Omnidirectional type inference for ML: principality any way

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 which typically infer the types of let-bound expressions before typechecking their use sites. This 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 *partial type schemes*, 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 striking for several advanced ML extensions, typically those that rely on optional type annotations where principality is fragile. We illustrate them with OCaml's static overloading of record labels and datatype constructors, semi-explicit first-class polymorphism, and tuple projections *à la* SML.

1 Introduction

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

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, and well-typedness is preserved under common program transformations such as let-contraction and argument reordering.

Over the years, many extensions of ML have been proposed. Some of them, such as extensible records with row-polymorphism, higher-kinded types, or dimensional types, fit perfectly into the ML framework. Others such as GADTs, higher-rank polymorphism, or static overloading, are *fragile*, as they sometimes require 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, the type of the `match` scrutinee and return type can be annotated to a rigid type which will be refined by type equalities in each branch. Those type annotations may sometimes—but not always—be omitted.

Consider impredicative higher-rank polymorphism for instance:

```
let self f = f f
```

With higher-rank types, one could *guess* the type of `f` to be either $\forall \alpha.\, \alpha \to \alpha$ or $\forall \alpha.\, \alpha \to \alpha \to \alpha$ in order to typecheck `self`—neither of which is more general than the other, violating principality.

To fix this, inference algorithms require a minimal amount of *known* type information to restore principality; in this example the binding of `f` should be annotated with a polymorphic type scheme. Yet specifying such requirements declaratively is difficult. As a result, the 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, ML^F [Le Botlan and Rémy 2009] accepts or rejects the following expression, depending on the position of the annotation (using a traffic-light scheme: green and red indicate typechecking success and failure; orange signals a warning):

```
let self' (f : ∀α. α → α) = if true then f f else f                MLF
let self' f = if true then f f else (f : ∀α. α → α)                MLF
```

Author's Contact Information:

1.1 Directional type inference

Each fragile construct admits a robust counterpart where the type annotation is mandatory. While robust constructs fit perfectly into the ML framework, they are significantly more cumbersome to use, as they always require explicit type annotations. Fragile constructs can be defined by elaboration into their robust counterpart. The difficulty lies in finding a specification that is sufficiently expressive, principled, intuitive for the user, and for which we have a complete and effective elaboration algorithm.

The solutions proposed so far all enforce some ordering in which type inference is performed, which can then be used to propagate both inferred types and user-provided type annotations as *known* types that can be used for disambiguation and enable the omission of some annotations.

π-directional type inference. Most ML inference algorithms enforce a fixed order when typechecking let-bindings let $x = e_1$ in e_2: first typecheck the definition e_1, then the body e_2. OCaml leverages this ordering to resolve overloaded or ambiguous constructs in a *principal* way: polymorphic types are treated as *known* and may guide disambiguation, whereas monomorphic types 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. This strategy ensures principality for fragile constructs, but can lead to counter-intuitive behavior.

To illustrate the problem, consider the following two record types with overlapping field names:

```
type α one = {x : α; y : int}
type two = {x : int; z : int}
```

In OCaml, both definitions are in scope, and the compiler must statically disambiguate field usage. The expression `{x = 1; z = 1}` can only be `two`, and `r.y` necessarily infers `one` for `r`. But field accesses such as `r.x` are ambiguous unless the type of `r` is *known*. Consider:

```
let ex₁ r = r.x                                  OCaml  OmniML
let ex₂ = let r = {x = 1; y = 1} in r.x          OCaml  OmniML
let ex₃ = (fun r → r.x) {x = 1; y = 1}           OCaml  OmniML
```

In $\mathtt{ex}_1$, the type of `r` is unconstrained, so disambiguation fails[1]. At first glance, $\mathtt{ex}_2$ and $\mathtt{ex}_3$ appear equivalent: in both, the expression `r.x` can only refer to the field from type `int one`. Yet OCaml accepts $\mathtt{ex}_2$ and rejects $\mathtt{ex}_3$. This is because the **`let`**-binding in $\mathtt{ex}_2$ allows `r` to be treated polymorphically, and thus its type is considered *known*–enabling disambiguation. In $\mathtt{ex}_3$, by contrast, `r` is monomorphic at the point of projection, and disambiguation is therefore forbidden.

To emphasize that this behavior is specification-drive and not an artifact of OCaml's inference algorithm, consider two equivalent versions of $\mathtt{ex}_3$, where `@@` and `|>` are the application and reverse application functions:

```
let ex₃₂ = (fun r → r.x) @@ {x = 1; y = 1}       OCaml  OmniML
let ex₃₃ = {x = 1; y = 1} |> (fun r → r.x)       OCaml  OmniML
```

While these terms are semantically equivalent, they highlight a potential hazard: their typability would vary under a directionally biased inference algorithm, depending on whether the function or argument is typed first. To avoid such implementation-dependent behavior, OCaml chooses to infer all subexpressions *simultaneously*, until they are **`let`**-bound.

---

[1] In fact, OCaml uses a default resolution strategy instead of failing when the type is ambiguous, which is to emit a warning and use the last definition in scope. To check these examples, you should use the options `-principal -w +41`, which enforce principality checks and enables the warning on default resolution.

