1][\tau = \mathfrak{g}] \rfloor$	$\implies$ I
$\phi[\alpha := \mathfrak{g}'] \vdash \lfloor \mathscr{C}_3[\mathscr{C}_1][\tau = \mathfrak{g}] \rfloor$		Simple inversion
$\mathrm{shape}(\mathfrak{g}) = \varsigma$		$\implies$ 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 $\mathsf{let}\ x = \lambda\alpha.\,\mathscr{C}_3\ \mathsf{in}\ C$.

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

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

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

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

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

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

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

□

Lemma B.4 (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.5 (Decanonicalization). *If $\phi \Vdash C$, then $\phi \vdash C$.*

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

Theorem B.6 (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.5
$\phi \vdash \mathscr{C}[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}] \wedge C_2$	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.3
☞ $\phi \Vdash \mathscr{C}[\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}]$	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.5
$\phi \vdash \exists \alpha.\, \mathscr{C}[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}]$	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.3
☞ $\phi \Vdash \exists \alpha.\, \mathscr{C}[\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}]$	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

$$\frac{\phi \vdash \exists \alpha.\, C_1 \qquad \phi[x := \phi(\lambda \alpha.\, C_1)] \vdash C_2}{\phi \vdash \mathsf{let}\ x = \lambda \alpha.\, C_1\ \mathsf{in}\ C_2}\ \textsc{Let}$$

$$\begin{array}{rl} \phi \vdash \exists \alpha.\, C_1 & \text{Premise} \\ \phi \Vdash \exists \alpha.\, C_1 & \text{By } \textit{i.h.} \\ \phi[x := \phi(\lambda \alpha.\, C_1)] \vdash C_2 & \text{Premise} \\ \phi[x := \phi(\lambda \alpha.\, C_1)] \Vdash C_2 & \text{By } \textit{i.h.} \end{array}$$

By cases on $\phi \Vdash \exists \alpha.\, C_1, \phi[x := \phi(\lambda \alpha.\, C_1)] \Vdash C_2$.

Subcase

$$\frac{\phi \vdash \exists \alpha.\, C_1 \qquad \exists \alpha.\, C_1\ \text{simple}}{\phi \Vdash \exists \alpha.\, C_1}\ \textsc{Can-Base}$$

$$\frac{\phi[x := \phi(\lambda \alpha.\, C_1)] \vdash C_2 \qquad C_2\ \text{simple}}{\phi[x := \phi(\lambda \alpha.\, C_1)] \Vdash C_2}\ \textsc{Can-Base}$$

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

Subcase

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

$\phi[x := \phi(\lambda \alpha.\, C_1)] \Vdash C_2$

$$\begin{array}{ll} (\exists \alpha.\, \mathscr{C})[\tau\,!\,\varsigma] & \text{Premise} \\ \mathscr{C}[\tau\,!\,\varsigma] & \text{Lemma B.4} \\ \phi \Vdash \exists \alpha.\, \mathscr{C}[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}] & \text{Premise} \\ \phi \vdash \exists \alpha.\, \mathscr{C}[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}] & \text{Lemma B.5} \end{array}$$

$$\begin{array}{rl} \phi(\lambda \alpha.\, C_1) = \phi(\lambda \alpha.\, \mathscr{C}[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}]) & \text{Corollary B.8} \\ \phi \vdash \mathsf{let}\ x = \lambda \alpha.\, \mathscr{C}[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}]\ \mathsf{in}\ C_2 & \text{By } \textsc{Let} \\ \phi \Vdash \mathsf{let}\ x = \lambda \alpha.\, \mathscr{C}[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}]\ \mathsf{in}\ C_2 & \text{By } \textit{i.h.} \\ (\mathsf{let}\ x = \lambda \alpha.\, \mathscr{C}\ \mathsf{in}\ C_2)[\tau\,!\,\varsigma] & \text{Lemma B.3} \\ ☞ \quad \phi \Vdash \mathsf{let}\ x = \lambda \alpha.\, \mathscr{C}[\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}]\ \mathsf{in}\ C_2 & \text{By } \textsc{Can-Match-Ctx} \end{array}$$

Subcase

$\phi \Vdash \exists \alpha. C_1$

$$\frac{\mathscr{C}[\tau \,!\, \varsigma] \qquad \phi[x := \phi(\lambda\alpha. C_1)] \Vdash \mathscr{C}[\text{match } \tau := \varsigma \text{ with } \bar{\chi}]}{\phi[x := \phi(\lambda\alpha. C_1)] \Vdash \underbrace{\mathscr{C}[\text{match } \tau \text{ with } \bar{\chi}]}_{C_2}} \text{ Can-Match-Ctx}$$

$\mathscr{C}[\tau \,!\, \varsigma]$	Premise
$(\text{let } x = \lambda\alpha. C_1 \text{ in } \mathscr{C})[\tau \,!\, \varsigma]$	Lemma B.3
$\phi[x := \phi(\lambda\alpha. C_1)] \Vdash \mathscr{C}[\text{match } \tau := \varsigma \text{ with } \bar{\chi}]$	Premise
$\phi[x := \phi(\lambda\alpha. C_1)] \vdash \mathscr{C}[\text{match } \tau := \varsigma \text{ with } \bar{\chi}]$	Lemma B.5
$\phi \vdash \text{let } x = \lambda\alpha. C_1 \text{ in } \mathscr{C}[\text{match } \tau := \varsigma \text{ with } \bar{\chi}]$	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

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

Similar to the True case.

Case

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

Similar to the Let case.

Case

$$\frac{\alpha[\phi'] \in \phi(x) \qquad \phi(\tau) = \phi'(\alpha)}{\phi \vdash x\ \tau} \text{ AppR}$$

Similar to the App case.

Case

$$\frac{\alpha[\phi'] \in \phi(x) \qquad \phi[i := \phi'] \vdash C}{\phi \vdash \exists i^x. C} \text{ Exists-Inst}$$

Similar to the Exists case.

Case

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

Similar to the Unif case.

Case

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

Similar to the App case.

□

Lemma B.7 (Inversion of suspension). *If* $\phi \vdash \mathscr{C}[\texttt{match } \tau \texttt{ with } \bar{\chi}]$ *and* $\mathscr{C}[\tau \,!\, \varsigma]$, *then* $\phi \vdash \mathscr{C}[\texttt{match } \tau := \varsigma \texttt{ with } \bar{\chi}]$.

Proof. We use canonicalization (Theorem B.6) to induct on $\phi \Vdash \mathscr{C}[\texttt{match } \tau \texttt{ with } \bar{\chi}]$ instead of $\phi \vdash \mathscr{C}[\texttt{match } \tau \texttt{ with } \bar{\chi}]$.

This simplifies the proof, but introduces a circular dependency between Theorem B.6 and Lemma B.7. However, this does not compromise the well-foundedness of induction, as the application of Lemma B.7 (via Corollary B.8) within the proof of Theorem B.6 is restricted to strictly smaller constraints.

Case

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

The second premise is a contradiction.