Consequently, OCaml does not make any difference between $\texttt{ex}_3$, $\texttt{ex}_{32}$, or $\texttt{ex}_{33}$; in all cases, disambiguation is disallowed, and they are ill-typed. This criterion also warns on the following example, where r has a monomorphic type:

```
let ex₄ p r = if p then r.x else (r : α one).x        OCaml  OmniML
```

Warning here is preferable to silently accepting or rejecting the program based on the inference order between the **if** branches.

π-directional inference offers a way to specify and implement principal type inference for fragile features, aligning with the implicit inference order present in most ML-like typecheckers. This mechanism was originally proposed by Garrigue and Rémy [1999] for semi-explicit first-class polymorphism, and used in 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.

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*, which typechecks a term e against a *known* type τ in a given context, and *inference mode* which infers e from the context alone.

For example, the type system designer can decide to typecheck function applications $e_1\ e_2$ by first *inferring* that e_1 has some function type $\tau \longrightarrow \tau'$, and then *checking* e_2 against τ. This is not the only possible choice: bidirectional type inference is a framework that must be instantiated by assigning modes—checking or inference—to each language construct. 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.

Bidirectional type inference has been largely used for languages with higher-rank polymorphism, dependent types, or subtyping. Still, both OCaml and Haskell only use a limited form of bidirectional type checking with an underlying first-order unification-based type inference engine, that limits the downsides of bidirectional type inference.

Limitations of directional type inference. Bidirectional type inference is lightweight, practical, and well-suited for complex language features. 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! For example, the following would be rejected as ambiguous with π-directional type inference alone:

```
let g (f : two → int) : int = f {x = 1; z = 1} in g (fun r → r.x)   OCaml  OmniML
```

OCaml uses π-directional type inference as its primary mechanism, alongside a weak form of bidirectional propagation. In this example the type of the argument of g is known π-directionally, but OCaml then propagates this expected type within the function definition in bidirectional fashion, so that this example may be considered non-ambiguous.

Besides, the implementation of π-directional type inference has an algorithmic cost. For technical reasons, type annotations must unshare types (from acyclic graphs as naturally produced by unification to trees), which may increase the size of types and the cost of type inference. For that reason, the implementation of OCaml cheats and is incomplete by default. The user must explicitly pass the `-principal` flag to require the more expensive computation when desired.

1.2 Omnidirectional type inference

In absence of *implicit* polymorphism, type inference is solely based on unification constraints which can be solved in any order; omnidirectional inference 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 variant $\mathcal{W}$ 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. However, this state of affairs is not a necessity.

To efficiently achieve omnidirectional type inference for fragile ML extensions:

(1) We introduce *suspended match constraints* as a way to suspend ambiguity resolution until sufficient information has been found from context so that they can be discharged.
(2) We work with *partial types schemes*, *i.e.*, with the ability 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.

These 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:

(1) 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*.
(2) Implementing partial types schemes is also hard.

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

Illustrative examples. Examples ex_2 to ex_4 are all typable with omnidirectional inference, as indicated by the green traffic light labeled OmniML—the calculus formalized in this paper.

In contrast, both bidirectional and π-directional inference rely on specifications that include choices that are subjective and somewhat arbitrary. As a result, they reject programs that have a unique well-typed solution. We now turn to further examples that illustrate such failures:

```
let ex₆ r = (r.x, r.y)                                OCaml  OmniML
let ex₇ r = let x = r.x in x + r.y                    OCaml  OmniML
let ex₈ = let getx r = r.x in getx {x = 1; y = 1}     OCaml  OmniML
let ex₉ r = (r.x : bool)                              OCaml  OmniML
let ex₁₀ r = r.x.x                                    OCaml  OmniML
```

All are arguably unambiguous; OCaml accepts none of them, OmniML accepts the first three.

In ex_6, `r` can only be of type α `one`. Indeed, considering the second projection first, we should learn that `r` is of type α `one` and since it is λ-bound, this should then make the first projection unambiguous. Disambiguating this example is a matter of solving the typing constraints in the right order.

A similar failure occurs in ex_7, where the type of the λ-bound variable `r` is initially ambiguous and unknown. It is only upon typing the projection `r.y` that `r` is forced to have the type α `one`; this requires inferring the let-body to disambiguate the let-definition. In ex_8, disambiguation information flows from an instance to back the definition, opposite to the π-order; we call this *backpropagation.*

The example ex_9 can be disambiguated from the return type of the projection, rather than from the type of r. The typing rules for records that we present in this work restrict disambiguation to the record type alone, and thus rejects this example. Alternative typing rules using omnidirectional type inference could support this example as well.

Finally, ex_{10} is an example where none of the field projections has enough type information to be disambiguated on its own, but the constraints they impose can be combined to deduce that the type of r must be one, as the x field of two does not have a record type. This lies outside the framework of omnidirectional type inference, in which suspended constraints must be discharged one by one in some order, independently of other still-suspended constraints. We believe that this restriction is necessary for effective type inference, since the complexity of general overloading without this restriction is NP-hard, even in the absence of let-polymorphism, as shown by an encoding of 3-SAT problem by Charguéraud, Bodin, Dunfield and Riboulet [2025].