Case

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

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

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

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

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

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

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

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

$$\begin{aligned} &\phi' \vdash \lfloor \mathscr{C}_2[\tau = \mathfrak{g}', \texttt{match } \tau' := \varsigma' \texttt{ with } \bar{\chi}'] \rfloor && \Longrightarrow \mathrm{I} \\ &\phi' \vdash \lfloor \mathscr{C}_2[\tau = \mathfrak{g}', \texttt{true}] \rfloor && \text{Lemma B.2} \end{aligned}$$

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

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

$$
\begin{array}{lrll}
 & \phi' & \vdash \lfloor \mathscr{C}_2[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}, \tau' = \mathfrak{g}'] \rfloor & \Longrightarrow \mathrm{I} \\
 & \phi' & \vdash \lfloor \mathscr{C}_2[\mathsf{true}, \tau' = \mathfrak{g}'] \rfloor & \text{Lemma B.2} \\
 & \lfloor \mathscr{C}_2[\mathsf{true}, \tau' = \mathfrak{g}'] \rfloor & = \lfloor \mathscr{C}_2[\lfloor \mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi} \rfloor, \tau' = \mathfrak{g}'] \rfloor & \text{By definition} \\
 & & = \lfloor \mathscr{C}'[\tau' = \mathfrak{g}'] \rfloor & \text{By definition} \\
 & \phi' & \vdash \lfloor \mathscr{C}[\tau = \mathfrak{g}'] \rfloor & \text{Above} \\
 & & \mathscr{C}'[\tau' \,!\, \varsigma'] & \text{Premise} \\
 & \mathsf{shape}(\mathfrak{g}') = \varsigma' & & \Longrightarrow \mathrm{E}\ \text{on}\ \mathscr{C}'[\tau' \,!\, \varsigma'] \\
 & & \mathscr{C}_2[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}, \square][\tau' \,!\, \varsigma'] & \text{Above} \\
\text{☞} & \phi & \Vdash \mathscr{C}_2[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}, \mathsf{match}\ \tau'\ \mathsf{with}\ \bar{\chi}'] & \text{By Con-Match-Ctx}
\end{array}
$$

□

Corollary B.8. *If* $\mathscr{C}[\tau \,!\, \varsigma]$, *then* $\phi(\lambda\alpha.\,\mathscr{C}[\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}]) = \phi(\lambda\alpha.\,\mathscr{C}[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}])$. *Similarly,* $\phi(\lambda\alpha[\bar{\alpha}].\,\mathscr{C}[\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}]) = \phi(\lambda\alpha[\bar{\alpha}].\,\mathscr{C}[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}])$.

Proof. It is sufficient to show that $\phi[\alpha := \mathfrak{g}] \vdash \mathscr{C}[\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}]$ if and only if $\phi \vdash \mathscr{C}[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}]$.

Case $\Longrightarrow$.

$$
\begin{array}{lrl}
 & \mathscr{C}[\tau \,!\, \varsigma] & \text{Premise} \\
 & \phi[\alpha := \mathfrak{g}] \vdash \mathscr{C}[\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}] & \text{Premise} \\
\text{☞} & \phi[\alpha := \mathfrak{g}] \vdash \mathscr{C}[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}] & \text{Lemma B.7}
\end{array}
$$

Case $\Longleftarrow$.

$$
\begin{array}{lrl}
 & \mathscr{C}[\tau \,!\, \varsigma] & \text{Premise} \\
 & \phi[\alpha := \mathfrak{g}] \vdash \mathscr{C}[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}] & \text{Premise} \\
\text{☞} & \phi[\alpha := \mathfrak{g}] \vdash \mathscr{C}[\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}] & \text{By Match-Ctx}
\end{array}
$$

For $\phi(\lambda\alpha[\bar{\alpha}].\,\mathscr{C}[\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}]) = \phi(\lambda\alpha[\bar{\alpha}].\,\mathscr{C}[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}])$, the proof is identical.

□

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 $\mathsf{match}\ \alpha\ \mathsf{with}\ \bar{\chi} \equiv \mathsf{false}$, yet $\mathscr{C}[\mathsf{match}\ \alpha\ \mathsf{with}\ \bar{\chi}] \not\equiv \mathscr{C}[\mathsf{false}]$ for $\mathscr{C} := \square \wedge \alpha = \mathsf{int}$.

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.2. □

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.6
$\mathscr{C}'[\tau\,!\,\varsigma]$	Inversion of Can-Match-Ctx
$\phi \Vdash \mathscr{C}'[\text{match } \tau := \varsigma \text{ with } \bar{\chi}]$	″
$\mathscr{C}[C_1] = \mathscr{C}'[\text{match } \tau := \varsigma \text{ with } \bar{\chi}]$	″
$= \mathscr{C}_2[\text{match } \tau := \varsigma \text{ with } \bar{\chi}, C_1]$	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:

$$\mathscr{C}_1 \equiv^P_\square \mathscr{C}_2 \triangleq \forall \bar{C}.\ P(\bar{C}) \Longrightarrow \mathscr{C}_1[\bar{C}] \equiv_{\text{ctx}} \mathscr{C}_2[\bar{C}]$$

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.2. $\square$

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.6
$\phi \Vdash \mathscr{C}'[\text{match } \tau := \varsigma \text{ with } \bar{\chi}]$	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.

$\square$

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:

$$\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = \mathscr{C}[\bar{\square}]\ \mathsf{in}\ \mathscr{C}'[C_1] \equiv^{P}_{\square} \mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = \mathscr{C}[\bar{\square}]\ \mathsf{in}\ \mathscr{C}'[C_2]$$

for any match-closed guard P on the holes.

Proof. Let us assume (†):

$$\forall \phi, \phi', \bar{C}.\ \phi'(x) = \phi(\lambda\alpha[\bar{\alpha}].\mathscr{C}[\bar{C}]) \implies \phi' \vdash C_1 \iff \phi' \vdash C_2$$

We proceed by induction on the number of suspended match constraints in $\mathscr{C}'', \mathscr{C}', \bar{C}$ with the statement $P(n) := \forall \mathscr{C}'', \mathscr{C}', \bar{C}.\ \#\mathsf{match}\ \mathscr{C}'', \mathscr{C}', \bar{C} = n \implies \mathscr{C}''[\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = \mathscr{C}[\bar{C}]\ \mathsf{in}\ \mathscr{C}'[C_1]] \equiv \mathscr{C}''[\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = \mathscr{C}[\bar{C}]\ \mathsf{in}\ \mathscr{C}'[C_2]]$.

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 (‡):

$$\forall \phi, \phi'.\ \phi'(x) = \phi(\lambda\alpha[\bar{\alpha}].\mathscr{C}[\bar{C}]) \implies \phi' \vdash \mathscr{C}'[C_1] \iff \phi' \vdash \mathscr{C}'[C_2]$$

This holds due to the compositionality of simple equivalence using † as a base case.

Subcase $\implies$.

$\phi \vdash \mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = \mathscr{C}[\bar{C}]\ \mathsf{in}\ \mathscr{C}'[C_1]$	Premise
$\phi \vdash \exists \alpha, \bar{\alpha}.\ \mathscr{C}[\bar{C}]$	Simple inversion
$\phi[x := \phi(\lambda\alpha[\bar{\alpha}].\mathscr{C}[\bar{C}])] \vdash \mathscr{C}'[C_1]$	″
$\phi[x := \phi(\lambda\alpha[\bar{\alpha}].\mathscr{C}[\bar{C}])] \vdash \mathscr{C}'[C_2]$	‡
$\phi \vdash \mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = \mathscr{C}[\bar{C}]\ \mathsf{in}\ \mathscr{C}'[C_2]$	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**

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

$\tau \notin \mathcal{V}$	Premise
$\tau = \mathsf{shape}(\tau)\ \bar{\tau}$	For some $\bar{\tau}$
For all $\phi, \mathfrak{g}$	Defn. of $\mathscr{C}[\tau\,!\ \mathsf{shape}(\tau)]$
$\phi \vdash \lfloor \mathscr{C}[\tau = \mathfrak{g}] \rfloor$	Premise
$\phi_1 \vdash \tau = \mathfrak{g}$	Inversion of $\lfloor \mathscr{C}_1 \rfloor$
$\mathfrak{g} = \phi_1(\tau)$	Simple inversion
$= \mathsf{shape}(\tau)\ \phi_1(\bar{\tau})$	″
☞ $\mathsf{shape}(\mathfrak{g}) = \mathsf{shape}(\tau)$	Applying shape to both sides

Case

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

$$\begin{array}{ll}
\alpha \mathbin{\#} \mathrm{bv}(\mathscr{C}_2) & \text{Premise} \\
\tau = \mathrm{shape}(\tau)\ \bar{\tau} & \text{For some } \bar{\tau} \\
\text{For all } \phi, \mathfrak{g} & \text{Defn. of } \mathscr{C}[\alpha \,!\, \mathrm{shape}(\tau)] \\
\quad \phi \vdash \lfloor \mathscr{C}_1[\alpha = \mathrm{shape}(\tau)\ \bar{\tau} = \epsilon \wedge \mathscr{C}_2[\alpha = \mathfrak{g}]] \rfloor & \text{Premise} \\
\quad \phi_1 \vdash \alpha = \mathrm{shape}(\tau)\ \bar{\tau} = \epsilon & \text{Inversion of } \lfloor \mathscr{C}_1 \rfloor \\
\quad \phi_2 \vdash \alpha = \mathfrak{g} & \text{Inversion of } \lfloor \mathscr{C}_2 \rfloor \\
\quad \mathfrak{g} = \phi_2(\alpha) & \text{Simple inversion} \\
\quad\quad = \phi_1(\alpha) & \alpha \mathbin{\#} \mathrm{bv}(\mathscr{C}_2) \\
\quad\quad = \mathrm{shape}(\tau)\ \phi_1(\bar{\tau}) & \text{Simple inversion} \\
☞\ \mathrm{shape}(\mathfrak{g}) = \mathrm{shape}(\tau) & \text{Applying shape to both sides}
\end{array}$$

Case

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

$$\begin{array}{ll}
\alpha' \in \alpha, \bar{\alpha} & \text{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] \wedge -][\gamma \,!\, \varsigma] & '' \\
\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = \mathscr{C}_1[\mathsf{true}]\ \mathsf{in}\ \mathscr{C}_2[i^x[\alpha' \rightsquigarrow \gamma] \wedge -][\gamma \,!\, \varsigma] & \text{By } \textit{i.h.} \\
\text{For all } \phi, \mathfrak{g} & \text{Defn. of } \ldots[\alpha \,!\, \mathrm{shape}(\tau)] \\
\quad \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 & \text{Premise} \\
\quad \text{Let } \phi_1 = \phi[x := \phi(\lambda\alpha[\bar{\alpha}].\lfloor \mathscr{C}_1[\alpha' = \mathfrak{g}] \rfloor)]. & \\
\quad \phi'(\alpha') = \mathfrak{g} & \text{For any } \alpha[\phi'] \in \phi_1(x) \\
\quad \phi_2 \vdash i^x[\alpha' \rightsquigarrow \gamma] & \text{Inversion of } \lfloor \mathscr{C}_2 \rfloor \\
\quad \phi_2(i^x)(\alpha') = \phi_2(\gamma) & \text{Simple inversion} \\
\quad \phi_2(i^x) \in \phi_2(x) & \text{Since } \exists i^x. \in \mathscr{C}_2, \phi_2 \text{ extends } \phi_1 \\
\quad \phi_2(i^x)(\alpha') = \mathfrak{g} & \text{Above} \\
\quad\quad = \phi_2(\gamma) & '' \\
\quad \phi_1 \vdash \lfloor \mathscr{C}_2[i^x[\alpha' \rightsquigarrow \gamma] \wedge \gamma = \mathfrak{g}] \rfloor & \text{Entailment for } \lfloor \mathscr{C}_2 \rfloor \\
\quad \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] \wedge \gamma = \mathfrak{g}] \rfloor & \text{By } \textsc{LetR} \\
\quad \phi \vdash \mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = \mathscr{C}_1[\mathsf{true}]\ \mathsf{in}\ \mathscr{C}_2[i^x[\alpha' \rightsquigarrow \gamma] \wedge \gamma = \mathfrak{g}] & \text{Simple congruence} \\
☞\ \mathrm{shape}(\mathfrak{g}) = \varsigma & \implies E \text{ on } \ldots[\gamma \,!\, \varsigma]
\end{array}$$

□

Lemma C.11. *If $\mathscr{C}$ is normalized, then $\mathscr{C}[\tau \,!\, \varsigma]$ if and only $\vdash \mathscr{C}[\tau \,!\, \varsigma]$.*

Proof.

Case $\implies$.

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-Lower-Exists 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 C.13 (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

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

$U_1 \longrightarrow U_2$	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

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

$\alpha \# C_2$ 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-ExistsLeft*, *S-Let-Exists-InstLeft*, *S-Let-ExistsRight*, *S-Let-Exists-InstRight*, *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

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

$\vdash \mathscr{C}[\tau \,!\, \varsigma]$	Premise
$\mathscr{C}[\tau \,!\, \varsigma]$	Lemma C.10
Sufficient to show equivalences between constraints.	Lemma B.3

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.7

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

$$\frac{\text{let } x\ \alpha\ [\bar{\alpha}] = C_1 \text{ in } \mathscr{C}[x\ \tau] \qquad \gamma \# \tau \qquad x \# \text{bv}(\mathscr{C})}{\text{let } x\ \alpha\ [\bar{\alpha}] = C_1 \text{ in } \mathscr{C}[\exists \gamma, i^x. \gamma = \tau \wedge i[\alpha \rightsquigarrow \gamma]]} \longrightarrow \text{S-Let-AppR}$$

$\gamma \# \tau$	Premise
$x \# \text{bv}(\mathscr{C})$	Premise

Sufficient to show equivalence between $x\ \tau$ and $\exists \gamma, i^x. \gamma = \tau \wedge i[\alpha \rightsquigarrow \gamma]$.	Lemma C.9
Suppose $\phi'(x) = \phi(\lambda \alpha[\bar{\alpha}]. C_1)$.	Premise

Subcase $\Longrightarrow$.

$\phi' \vdash x\ \tau$	Premise
$\alpha[\phi_1] \in \phi(x)$	Simple inversion
$\phi_1(\alpha) = \phi'(\tau)$	″
$\phi'[\gamma := \phi'(\tau), i := \phi_1] \vdash i[\alpha \rightsquigarrow \gamma]$	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

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

$x \# \text{bv}(\mathscr{C})$	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
$\alpha[\phi_1] \in \phi(x)$	$\exists i^x. \in \mathscr{C}$
$\phi'(i) = \phi_1$	″
$\phi'(\gamma) = \phi(i)(\alpha')$	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

$$\frac{i[\alpha \rightsquigarrow \gamma_1] \wedge i[\alpha \rightsquigarrow \gamma_2]}{i[\alpha \rightsquigarrow \gamma_1] \wedge \gamma_1 = \gamma_2}\ \textsc{S-Inst-Unif}$$

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

$$\frac{\text{let } x\ \alpha\ [\bar{\alpha}] = \bar{\epsilon} \wedge C \text{ in } \mathscr{C}[i^x[\alpha' \rightsquigarrow \gamma]] \quad \forall \alpha'.\, \exists \alpha, \bar{\alpha}.\, \bar{\epsilon} \equiv \text{true} \quad \alpha' \in \alpha, \bar{\alpha} \quad \alpha' \# C \quad i.\alpha' \# \text{insts}(\mathscr{C}) \quad x \# \text{bv}(\mathscr{C})}{\text{let } x\ \alpha\ [\bar{\alpha}] = \bar{\epsilon} \wedge C \text{ in } \mathscr{C}[\text{true}]} \longrightarrow \textsc{S-Inst-Poly}$$

$\forall \alpha'.\, \exists \alpha, \bar{\alpha}.\, \bar{\epsilon} \equiv \text{true}$	Premise
$\alpha' \# C$	Premise
$i.\alpha' \# \text{insts}(\mathscr{C})$	Premise
$x \# \text{bv}(\mathscr{C})$	Premise

Sufficient to show equivalence between $i^x[\alpha' \rightsquigarrow \gamma]$ and true.	Lemma C.9
Suppose $\phi'(x) = \phi(\lambda\alpha[\bar{\alpha}, \alpha'].\, \bar{\epsilon} \wedge C)$.	Premise

Subcase $\Longrightarrow$.