Plan. The paper is organized as follows: In §2, we give an overview of suspended constraints and their application to three extensions for ML of various kind. In §3, we describe suspended match constraints and their semantics. In §4, we define OmniML, an extension of ML featuring static overloading of record labels, overloaded tuple projections, and semi-explicit first-class polymorphism. We sketch its typing rules and state the theorems of soundness and completeness for constraint generation, as well as principality. By lack of space, detailed typing rules and constraint generation are postponed to §A. In §5, we provide a formal definition of our constraint solver as a series of non-deterministic rewriting rules and state the main theorems for correctness. In §6, we describe an efficient implementation of suspended constraints and partial type schemes. In §7, we compare with related work, and in §8, we conclude with a discussion of future work, including prototyped extensions whose theory is less clear. Appendix §C contains a complete technical reference, collecting key definitions and figures for convenient lookup. All proofs are postponed to appendices.

Our contributions. Our contributions are: (1) A novel *omnidirectional* type inference framework for extensions of ML with advanced features, based on two new devices, suspended constraints and partial type schemes; (2) A declarative semantics of suspended constraints that captures the idea that they wait on information that must be propagated from the context, not *guessed.* This includes, in particular, a new declarative characterization of *known* type information. (3) A complete yet efficient constraint-solving type inference algorithm. (4) Three instantiation of our framework that give new declarative type systems and their implementation using suspended constraints for tuple projection in the style of SML, static overloading of record fields and datatype constructors, and for semi-explicit first-class polymorphism.

2 Suspended constraints: an overview

The syntax of types and constraints is given in Figure 1. Monotypes (or just types) include, as usual, type variables α, the unit type 1, arrow types, but also[2] structural tuples $\Pi_{i=1}^{n}\tau_i$, nominal types[3] 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 $\mathcal{V}$ the set of type variables.

Building atop the constraint-based type inference framework of Pottier and Rémy [2005], we adopt a constraint language that includes both term and type variables. The language (in Figure 1) contains

[2]These are grayed, as they which will be introduced in the following subsections.

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

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

Fig. 1. Syntax of types and constraints.

tautological (true) and unsatisfiable (false) constraints, conjunctions ($C_1 \wedge C_2$). The constraint form ($\exists \alpha.\, C$) binds an existentially quantified type variable α in C, while the constraint ($\forall \alpha.\, C$) binds α universally. The constraint form ($\tau_1 = \tau_2$) asserts that the types τ_1 and τ_2 are equal. 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 $\exists \bar{\alpha}.\, \tau' = \tau$.

Two constructs deal with the introduction and elimination of constraint abstractions. A constraint abstraction $\lambda \alpha.\, C$ can simply be seen as a function which when applied to some type τ returns $C[\alpha := \tau]$. Constraint abstractions are introduced by a let construct ($\mathsf{let}\ x = \lambda \alpha.\, C_1\ \mathsf{in}\ C_2$) which binds the constraint abstraction to the term variable x in C_2—additionally ensuring the abstraction is satisfiable. They are eliminated using the application constraint ($x\ \tau$) which applies the type τ to the abstraction constraint bound to x.

Finally, we introduce *suspended match constraints* ($\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}$). These constraints are *suspended* until the *shape* of τ, such as 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.

More precisely:

(1) The matchee τ is a type. The constraint remains suspended while τ is a type variable, that is, until the shape of τ is determined.
(2) $\bar{\chi}$ is a list of branches of the form $\rho \to C$, where ρ is a shape pattern. For example, the pattern $\alpha \to \beta$ matches function types, binding its domain and codomain to α and β, respectively. 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.

We keep the grammar of shapes and patterns abstract in this section, to explain the general framework of suspended constraints. For now, it suffices to think of shapes as top-level constructors like function arrows $\cdot \to \cdot$. This will be made more precise in §3.

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 $\mathscr{C}$.

Suspended constraints in action. The remainder of this section illustrates the role of suspended constraints in supporting *fragile* language features as defined above. In particular:

(§2.1) Semi-explicit first-class polymorphism;
(§2.2) Constructor and record label overloading for nominal algebraic datatypes;
(§2.3) Overloaded tuple projection in the style of SML.

We demonstrate how the typability of each of these features can be elaborated 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 is well-typed.

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.

2.1 Semi-explicit first-class polymorphism

Semi-explicit first-class polymorphism [Garrigue and Rémy 1999] brings some System F-like expressiveness to ML by allowing impredicative, first-class polymorphism while preserving principal type inference. It has since been adopted by OCaml, notably for polymorphic object methods.

The type constructor $[\sigma]^{\varepsilon}$ boxes a polymorphic type scheme σ, turning it into a *polytype* annotated with the *annotation variable* ε. Once boxed, the polytype $[\sigma]^{\varepsilon}$ is considered a monotype, thereby enabling impredicative polymorphism. 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, and it captures π-directionality explicitly.

The introduction form for polytypes is a boxing operator $[e : \exists\bar{\alpha}.\, \sigma]$ with an explicit polytype annotation $\exists\bar{\alpha}.\, \sigma$ where the $\bar{\alpha}$ are all the type variables that are free in σ. The resulting expression has type $[\sigma[\bar{\alpha} := \bar{\tau}]]^{\varepsilon}$ where ε is an arbitrary (typically fresh) annotation variable and $\bar{\tau}$ are arbitrary types that replace the free variables $\bar{\alpha}$. The annotation variable ε can thus be generalized. That is $[e : \exists\bar{\alpha}.\, \sigma]$ can also be assigned the type scheme $\forall\varepsilon.\, [\sigma[\bar{\alpha} := \bar{\tau}]]^{\varepsilon}$.

Conversely, to instantiate a polytype expression, 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 $\langle e \rangle$ a type τ that is an instance of 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.