	$\phi' \vdash i^x[\alpha' \rightsquigarrow \gamma]$	Premise
☞	$\phi' \vdash \text{true}$	By True

Subcase $\Longleftarrow$.

$\phi' \vdash \text{true}$	Premise
$\alpha[\phi_1] \in \phi'(x)$	$\mathscr{C} = \mathscr{C}_1[\exists i^x.\, \mathscr{C}_2]$
$\phi'(i) = \phi_1$	″

By cases on $\phi_1(\alpha')$.

Subsubcase $\phi_1(\alpha') = \phi'(\gamma)$.

	$\phi_1(\alpha') = \phi'(\gamma)$	Premise
☞	$\phi' \vdash i^x[\alpha' \rightsquigarrow \gamma]$	By Incr-Inst

Subsubcase $\phi_1(\alpha') \neq \phi'(\gamma)$.

Let $\phi_2 = \phi_1[\alpha' := \phi'(\gamma)]$.

$\phi_1 \vdash \bar{\epsilon} \wedge C$	By definition
$\phi_1 \vdash \bar{\epsilon}$	Simple inversion
$\phi_2 \vdash \bar{\epsilon}$	α' is polymorphic
$\phi_2 \vdash C$	$\alpha' \# C$
$\phi_2 \vdash \bar{\epsilon} \wedge C$	By Conj
$\alpha[\phi_2] \in \phi(x)$	By definition
Suppose $\phi_3 \vdash \mathscr{C}_2[\text{true}]$.	Considering entailment on $\exists i^x$.
$\phi_3(i) = \phi_1$	''
$\phi_3[i := \phi_2] \vdash \mathscr{C}_2[\text{true}]$	$i.\alpha' \# \text{insts}(\mathscr{C}_2)$
$\mathscr{D} :: \phi_3 \vdash \mathscr{C}_2[\text{true}]$	By Exists-Inst

$\mathscr{D}$ is a derivation that satisfies $\phi_1(\alpha') = \phi'(\gamma)$.

☞ So this case degenerates to the former case.

Case

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

$\beta \# \alpha, \bar{\alpha}$	Premise
$x, \beta \# \text{bv}(\mathscr{C})$	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
$\alpha[\phi_1] \in \phi(C)$	$\exists i^x. \in \mathscr{C}$
$\phi'(i) = \phi_1$	''
$\phi'(\gamma) = \phi_1(\beta)$	Simple inversion
$\phi_1(\beta) = \phi(\beta)$	$\beta \# \alpha, \bar{\alpha}$
$\phi'(\beta) = \phi(\beta)$	$\beta \# \text{bv}(\mathscr{C})$
$\phi'(\gamma) = \phi'(\beta)$	Above
$\phi' \vdash \gamma = \beta$	By Unif

Subcase $\Longleftarrow$.

Symmetric argument.

Case

$$\frac{\text{let } x\ \alpha\ [\bar{\alpha}] = \bar{\epsilon} \text{ in } C \qquad x \# C \qquad \exists \alpha, \bar{\alpha}. \bar{\epsilon} \equiv \text{true}}{C} \longrightarrow \text{S-Let-Solve}$$

$x \# C$	Premise
$\exists \alpha, \bar{\alpha}. \bar{\epsilon} \equiv \text{true}$	

Sufficient to show equivalence for simple constraints.	Lemma C.8
Suppose C simple.	Premise

Subcase $\Longrightarrow$.

For all ϕ

$$\begin{array}{rll} & \phi \vdash \mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = \bar{\epsilon}\ \mathsf{in}\ C & \text{Premise} \\ & \phi \vdash \exists \alpha, \bar{\alpha}.\, \bar{\epsilon} & \text{Simple inversion} \\ \phi[x := \phi(\lambda\alpha[\bar{\alpha}].\, \bar{\epsilon})] \vdash C & & '' \\ ☞ & \phi \vdash C & x \mathrel{\#} C \end{array}$$

Subcase $\Longleftarrow$.

For all ϕ

$$\begin{array}{rll} & \phi \vdash C & \text{Premise} \\ \phi[x := \phi(\lambda\alpha[\bar{\alpha}].\, \bar{\epsilon})] \vdash C & & x \mathrel{\#} C \\ & \phi \vdash \exists \alpha, \bar{\alpha}.\, \bar{\epsilon} & \\ ☞ & \phi \vdash \mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = \bar{\epsilon}\ \mathsf{in}\ C & \text{By LetR} \end{array}$$

Case

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

$$\begin{array}{rl} \exists \alpha, \bar{\alpha}.\, C_1\ \mathsf{determines}\ \bar{\beta} & \text{Premise} \\ \text{Sufficient to show equivalence for simple constraints.} & \text{Lemma C.8 and Lemma C.7} \\ \text{Suppose } C_1, C_2 \text{ simple.} & \text{Premise} \end{array}$$

Subcase $\Longrightarrow$.

$$\begin{array}{rll} & \phi \vdash \mathsf{let}\ x\ \alpha\ [\bar{\alpha}, \bar{\beta}] = C_1\ \mathsf{in}\ C_2 & \text{Premise} \\ & \phi \vdash \exists \alpha, \bar{\alpha}, \bar{\beta}.\, C_1 & \text{Simple inversion} \\ \phi[x := \phi(\lambda\alpha[\bar{\alpha}, \bar{\beta}].\, C_1)] \vdash C_2 & & '' \\ \phi[\alpha := \mathfrak{g}, \bar{\alpha} := \bar{\mathfrak{g}}, \bar{\beta} := \bar{\mathfrak{g}}'] \vdash C_1 & & '' \\ & \phi[\bar{\beta} := \bar{\mathfrak{g}}'] \vdash \exists \alpha, \bar{\alpha}.\, C_1 & \text{By Exists} \end{array}$$

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 $\Longrightarrow$.

$$\begin{array}{rl} \phi[\alpha := \mathfrak{g}_1, \bar{\alpha} := \bar{\mathfrak{g}}_1, \bar{\beta} := \bar{\mathfrak{g}}_2] \vdash C_1 & \text{Premise} \\ \phi[\bar{\beta} := \bar{\mathfrak{g}}_2] \vdash \exists \alpha, \bar{\alpha}.\, C_1 & \text{By Exists} \\ \bar{\mathfrak{g}}_2 = \bar{\mathfrak{g}}' & \text{By definition of determines} \\ ☞\ \phi[\bar{\beta} := \bar{\mathfrak{g}}', \alpha := \mathfrak{g}_1, \bar{\alpha} := \bar{\mathfrak{g}}_1] \vdash C_1 & \text{Above} \end{array}$$

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.14 (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.15 (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 $i.\alpha \#$* insts*$(\hat{\mathscr{C}})$; (d) for every match constraint $\hat{\mathscr{C}}[\text{match } \alpha \text{ with } \bar{\chi}]$ in C, $\hat{\mathscr{C}}[\alpha\,!\,\varsigma]$ does not hold for any ς. Here, $\hat{\mathscr{C}}$ is a normal context,* i.e.*, such that no other rewrites can be applied.*

Proof. We proceed by induction on the structure of C. We focus on suspended match constraints, conjunctions, and let rules.

Case match τ 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 $\square[\alpha\,\xcancel{!}]$. It suffices that every match constraint in a context-reachable position $\hat{\mathscr{C}}[\text{match } \alpha' \text{ with } \bar{\chi}]$ satisfies $\hat{\mathscr{C}}[\alpha'\,\xcancel{!}]$. By the definition of constraint contexts, there is only one such $\hat{\mathscr{C}}$, namely $\square$, for which we already have $\square[\alpha\,\xcancel{!}]$. Hence match τ 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 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 $x \# \text{bv}(\hat{\mathscr{C}}_1)$.