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.

However, type annotations can be used to freshen annotation variables. 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 $\forall\varepsilon_1.\, [\sigma]^{\varepsilon_1}$.

This behavior can be counter-intuitive: type information that has just been inferred must still be considered as yet-unknown until its generalization. It also makes the system sensitive to the placement of type annotations, an artifact of the fixed directionality of generalization in π-directional inference. 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.

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

The difference lies in how generalization and annotation variables interact. In the first term, the annotation occurs in an unboxing operator introducing fresh annotation variables and may therefore be generalized to the type scheme $\forall \varepsilon.\, [\forall \alpha.\, \alpha \to \alpha]^{\varepsilon}$, enabling unboxing to proceed. Whereas the second term applies the annotation to the argument f, which fixes f's type to the monotype $[\forall \alpha.\, \alpha \to \alpha]^{\varepsilon_1}$ for some fresh annotation variable ε_1. Because this type is assigned to f at its binding site, ε_1 is bound in the context when typing $\langle f \rangle$ and cannot be generalized, so the second term is ill-typed despite the annotation.

Suspended match constraints eliminate this sensitivity to directionality when typechecking $\langle e \rangle$. If e is already known to have the type $[\sigma]$, then we can simply instantiate it. 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:

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

The match constraint is suspended until β is resolved to a polytype $[\sigma]$ matching the pattern $[s]$, which binds the type scheme σ to the scheme variable s. The selected branch then performs the instantiation $s \le \alpha$, that is $\sigma \le \alpha$. By waiting for the type of e to be *known*, we ensure principal types without annotation variables.

2.2 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 languages offer statically resolved overloading to avoid the overhead of dynamic dispatch. C++ and Java resolve overloaded functions through compile-time specialization based on argument types. Conversely, languages like Rust and Haskell primarily employ dynamic overloading via traits and type classes, respectively, which can incur runtime overhead unless optimized away by monomorphization and aggressive inlining.

As noted in the introduction, OCaml supports a limited yet useful form of static overloading for record labels and datatype constructors. When encountering overloaded labels or constructors, OCaml resolves ambiguity using local type information, guided by π-directional inference. Nominal types $\mathsf{t}\ \bar{\tau}$ carry annotation variables ε, written $\mathsf{t}^{\varepsilon}\ \bar{\tau}$. As discussed in §2.1, this mechanism allows one to deduce that types polymorphic over their annotation variable $\forall \varepsilon.\, \mathsf{t}^{\varepsilon}\ \bar{\tau}$ are *known*.

Because static overloading involves more intricate flows of information than polytype inference, OCaml supplements π-directionality with a limited, ad-hoc form of bidirectional type inference. This mechanism is folklore; no formal account has been given.

Beyond propagation, OCaml also exploits *closed-world reasoning* to resolve ambiguities in record types. For instance:

```
let ex₁₁ = {x = 42; z = 1337}                                OCaml  OmniML
```

Here, x and y appear together only in the type two, allowing the type checker to unambiguously infer the type of e_{11} as two. 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:

```
let getx r = r.x                                             OCaml  OmniML
```

The expression is compatible with both `one` and `two`, since each defines a field `x`. But `two` 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 $\mathtt{ex}_7$ from §1:

```
let ex₇ r = let x (* infers [two] *) = r.x in x + r.y          OCaml  OmniML
```

Here, OCaml defaults to `two` for `r` when typing `r.x`, but then fails to type `r.y`, as this default choice is fixed—even though `one` would have satisfied both projections.

We assume a global typing environment Ω mapping labels to type schemes, written $\ell : \forall \bar{\alpha}.\, \tau \to \mathrm{t}\ \bar{\tau} \in \Omega$. A given label ℓ may be defined several times in Ω, but at most once at a given record type t. We write $\Omega(\ell/\mathrm{t})$ for the type scheme of ℓ in t when it exists.

We propose an alternative account of static overloading using suspended match constraints. For example, in the case of an ambiguous record projection $e.\ell$, we generate the typing constraint:

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

This constraint suspends resolution of the return type α until the record type β of e is known. Its branch matches against the nominal type pattern $t\ \_$, binding the type constructor name to t. Using this, the appropriate type scheme for ℓ is retrieved from $\Omega(\ell/t)$, instantiated, and the resulting constraints are imposed on the domain and codomain of the field-access type.

OCaml programs that do not use the default rule are accepted by this approach. Certain expressions, such as $\mathtt{e}_{12}$ are well-typed under our account but rejected by OCaml's current type checker.

Our approach also applies to overloaded datatype constructors. Since the formal treatment is analogous to that of record fields, we focus only on fields in this work. However, our prototype implementation of OmniML supports both.

2.3 Tuple projections à la SML

SML supports positional projections from tuples using expressions of the form **#**j `e` to extract the j-th component of the tuple `e`. Internally, tuples in SML are treated as structural records with numeric labels, so (`#`j `e`) desugars into a structural record field access `e.`j: if e has the type $\{j = \tau_j; \varrho\}$, where ϱ is a row describing the remaining tuple fields, then $e.j$ has type τ_j.

SML enforces an additional restriction: the tail ϱ must be fully determined (*i.e.*, it cannot be a polymorphic row variable). This ensures that the arity of the tuple is *known* statically from the surrounding context, thereby avoiding the need for row polymorphism. However, this restriction is not expressed in the typing rules themselves, but is specified operationally as part of the type inference process.

From a typing perspective, tuple projection in SML behaves like a form of static overloading: the expression $e.j$ is valid only when e is known to be an n-ary tuple for some fixed $n \geq j$.

We can capture the typing of tuple projections precisely using suspended constraints. For the projection $e.j$, we generate the following constraint:

$$[\![e.j : \alpha]\!] \;\triangleq\; \exists \beta.\, [\![e : \beta]\!] \wedge \mathsf{match}\ \beta\ \mathsf{with}\ \Pi\ \gamma_j \to \alpha = \gamma$$

The suspended constraint ($\mathsf{match}\ \beta\ \mathsf{with}\ \Pi\ \gamma_j \to \alpha = \gamma$) blocks until the shape of e (β) is known to be a tuple of sufficient arity. The pattern $\Pi\ \gamma_j$ matches only tuple types $\Pi_{i=1}^{n} \tau_i$, where $n \geq j$, binding the j-th component to γ, which is then unified with the expected result type α.

Comparison to SML. Our understanding is that SML typecheckers implement row-polymorphic records under the hood, but they never generalize row variables, rejecting any declaration that leaves a row variable undetermined. This is a neat approach, and we conjecture that it accepts the same programs as our implementation of overloaded tuples using suspended constraints. On the other hand, it only works for structural types. It cannot be applied to disambiguate between nominal records or variants that share field or constructor names, particularly when some field projections or constructors need non-uniform typing rules, such polymorphic fields or GADT constructors.

$$\phi \quad ::= \quad \emptyset \mid \phi[\alpha := \mathfrak{g}] \mid \phi[x := \mathfrak{G}] \qquad\qquad \text{Semantic environments}$$

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

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

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

Fig. 2. Semantics of constraints (without suspended constraints).

3 Semantics of constraints

To implement a typechecker using constraint-based type inference, it suffices to generate constraints from terms and to solve them. To study the meta-theory of this approach, we follow the standard approach of assigning a *semantics* for our constraints—as declaratively as possible. The existence of well-defined declarative semantics provides a foundation for reasoning about correctness and validates the design of the constraint language.

In our work on suspended constraints, defining a satisfying semantics was the most challenging aspect. The key difficulty lies in capturing what it means for type information to be *known*. Our semantics is declarative, but not syntax-directed unlike the standard constraint semantics of Pottier and Rémy [2005]. This lack of syntax-directness complicates reasoning and proofs. On the upside, the semantics directly suggest declarative typing rules for the surface language.

The semantics of constraints follows the standard form of a satisfiability judgment $\phi \vdash C$. The semantic environment ϕ contains a ground assignment for each free variable of C (type and term variable), and $\phi \vdash C$ states that these assignments indeed satisfy C. Let us write $\mathcal{G}$ for the set of *ground* types, types without free variables[4]. ϕ maps each type variable α to a ground types $\mathfrak{g} \in \mathcal{G}$, and each term variable x to sets of ground types $\mathfrak{G} \subseteq \mathcal{G}$ (the set of ground instances of a type scheme for 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.

The judgment is defined in Figure 2 for all constraint formers except suspended constraints; its definition on this fragment is standard and somewhat tautological. The constraint true is satisfied by any environment, and false by none. An environment ϕ satisfies $C_1 \wedge C_2$ if it satisfies both C_1 and C_2. Satisfying $\exists \alpha. C$ requires finding a witness $\mathfrak{g}$ for α. The universal constraint $\forall \alpha. C$ is satisfiable

[4]Ground types are thus finite trees, assuming the existence of some base types such as int. In §8, we discuss the alternative choice of regular trees for the set of ground types that models equirecursive types.

if C is satisfiable for any binding of α. The unification constraint $\tau_1 = \tau_2$ is satisfied when $\phi(\tau_1)$ and $\phi(\tau_2)$ are equal.

The rule for let $x = \lambda\alpha. C_1$ in C_2 states that C_1 must satisfied under *some* instantiation of its bound variable, and that C_2 must be satisfiable when x is bound to $\lambda\alpha. C_1$, or rather to its semantic interpretation as a set of ground types.

An application constraint $x\ \tau$ is interpreted by checking that τ belongs to the set of types mapped to x in ϕ, that is, $\phi(\tau) \in \phi(x)$. Note that when $\phi(x)$ is of the form $\phi'(\lambda\alpha. C)$, 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 $C[\alpha := \tau]$.

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.

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.*

There is an equivalent direct description of principal shapes ς. They are precisely the shapes $\nu\bar{\gamma}. \tau$ satisfying two conditions: (1) $\bar{\gamma}$ must be linear in τ *i.e.*, each variable γ in $\bar{\gamma}$ occurs exactly once in τ. (2) The type τ must be shallow, meaning that its structure is limited in the following way. When τ is not a polytype, all of its subterms must be variables. Shapes of this form are 1, $\gamma_1 \to \gamma_2$, and $\Pi_{i=1}^{n} \gamma_i$, or $\mathsf{t}\ \bar{\gamma}$. When τ is a polytype $[\forall\bar{\alpha}. \sigma']$, the only subterms of σ' that do not contain one of the polymorphic variables $\bar{\alpha}$ must be variables in $\bar{\gamma}$.