$\hat{\mathscr{C}}_1[x\ \tau] \wedge C_2$ 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 $x \# \text{bv}(\hat{\mathscr{C}}_1)$ and $i.\alpha \# \text{insts}(\hat{\mathscr{C}}_1)$.

If $i.\alpha \in \text{insts}(C_2)$ and $i \# \text{bv}(\hat{C}_1)$, 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, $x \# \text{bv}(\hat{\mathscr{C}}_1 \wedge C_2)$, thus $\hat{\mathscr{C}}_1[i^x[\alpha \rightsquigarrow \gamma]]$ is stuck.

Subsubcase $\hat{\mathscr{C}}_1[\text{match } \alpha' \text{ with } \bar{\chi}]$. We have $\mathscr{C}_1[\alpha'\,\xcancel{!}]$.

Consider a match constraint match α' 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 $(\mathscr{C}_1 \wedge C_2)[\alpha'\,\xcancel{!}]$.

Case let $x\ \alpha\ [\bar{\alpha}] = C_1$ 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 $x \# \mathsf{bv}(\hat{\mathscr{C}})$.
Apply S-Let-AppR.
Subsubcase $\hat{\mathscr{C}}[i^x[\alpha' \rightsquigarrow \gamma]]$. We have $x \# \mathsf{bv}(\hat{\mathscr{C}})$ or $i.\alpha' \# \mathsf{insts}(\hat{\mathscr{C}})$.

- *$\alpha' \in \alpha, \bar{\alpha}$.*
  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-Lower-Exists (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 $\hat{\mathscr{C}}[\alpha' \xcancel{!}]$.
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.

$\square$

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.16 (Unification termination). *The unifier terminates on all inputs.*

Proof. Let every shape ς have an integer *weight* defined by $\mathsf{sw}\ (\varsigma) \triangleq 4 + 2 \times |\varsigma|$, where $|\varsigma|$ is the arity of the shape ς. The weight of a type $\mathsf{tw}\ (\tau)$ is defined by:

$$
\begin{array}{rcl}
\mathsf{tw}\ (\alpha) & \triangleq & 1 \\
\mathsf{tw}\ (\varsigma\ \bar{\tau}) & \triangleq & \mathsf{iw}\ (\varsigma\ \bar{\tau}) - 2 \\[4pt]
\mathsf{iw}\ (\alpha) & \triangleq & 0 \\
\mathsf{iw}\ (\varsigma\ \bar{\tau}) & \triangleq & \mathsf{sw}\ (\varsigma) + \mathsf{iw}\ (\bar{\tau}) \\
\mathsf{iw}\ (\bar{\tau}) & \triangleq & \sum_{i=1}^{n} \mathsf{iw}\ (\tau_i)
\end{array}
$$

The helper $\mathsf{iw}\ (\tau)$ 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 $\mathsf{uw}\ (U)$ 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 $\mathsf{uw}\ (U)$ strictly decreases. The remaining rules are obviously destructive and either maintain or decrease the unification weight.

Case

$$
\frac{\varsigma\ \bar{\alpha} = \varsigma\ \bar{\beta} = \epsilon}{\varsigma\ \bar{\alpha} = \epsilon \wedge \bar{\alpha} = \bar{\beta}} \longrightarrow \text{U-Decomp}
$$

We have:

$$
\begin{array}{rrcl}
(+) & \mathsf{uw}\ (\varsigma\ \bar{\alpha} = \varsigma\ \bar{\beta} = \epsilon) & = & \mathsf{tw}\ (\varsigma\ \bar{\alpha}) + \mathsf{tw}\ (\varsigma\ \bar{\beta}) + \mathsf{tw}\ (\epsilon) \\
(-) & \mathsf{uw}\ (\varsigma\ \bar{\alpha} = \epsilon \wedge \bar{\alpha} = \bar{\beta}) & = & \mathsf{tw}\ (\varsigma\ \bar{\alpha}) + \mathsf{tw}\ (\epsilon) + \mathsf{tw}\ (\bar{\alpha}) + \mathsf{tw}\ (\bar{\beta}) \\
\hline
 & & = & \mathsf{tw}\ (\varsigma\ \bar{\beta}) - \mathsf{tw}\ (\bar{\alpha}) - \mathsf{tw}\ (\bar{\beta}) \\
 & & = & (\mathsf{sw}\ (\varsigma) + 0 - 2) - 2|\varsigma| \\
 & & = & (2 + 2|\varsigma|) - 2|\varsigma| \;\; = \;\; \mathbf{2}
\end{array}
$$

Hence $\mathsf{uw}\ (\varsigma\ \bar{\alpha} = \varsigma\ \bar{\beta} = \epsilon) > \mathsf{uw}\ (\varsigma\ \bar{\alpha} = \epsilon \wedge \bar{\alpha} = \bar{\beta})$.

Case

$$
\frac{\varsigma\ (\bar{\tau}, \tau_i, \bar{\tau}') = \epsilon \qquad \alpha \# \bar{\tau}, \bar{\tau}', \epsilon \qquad \tau_i \notin \mathcal{V}}{\exists \alpha.\, \varsigma\ (\bar{\tau}, \alpha, \bar{\tau}') = \epsilon \wedge \alpha = \tau_i} \longrightarrow \text{U-Name}
$$

Given $\tau_i \notin \mathcal{V}$, by Theorem B.1, $\tau_i = \varsigma'\ \bar{\tau}''$ for some shape ς' and types $\bar{\tau}''$. So we have:

$$
\begin{array}{rrcl}
(+) & \mathsf{uw}\ (\varsigma\ (\bar{\tau}, \tau_i, \bar{\tau}') = \epsilon) & = & \mathsf{sw}\ (\varsigma) + \mathsf{iw}\ (\bar{\tau}) + \mathsf{iw}\ (\tau_i) + \mathsf{iw}\ (\bar{\tau}') - 2 + \mathsf{uw}\ (\epsilon) \\
(-) & \mathsf{uw}\ (\exists \alpha.\, \alpha = \tau_i \wedge \varsigma\ (\bar{\tau}, \alpha, \bar{\tau}') = \epsilon) & = & \mathsf{sw}\ (\varsigma) + \mathsf{iw}\ (\bar{\tau}) + 0 + \mathsf{iw}\ (\bar{\tau}') - 2 + \mathsf{uw}\ (\epsilon) + 1 + \mathsf{tw}\ (\tau_i) \\
\hline
 & & = & \mathsf{iw}\ (\tau_i) - \mathsf{iw}\ (\alpha) - \mathsf{tw}\ (\tau_i) - 1 \\
 & & = & \mathsf{iw}\ (\tau_i) - 0 - (\mathsf{iw}\ (\tau_i) - 2) - 1 \\
 & & = & \mathbf{1}
\end{array}
$$

Hence $\mathsf{uw}\ (\varsigma\ (\bar{\tau}, \tau_i, \bar{\tau}') = \epsilon) > \mathsf{uw}\ (\exists \alpha.\, \varsigma\ (\bar{\tau}, \alpha, \bar{\tau}') = \epsilon \wedge \alpha = \tau_i)$.

□

Theorem C.17 (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 a 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 $(\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = C_1\ \mathsf{in}\ C_2)$ 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:

$$
\begin{aligned}
\mathsf{tw}\ (\varsigma\ \bar{\tau}) &\triangleq 2 \times \mathsf{sw}\ (\varsigma) + \textstyle\sum_{i=1}^{n} \mathsf{tw}\ (\tau_i) \\
\mathsf{tw}\ (\alpha) &\triangleq \begin{cases} \sup \{\mathsf{tw}\ (\tau) : \alpha = \tau \in C_1\} & \text{if } C_1 \text{ is acyclic} \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

The weight of a partial instantiation $\mathsf{cw}\ (i^x [\alpha \rightsquigarrow \tau])$ is defined as the sum of $\mathsf{tw}\ (\tau)$ and $\mathsf{tw}\ (\alpha)$. The weight of other constraints is given using the measure uw defined in the the proof of Lemma C.16.