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 $\mathrm{shape}\ (\tau)$. For example, $\mathrm{shape}\ (\mathsf{t}\ \bar{\tau})$ 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

We have left the syntax of shape patterns deliberately abstract. We also assume a matching relation:

$$\rho \text{ matches } \varsigma\ \bar{\gamma} = \theta$$

This partial function matches a pattern ρ against a principal shape ς opened with shape names $\bar{\gamma}$ (which must have the same arity as ς), yielding a substitution θ. The substitution binds the pattern variables to shape components, that may contain occurrences of the shape variables $\bar{\gamma}$. For our examples we define the trivial pattern _ which matches any shape and binds nothing:

$$\_ \text{ matches } \varsigma\ \bar{\gamma} \triangleq \emptyset$$

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:

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

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.

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.

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

This rule states that a suspended constraint is satisfied by ϕ whenever the corresponding discharged constraint holds for the canonical shape ς of τ in the semantic environment ϕ. If ς matches no branch in $\bar{\chi}$, then the discharged constraint is not defined, so this rule cannot be applied, and the suspended constraint is unsatisfiable.

This semantics rule is nicely declarative, but unfortunately accepts too many constraints. For example, $\exists\alpha.\,\mathsf{match}\ \alpha\ \mathsf{with}\ \_ \to \alpha = \mathsf{int}$ is satisfiable under this natural semantics:

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

The semantics can *guess* the type of α and use it to unlock the match constraint, rather than requiring it to be *known* from the surrounding context. One could call the guess of $\alpha = \mathsf{int}$ an "out of thin air" behavior. This 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 $\lambda x.\,(x.2)$, which projects the second component of a tuple. The natural semantics lets us guess for x any tuple type of arity at least 2; so there is no principal type for x.

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, Susp-Ctx in Figure 3, is the only non-syntax-directed rule in our semantics. In this rule, the shape ς is not guessed from ϕ, but it must be recovered from the constraint context $\mathscr{C}$. The *unicity* condition $\mathscr{C}[\tau \,!\, \varsigma]$ (defined below) ensures that ς is uniquely determined by $\mathscr{C}$.

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 only uses the rules listed in Figure 2, without using Susp-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: $\forall \phi, \mathfrak{g}.\ \phi \vdash_{\mathsf{simple}} \lfloor \mathscr{C}[\tau = \mathfrak{g}] \rfloor \implies \mathsf{shape}\ (\mathfrak{g}) = \varsigma$.

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

Fig. 3. Semantics of suspended constraints.

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. Implicitly, this induces a linear partial order between the suspended match constraints within a constraint, reflecting a *temporal* dependency: a match constraint may only be discharged once all of its dependencies have been discharged.

The erasure $\lfloor \mathscr{C}[\tau = \mathfrak{g}] \rfloor$ is simple, so the use of $\vdash_{\mathsf{simple}}$ avoids well-foundedness issues that would arise from a negative occurrence of $(\vdash)$ in a premise of Susp-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 interesting cases arise when τ is a type variable and $\mathscr{C}$ is satisfiable.

We summarize the definition of the unicity condition and Susp-Ctx in Figure 3. Together with the rules of Figure 2, this forms the complete semantics of our constraint language.

Example 3.7. Consider the two examples from above:

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

In the first example, we apply the contextual rule with the context $\mathscr{C} := \exists \alpha.\, \alpha = \mathsf{int} \wedge \square$. 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. By contrast, the second example has no contextual information around the suspended constraint: $\mathscr{C} := \square$. So any solution ϕ satisfies it, allowing $\phi(\alpha)$ to have an arbitrary shape (*e.g.* int, bool, *etc.*). As a result, the uniqueness condition $\mathscr{C}[\alpha \,!\, \varsigma]$ never holds and the constraint is unsatisfiable as intended.

Example 3.8. 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 Susp-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 is $\lfloor \mathscr{C} \rfloor = \mathsf{true} \wedge \square \wedge \alpha = \mathsf{int}$. In this constraint β is unconstrained, so for example $\lfloor \mathscr{C}[\beta = \mathsf{int}] \rfloor$ and $\lfloor \mathscr{C}[\beta = \mathsf{bool}] \rfloor$ are both satisfiable: unicity does not hold and Susp-Ctx cannot be applied.

Now consider instead applying Susp-Ctx to the match on α first. To do so, we must show that α has a uniquely determined shape in the context $\mathscr{C}$ equal to $\square \wedge \mathsf{match}\ \beta\ \mathsf{with}\ \_ \to \mathsf{true} \wedge \alpha = \mathsf{int}$. Its erasure $\lfloor \mathscr{C} \rfloor$ is $\square \wedge \mathsf{true} \wedge \alpha = \mathsf{int}$. Since α is unified with int in the erasure, we have $\mathscr{C}[\alpha \,!\, \mathsf{int}]$. We may now discharge the match on α, rewriting it as $(\mathsf{match}\ \alpha := \mathsf{int}\ \mathsf{with}\ \_ \to \beta = \mathsf{bool})$, that is, $(\alpha = \mathsf{int} \wedge \beta = \mathsf{bool})$. Substituting back, we are left to satisfy the constraint $\mathscr{C}[\alpha = \mathsf{int} \wedge \beta = \mathsf{bool}]$, that is, $(\alpha = \mathsf{int} \wedge \beta = \mathsf{bool} \wedge \mathsf{match}\ \beta\ \mathsf{with}\ \_ \to \mathsf{true} \wedge \alpha = \mathsf{int})$.