Case

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

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 $i^x[\alpha' \rightsquigarrow \gamma]$.

$$\begin{aligned}
\mathsf{cw}\ (i^x[\alpha' \rightsquigarrow \gamma]) &= 1 + \mathsf{tw}\ (\alpha) \\
&\geq 1 + 2 \times \mathsf{sw}\ (\varsigma) + \textstyle\sum_{i=1}^{n} \mathsf{tw}\ (\beta_i) \\
\mathsf{cw}\ (\exists \bar\beta'.\, \gamma = \varsigma\,\bar\beta' \wedge i^x[\bar\beta \rightsquigarrow \bar\beta']) &= 1 + \mathsf{sw}\ (\varsigma) + \textstyle\sum_{i=1}^{n} \mathsf{tw}\ (\beta_i) + |\bar\beta'|
\end{aligned}$$

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.18. *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.19 (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.20. *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 $\implies$.

By transfinite induction on the well-ordering of constraints whose existence is shown in Theorem C.17.

We have C is satisfiable. By Theorem C.15, we have three cases:

Subcase *C is solved*. We have $C \longrightarrow^* C$ and $C \equiv C$ by reflexitivity. 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.18, 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 C.13, we have $C \equiv C'$, thus C' is satisfiable. Additionally, by Lemma C.19, 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 $\impliedby$.

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 C.13, $C \equiv C'$, 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 $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} e_1 : \tau'$, $\bar{\alpha} \# \Gamma$, 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* $1 \le i \le n$ *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* $n \ge j$.
(ix) *If* $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} [e : \exists \bar\alpha.\, \forall \bar\beta.\, \tau'] : \tau$, *then* $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} e : \tau[\bar\alpha := \bar\tau]$, $\bar\beta \mathbin{\#} \Gamma$ *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* $\sigma \le \tau$.
(xi) *If* $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} \{\bar e\} : \tau$, *then* $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} e_i : \tau'_i$ *for all* $1 \le i \le n$.
(xii) *If* $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} \mathsf{t}.\{\overline{\ell = e}\} : \tau$, *then* $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} e_i : \tau_i$ *and* $\mathsf{t}.\ell \le \tau \to \tau_i$ *for* $1 \le i \le n$ *and* $\mathrm{dom}\ (\mathsf{t}.\Omega) = \bar\ell$.
(xiii) *If* $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} \{\overline{\ell = e}\} : \tau$, *then* $\bar\ell \blacktriangleright \mathsf{t}$ *and* $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} \mathsf{t}.\{\overline{\ell = e}\} : \tau$.
(xiv) *If* $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} e.\mathsf{t}.\ell : \tau$, *then* $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} e : \tau'$, $\mathsf{t}.\ell \le \tau' \to \tau$.
(xv) *If* $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} e.\ell : \tau$, *then* $\ell \triangleright \mathsf{t}$ *and* $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} e.\mathsf{t}.\ell : \tau$.

D.2 Canonicalization of typability

Our system satisfies a similar canonicalization theorem to constraint satisfiability.

LEMMA D.4 (COMPOSABILITY OF UNICITY).
(i) *If* $\mathscr{E}_1[\Box \triangleleft \varsigma \mid \bar e]$, *then* $\mathscr{E}_2[\mathscr{E}_1][\Box \triangleleft \varsigma \mid \bar e]$.
(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:

$$\|e\| \triangleq \langle \#\mathrm{implicit}\ e, |e|\rangle$$

where $\langle \ldots \rangle$ denotes a lexicographically ordered pair, and

(1) $\#\mathrm{implicit}\ e$ 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 $\vartheta(\alpha) \ne \alpha$ for any $\alpha \in \mathrm{dom}\ \vartheta$, 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 $\bar\beta = \mathrm{rng}\ \vartheta$, $\bar\alpha = \mathrm{dom}\ \vartheta$ 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 $\exists\vartheta \models \exists\vartheta'$, 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