At this point, we can safely apply Susp-Ctx to the remaining match constraint on β. The unicity condition now holds, as the erasure of the context includes the discharged constraint $\beta = \mathsf{bool}$, allowing us to eliminate the final match constraint.

This 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 3.9. Let us consider a constraint with a cyclic dependency between match constraints:

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

This constraint can be proved satisfiable under the “natural semantics” introduced earlier: by guessing the assignment $\alpha := \mathsf{int}, \beta := \mathsf{bool}$, the two match constraints succeed. However, our solver and the contextual semantics reject it.

Without loss of generality, suppose we attempt to apply Susp-Ctx on α first. We must show $\mathscr{C}[\alpha \,!\, \mathsf{int}]$ where $\mathscr{C}$ is $\square \wedge \mathsf{match}\ \beta\ \mathsf{with}\ \_ \to \alpha = \mathsf{int}$ But the erasure $\lfloor \mathscr{C} \rfloor$ is $\square \wedge \mathsf{true}$ imposes no constraint on α, so unicity fails, and Susp-Ctx cannot be applied.

Example 3.10. Considering the example $\mathtt{ex}_7$ from §1:

```
let ex7 r = let x = r.x in x + r.y                                OCaml OmniML
```

The typing constraint generated for $\mathtt{ex}_7$ contains the following, where α stands for the type of `r`:

$$\exists \alpha, \gamma.\, \mathsf{let}\ x = \lambda\beta.\, (\mathsf{match}\ \alpha\ \mathsf{with}\ \dots)\ \mathsf{in}\ x\ \mathsf{int} \wedge \alpha = \mathsf{one}\ \gamma$$