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

$$
\begin{array}{lcl}
[\![\square[\square:\alpha]:\alpha]\!] & \triangleq & \square \\
[\![(\mathscr{E}\ e)[\square:\alpha']:\alpha]\!] & \triangleq & \exists\beta\gamma.\, \gamma = \beta \to \alpha \wedge [\![\mathscr{E}[\square:\alpha']:\gamma]\!] \wedge [\![e:\beta]\!] \\
[\![(e\ \mathscr{E})[\square:\alpha']:\alpha]\!] & \triangleq & \exists\beta\gamma.\, \gamma = \beta \to \alpha \wedge [\![e:\gamma]\!] \wedge [\![\mathscr{E}[\square:\alpha']:\beta]\!] \\
[\![(\mathsf{let}\ x = \mathscr{E}\ \mathsf{in}\ e)[\square:\alpha']:\alpha]\!] & \triangleq & \mathsf{let}\ x = \lambda\beta.\, [\![\mathscr{E}[\square:\alpha']:\beta]\!]\ \mathsf{in}\ [\![e:\alpha]\!] \\
[\![(\mathsf{let}\ x = e\ \mathsf{in}\ \mathscr{E})[\square:\alpha']:\alpha]\!] & \triangleq & \mathsf{let}\ x = \lambda\beta.\, [\![e:\beta]\!]\ \mathsf{in}\ [\![\mathscr{E}[\square:\alpha']:\alpha]\!] \\
[\![(\mathscr{E}:\exists\bar{\alpha}.\,\tau)[\square:\alpha']:\alpha]\!] & \triangleq & \exists\bar{\alpha}.\, \alpha = \tau \wedge [\![\mathscr{E}[\square:\alpha']:\alpha]\!] \\
[\![(e_1,\ldots,\mathscr{E}_j,\ldots,e_n)[\square:\alpha']:\alpha]\!] & \triangleq & \exists\bar{\alpha}.\, \alpha = \Pi_{i=1}^n \alpha_i \wedge \bigwedge_{i\neq j}[\![e_i:\alpha_i]\!] \wedge [\![\mathscr{E}_j[\square:\alpha']:\alpha_j]\!] \\
[\![(\mathscr{E}.j/n)[\square:\alpha']:\alpha]\!] & \triangleq & \exists\beta,\bar{\beta}.\, [\![\mathscr{E}[\square:\alpha']:\beta]\!] \wedge \beta = \Pi_{i=1}^n\beta_i \wedge \alpha = \beta_j \\
[\![(\mathscr{E}.j)[\square:\alpha']:\alpha]\!] & \triangleq & \exists\beta.\, [\![\mathscr{E}[\square:\alpha']:\beta]\!] \wedge \mathsf{match}\ \beta\ \mathsf{with}\ \Pi\ \gamma_j \to \alpha = \gamma \\
[\![[\mathscr{E}:\exists\bar{\alpha}.\,\sigma][\square:\alpha']:\alpha]\!] & \triangleq & \exists\bar{\alpha}.\, [\![\mathscr{E}[\square:\alpha']:\sigma]\!] \wedge \alpha = [\sigma] \\
[\![\langle\mathscr{E}:\exists\bar{\alpha}.\,\sigma\rangle[\square:\alpha']:\alpha]\!] & \triangleq & \exists\bar{\alpha},\beta.\, [\![\mathscr{E}[\square:\alpha']:\beta]\!] \wedge \beta = [\sigma] \wedge \sigma \leq \alpha \\
[\![[\mathscr{E}][\square:\alpha']:\alpha]\!] & \triangleq & \mathsf{let}\ x = \lambda\beta.\, [\![\mathscr{E}[\square:\alpha']:\beta]\!]\ \mathsf{in} \\
 & & \mathsf{match}\ \alpha\ \mathsf{with}\ [s] \to x \leq s \\
[\![\langle\mathscr{E}\rangle[\square:\alpha']:\alpha]\!] & \triangleq & \exists\beta.\, [\![\mathscr{E}[\square:\alpha']:\beta]\!] \wedge \mathsf{match}\ \beta\ \mathsf{with}\ [s] \to s \leq \alpha \\
[\![(\mathscr{E}.\ell)[\square:\alpha']:\alpha]\!] & \triangleq & \begin{cases} \exists\beta.\, [\![\mathscr{E}[\square:\alpha']:\beta]\!] \wedge \Omega(\mathrm{t}.\ell) \leq \beta \to \alpha & \text{if } \ell \triangleright \mathrm{t} \\ \exists\beta.\, [\![\mathscr{E}[\square:\alpha']:\beta]\!] & \\ \wedge \mathsf{match}\ \beta\ \mathsf{with}\ t\ \_ \to \Omega(t.\ell) \leq \beta \to \alpha & \text{otherwise} \end{cases} \\
[\![(\mathscr{E}.\mathrm{t}.\ell)[\square:\alpha']:\alpha]\!] & \triangleq & \exists\beta.\, [\![\mathscr{E}[\square:\alpha']:\beta]\!] \wedge \Omega(\mathrm{t}.\ell) \leq \beta \to \alpha \\
[\![\{\ell_1 = e_1;\ldots;\ell_j = \mathscr{E}_j;\ldots;\ell_n = e_n\}[\square:\alpha']:\alpha]\!] & \triangleq & \begin{cases} \exists\bar{\beta}.\ \bigwedge_{i\neq j}[\![e_i:\beta_i]\!] \wedge [\![\mathscr{E}_j[\square:\alpha']:\beta_j]\!] & \text{if } \bar{\ell} \blacktriangleright \mathrm{t} \\ \wedge \bigwedge_{i=1}^n \Omega(\mathrm{t}.\ell_i) \leq \alpha \to \beta_i & \\ \exists\bar{\beta}.\ \bigwedge_{i=1}^n [\![e_i:\beta_i]\!] & \text{otherwise} \\ \ \wedge\ \mathsf{match}\ \alpha\ \mathsf{with}\ t\ \_ \to \mathsf{dom}\ t = \bar{\ell} & \\ \qquad\qquad \wedge \bigwedge_{i=1}^n \Omega(t.\ell_i) \leq \alpha \to \beta_i & \end{cases} \\
[\![\mathrm{t}.\{\ell_1 = e_1;\ldots;\ell_j = \mathscr{E}_j;\ldots;\ell_n = e_n\}[\square:\alpha']:\alpha]\!] & \triangleq & \exists\bar{\beta}.\ \bigwedge_{i\neq j}[\![e_i:\beta_i]\!] \wedge [\![\mathscr{E}_j[\square:\alpha']:\beta_j]\!] \wedge \mathsf{dom}\ \mathrm{t} = \bar{\ell} \\
 & & \wedge \bigwedge_{i=1}^n \Omega(\mathrm{t}.\ell_i) \leq \alpha \to \beta_i \\
[\![\{e_1,\ldots,\mathscr{E}_j,\ldots,e_n\}[\square:\alpha']:\alpha]\!] & \triangleq & \exists\bar{\beta}.\ \bigwedge_{i\neq j}[\![e_i:\beta_i]\!] \wedge [\![\mathscr{E}_j[\square:\alpha']:\beta_j]\!] \\
 & & \\
[\![\mathscr{E}[\square:\alpha']:\tau]\!] & \triangleq & \exists\alpha.\, \alpha = \tau \wedge [\![\mathscr{E}[\square:\alpha']:\alpha]\!] \\
[\![\mathscr{E}[\square:\alpha']:\forall\bar{\alpha}.\,\tau]\!] & \triangleq & \forall\bar{\alpha}.\, [\![\mathscr{E}[\square:\alpha']:\tau]\!]
\end{array}
$$

Lemma D.12. *For any term context $\mathscr{E}$, term e, $[\![\mathscr{E}[\square:\alpha]:\beta]\!][[\![e:\alpha]\!]] = [\![\mathscr{E}[e]:\beta]\!]$.*

Proof. By induction on the structure of $\mathscr{E}$. □

Lemma D.13. *For any term e, $\lfloor[\![e:\alpha]\!]\rfloor = [\![\lfloor e\rfloor:\alpha]\!]$.*

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
$\emptyset \vdash^{sd}_{simple} e : \vartheta(\alpha) \iff$	ϑ unifies $[\![e : \alpha]\!]$	Lemma D.14
$\emptyset \vdash^{sd}_{simple} e : \vartheta(\alpha) \iff$	$\Vdash e : \vartheta(\alpha)$	When e simple
☞ $\Vdash e : \vartheta(\alpha) \iff$	ϑ unifies $[\![e : \alpha]\!]$	Above

Case n *is* $k + 1$.

Subcase $\Longrightarrow$.

Subsubcase

$$\frac{\mathscr{E}[e \triangleright \nu\bar{\gamma}.\,\Pi^n_{i=1}\bar{\gamma}] \qquad \vartheta(\Gamma) \Vdash \mathscr{E}[e.j/n] : \vartheta(\alpha)}{\Vdash \mathscr{E}[e.j] : \vartheta(\alpha)}\ \textsc{Can-Proj-I}$$

$$\begin{array}{rcll}
& & \vartheta(\Gamma) \Vdash \mathscr{E}[e.j/n] : \vartheta(\alpha) & \text{Premise} \\
& & \vartheta \text{ unifies } [\![\Gamma \vdash \mathscr{E}[e.j/n] : \alpha]\!] & \text{By } \textit{i.h.} \\
[\![\Gamma \vdash \mathscr{E}[e.j/n] : \alpha]\!] & = & \text{let } \Gamma \text{ in } [\![\mathscr{E}[e.j/n] : \alpha]\!] & \text{By definition} \\
& = & \text{let } \Gamma \text{ in } [\![\mathscr{E}[\square : \beta] : \alpha]\!][[\![e.j/n : \beta]\!]] & \text{Lemma D.12} \\
[\![e.j/n : \beta]\!] & \equiv & \exists \alpha_1 \bar{\gamma}.\,[\![e : \alpha_1]\!] \wedge \alpha_1 = \Pi^n_{i=1}\bar{\gamma} \wedge \beta = \gamma_j & \text{By definition} \\
& \equiv & \exists \alpha_1.\,[\![e : \alpha_1]\!] \wedge \text{match } \alpha_1 := \nu\bar{\gamma}.\,\Pi^n_{i=1}\bar{\gamma} \text{ with } \Pi\,\gamma_j \to \beta = \gamma & '' \\
& & \vartheta \text{ unifies let } \Gamma \text{ in } [\![\mathscr{E}[\square : \beta] : \alpha]\!][\exists \alpha_1.\,[\![e : \alpha_1]\!] \wedge \ldots] & \text{Above} \\
& & \mathscr{E}[e \triangleright \nu\bar{\gamma}.\,\Pi^n_{i=1}\bar{\gamma}] & \text{Premise} \\
& & \text{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 & \text{Premise} \\
\exists \alpha_1.\,[\![e : \alpha_1]\!] \wedge \alpha_1 = \mathfrak{g} & = & \exists \alpha_1.\,[\![(e : \mathfrak{g}) : \alpha_1]\!] & \text{By definition} \\
& = & [\![\{(e : \mathfrak{g})\} : \beta]\!] & '' \\
\lfloor \mathscr{C}[\alpha_1 = \mathfrak{g}] \rfloor & = & \lfloor \text{let } \Gamma \text{ in } [\![\mathscr{E}[\square : \beta] : \alpha]\!][[\![\{(e : \mathfrak{g})\} : \beta]\!]] \rfloor & '' \\
& = & \lfloor \text{let } \Gamma \text{ in } [\![\mathscr{E}[\{(e : \mathfrak{g})\}] : \alpha]\!] \rfloor & \text{Lemma D.12} \\
& = & \text{let } \Gamma \text{ in } \lfloor [\![\mathscr{E}[\{(e : \mathfrak{g})\}] : \alpha]\!] \rfloor & \text{By definition} \\
& = & \text{let } \Gamma \text{ in } [\![\lfloor \mathscr{E}[\{(e : \mathfrak{g})\}] \rfloor : \alpha]\!] & \text{Lemma D.13} \\
& & \phi \text{ unifies let } \Gamma \text{ in } [\![\lfloor \mathscr{E}[\{(e : \mathfrak{g})\}] \rfloor : \alpha]\!] & \text{Above} \\
& & \Vdash \lfloor \mathscr{E}[\{(e : \mathfrak{g})\}] \rfloor : \phi(\alpha) & \text{By } \textit{i.h.} \\
& & \emptyset \vdash \lfloor \mathscr{E}[\{(e : \mathfrak{g})\}] \rfloor : \phi(\alpha) & \text{Lemma D.5} \\
\text{shape}(\mathfrak{g}) & = & \nu\bar{\gamma}.\,\Pi^n_{i=1}\bar{\gamma} & \Longrightarrow \text{E} \\
& & \mathscr{C}[\alpha_1 \,!\, \nu\bar{\gamma}.\,\Pi^n_{i=1}\bar{\gamma}] & \text{Above} \\
& & \vartheta \text{ unifies } \mathscr{C}[\text{match } \alpha_1 \text{ with } \Pi\,\gamma_j \to \beta = \gamma] & \text{By } \textsc{Match-Ctx} \\
[\![e.j : \beta]\!] & = & \exists \alpha_1.\,[\![e : \alpha_1]\!] \wedge \text{match } \alpha_1 \text{ with } \ldots & \text{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]\!]] & \text{Above} \\
& = & \text{let } \Gamma \text{ in } [\![\mathscr{E}[e.j] : \alpha]\!] & \text{Lemma D.12} \\
& = & [\![\mathscr{E}[e.j] : \alpha]\!] & \\
☞ & & \vartheta \text{ unifies } [\![\mathscr{E}[e.j] : \alpha]\!] &
\end{array}$$

Subsubcase *Can-Poly-I*, *Can-Use-I*, *Can-Rcd-I*, *Can-Rcd-Proj-I*.

Similar arguments.

Subcase $\Longleftarrow$.

Subsubcase

$$\dfrac{\mathscr{C}[\alpha_1 \,!\, \nu\bar{\gamma}.\,\Pi_{i=1}^{n}\bar{\gamma}] \qquad \vartheta \text{ unifies } \mathscr{C}[\mathsf{match}\ \alpha_1 := \nu\bar{\gamma}.\,\Pi_{i=1}^{n}\bar{\gamma} \ \mathsf{with}\ \ldots]}{\vartheta \text{ unifies } \mathscr{C}[\underbrace{\mathsf{match}\ \alpha_1\ \mathsf{with}\ \Pi\ \gamma_j \to \beta = \gamma}_{[\![e:\alpha]\!]}]}\ \textsc{Can-Match-Ctx}$$

$[\![e : \tau]\!] = \mathsf{let}\ \Gamma\ \mathsf{in}\ [\![\mathscr{E}[e.j] : \alpha]\!]$	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.*
$\Gamma' \vdash \mathscr{E}[\{(e : \mathfrak{g})\}] : \tau'$	Premise
$\Gamma' = \emptyset$	$\mathscr{E}[\{(e : \mathfrak{g})\}]$ is closed
$\Vdash \mathscr{E}[\{(e : \mathfrak{g})\}] : \tau'$	Lemma D.5
$[\alpha := \tau']$ unifies $[\![\mathscr{E}[\{(e : \mathfrak{g})\}] : \alpha]\!]$	By *i.h.*
$\phi[\alpha := \phi(\tau')] \vdash [\![\mathscr{E}[\{(e : \mathfrak{g})\}] : \alpha]\!]$	By definition
$\mathscr{C}[\alpha_1 \,!\, \nu\bar{\gamma}.\,\Pi_{i=1}^{n}\bar{\gamma}]$	Premise
$\mathrm{shape}(\mathfrak{g}) = \nu\bar{\gamma}.\,\Pi_{i=1}^{n}\bar{\gamma}$	$\Longrightarrow$ 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.

$\square$

D.5 Principal types

Theorem D.16 (Principal types). *For any well-typed closed 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.20, 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). $\square$

E Further study

E.1 Defaulting

Default rules, which does not fit well with π-inference are still often used in practice, and therefore would deserve further investigation.

Default shapes. In this section we study a particular form of defaulting where, rather than general default rules that could fire any constraint, we restrict to default shapes. That is, we may attach a default shape ς to a match constraints match τ with $\bar{\chi}$, which is then written match τ with $\bar{\chi}$ default ς. The default shape ς can then be used to force the shape of τ when it could not be determined from context.

Restricting to default shapes has several benefits. First, a strategy $\mathcal{S}$ can be reduced to the choice of a mapping from constrains C to of a subset $\mathcal{S}(C)$ of suspended constraints of c that should be defaulted, simultaneously. The behavior is then entirely determined, reusing the same logic that runs when the shape is determined by the context instead of been forced by the default clause. In particular, this ensures that the same behavior could have been obtained by an explicit shape constraint in the source.

Strategies. We write $\mathcal{S}_{\mathbb{0}}$ for the empty strategy that never defaults (and thus fails on all constraints with leftover suspensions) and $\mathcal{S}_{\mathbb{1}}$ the full strategy that defaults all suspended constraints, simultaneously. A strategy $\mathcal{S}$ is *reasonable* if for all constraints C, $\mathcal{S}(C)$ succeeds more often than the empty strategy on C. This criterion rules out weird strategies that would default a suspension before solving the other constraints that could discharge this suspension, possibly with another shape, hence a different output.

A strategy $\mathcal{S}$ applied to a constraint C, either allows to solve C, hence with a principal solution written $[\![C]\!]_{\mathcal{S}}$ or ends in error ($[\![C]\!]_{\mathcal{S}}$ is equal to $\perp$). Let use write $[\![C]\!]$ for the union of all $[\![C]\!]_{\mathcal{S}}$ for all successful reasonable strategies $\mathcal{S}$. We say that $\mathcal{S}$ is non-ambiguous if, for any C, $[\![C]\!]_{\mathcal{S}}$ is $\perp$ whenever $[\![C]\!]$ has more than two elements. This condition forces a non-ambiguous strategy to fail instead of picking an arbitrary solution when different defaulting strategies would give incompatible solutions.

A *good* strategy as one that is both *reasonable* and *non-ambiguous*. A good strategy should not fail more often that $\mathcal{S}_{\mathbb{0}}$ nor succeed when there are more than one possible solution. We claim that there is an optimal *good* strategy $\mathcal{S}_{\text{opt}}$ that explores all possible default subsets, succeeds when there is exactly one principal solution, then following a successful strategy, and fails otherwise.

Unfortunately, $\mathcal{S}_{\text{opt}}$ is inefficient: as described , it runs in time exponential in the number of remaining suspended constraints. Therefore, we should seek for sub-optimal, but more efficient good strategies.

Contents

Abstract		1
1	Introduction	1
2	Overview	3
3	Constraints	9
4	The OmniML calculus	14
5	Solving constraints	22
6	Implementation	27
7	Related work	29
8	Conclusions	30
References		30
A	Full technical reference	33
B	Properties of the constraint language	44
C	Properties of the constraint solver	52
D	Properties of OmniML	68
E	Further study	74
Contents		75