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

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

```
let ex8 = let getx r = r.x in getx {x = 1; y = 1}                 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 the type scheme is taken, which is only well-typed if `r` has a variable type or a type of the form α `one`. The projection `r.x` would be forbidden if `r` had a variable type, so

$$
\begin{array}{rcll}
e & ::= & x \mid () \mid \lambda x.\, e \mid e_1\ e_2 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid (e : \exists \bar{\alpha}.\, \tau) \mid \{\overline{l = e}\} \mid e.l & \text{Terms} \\
& \mid & (e_1, \ldots, e_n) \mid e.j \mid e.j/n \mid [e] \mid [e : \exists \bar{\alpha}.\, \sigma] \mid \langle e \rangle \mid \langle e : \exists \bar{\alpha}.\, \sigma \rangle & \\
\tau & ::= & \alpha \mid 1 \mid \tau_1 \to \tau_2 \mid \mathsf{t}\ \bar{\tau} \mid \Pi_{i=1}^{n} \tau_i \mid [\sigma] & \text{Types} \\
\sigma & ::= & \tau \mid \forall \alpha.\, \sigma & \text{Type schemes} \\
\Gamma & ::= & \emptyset \mid \Gamma, x : \sigma & \text{Contexts}
\end{array}
$$

Fig. 4. Syntax of OmniML.

α one is the unique solution. We call this flow of information from instances back to definitions *backpropagation*.

The constraint generated when typing $\mathtt{ex}_8$ is:

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

With the context $\mathscr{C}$ equal to let $getx = \lambda \delta.\, \exists \beta, \gamma.\, \delta = \beta \to \gamma \wedge \square$ in $getx\ (\mathsf{int\ one} \to \alpha)$, we can show the unicity predicate $\mathscr{C}[\beta \mathbin{!} \varsigma]$ for the shape $\varsigma = (\nu\gamma.\, \gamma\ \mathsf{one})$. For any $\phi, \mathfrak{g}$, the erasure $\lfloor \mathscr{C}[\beta = \mathfrak{g}] \rfloor$ is let $getx = \lambda \delta.\, \exists \beta, \gamma.\, \delta = \beta \to \gamma \wedge \beta = \mathfrak{g}$ in $getx\ (\mathsf{int\ one} \to \alpha)$. Since $getx$ is bound to the constraint abstraction $\lambda \delta.\, \exists \gamma.\ \delta = (\mathfrak{g} \to \gamma)$, the instantiation $getx\ (\mathsf{int\ one} \to \alpha)$ can only be satisfied when $\mathfrak{g} = \mathsf{int\ one}$. This proves unicity, hence $\mathtt{ex}_7$ is accepted by our semantics.

4 The OmniML calculus

To prove correctness of constraint generation, we must first 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, naïve specifications often fail to ensure principal types.

Take overloaded tuple projections *à la* SML. We can ask the user to provide the length of the tuple explicitly, via an annotated syntax $e.j/n$, which has a simple typing rule (Proj-X).

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

On the other hand, the natural typing rule for the fragile construct $e.j$ breaks principality (Proj-I-Nat). The term $e.j$ admits infinitely many typings for e, provided the tuple is of sufficient length. This is the exact same issue we had with the naïve semantics of suspended constraints, and in fact we solve it in the same way, with a unicity condition and a contextual rule (Proj-I) that transforms the fragile, implicit construct into the robust, explicit counterpart:

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

4.1 Syntax

In Figure 4, we give the grammar for our calculus. Terms include all of the ML calculus: variables x, the unit literal $()$, lambda-abstractions $\lambda x.\, e$, applications $e_1\ e_2$, annotations $(e : \exists \bar{\alpha}.\, \tau)$ and let-bindings let $x = e_1$ in e_2. Our extensions include:

1. Overloaded variant constructors and record labels, modeled using record literals $\{l_1 = e_1; \ldots; l_n = e_n\}$ and field projections $e.l$. Variant constructors are not treated formally in OmniML, but behave analogously in practice.
2. Tuples $(e_1, \ldots, e_n)$ with implicit projections $e.j$ and explicit projections $e.j/n$.

(3) For semi-explicit first-class polymorphism, we have implicit and explicit introduction and elimination forms: boxing $[e]$ and $[e : \exists\bar{\alpha}.\, \sigma]$, and unboxing $\langle e \rangle$ and $\langle e : \exists\bar{\alpha}.\, \sigma\rangle$.

We use the metavariable e^i to range over the fragile/implicit constructions, and e^x to range over their explicit counterpart.

4.2 Typing rules and unicity

We have detailed typing rules for the full OmniML calculus, but unfortunately they do not fit in the margins of the 25 pages of this document. We moved them all, along with detailed examples, in Appendix §A.

Our typing rules $\Gamma \vdash e : \sigma$ are mostly standard, except for the rules governing implicit (or fragile) constructs e^i. These rules are inspired by our contextual constraint semantics (§3): each is a contextual typing rule paired with a unicity condition and an elaboration into an explicit form.

The unicity condition requires that the shape ς is fully determined by the surrounding term context $\mathscr{E}$, including any subexpressions (*e.g.* e in $e.j$). They are analogous to the unicity condition $\mathscr{C}[\tau \,!\, \varsigma]$ for constraints, though the analogy is not exact. Different fragile features require slightly different formulations, depending on whether they infer a unique shape for a subexpression $\mathscr{E}[e \triangleright \varsigma]$ or for the expected type of the context $\mathscr{E}[e \triangleleft \varsigma]$.

In order to define the unicity conditions, we introduce *typed holes* $\{e\}$, which allow any well-typed term e to be treated as if it had any type (via Magic). Types holes are forbidden in the source language—they are a device solely used to define unicity conditions. We also 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 typed holes 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, $\lfloor e.j \rfloor$ is $\{\lfloor e \rfloor\}$. The full definition is given in Appendix §C.

$$\textsc{Magic}\quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \{e\} : \tau'}$$

We can now formalize the two unicity conditions:

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

We use the unicity condition $\mathscr{E}[e \triangleright \varsigma]$ when we disambiguate using the type of a subterm, as in overloaded tuple projections, record projections, and polytype unboxing. Conversely, we use $\mathscr{E}[e \triangleleft \varsigma]$ for polytype boxing and overloaded records, where we disambiguate them using the expected type of the context.

Example 4.1. Let e be let $f = \lambda x.\, x.1$ in $f\ (1, 2)$. e is well-typed using *backpropagation*. e is of the form $\mathscr{E}[x]$ where $\mathscr{E}$ is the context let $f = \lambda x.\, \square$ in $f\ (1, 2)$. We have $\emptyset \vdash \mathscr{E}[x.1/2] : \text{int}$. Let us show that $\mathscr{E}[x \triangleright \nu\gamma_1, \gamma_2.\, \gamma_1 * \gamma_2]$. Assume $\emptyset \vdash \mathscr{E}[\{(x : \mathfrak{g})\}] : \tau$. As $\mathfrak{g}$ is a ground type, the type $\mathfrak{g}$ of x is not a variable. Then, $\mathfrak{g}$ cannot be that of an arbitrary sized tuple, since there is no such type for a tuple of arbitrary size. Hence, $\mathfrak{g}$ must be a tuple $\Pi_{i=1}^{n} \bar{\tau}$ for some size n. Since the codomain of f must be a tuple of size 2 (for $f\ (1, 2)$ to be well-typed), then n must also be 2. This shows that $\mathscr{E}[x \triangleright \nu\gamma_1, \gamma_2.\, \gamma_1 * \gamma_2]$.

4.3 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 4.2 (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 4.3 (Principal types). *For any well-typed closed term e, there exists a type τ, which we call principal, such that: (i) $\vdash e : \tau$. (ii) For any other typing $\vdash e : \tau'$, then $\tau' = \theta(\tau)$ for some substitution θ.*

It is also interesting to discuss the stability of typing by common program transformations.

Application equi-typability does hold. The expressions $f\ e_1\ e_2$ and $\mathsf{swap} f\ e_2\ e_1$ are equitypable where swap is $\lambda f.\, \lambda x_1.\, \lambda x_2.\, f\ \ x_2\ \ e_1$. We also have that $f\ \ e$ and $\mathsf{app}\ \ f\ \ e$ and $\mathsf{rev\_app}\ \ e\ \ f$ are equitypable, where app and rev_app are the application function $\lambda f.\, \lambda x.\, f\ \ x$ and the reverse application function $\lambda x.\, \lambda f.\, f\ x$, respectively. It is well-known that bidirectional types inference breaks application equitypability. Both π-directional and omnidirectional type inference preserve it.

Factorization does not hold. If $\Gamma \vdash e[x := e_0] : \tau$ with x appearing in e, we do not necessarily have $\Gamma \vdash \mathsf{let}\ x = e_0\ \mathsf{in}\ e : \tau$. This is not a defect of our system, but a general property of all systems that support static overloading: the expanded term $e[x := e_0]$ can pick a different overloading choice for each occurrence of e_0, and if they are incompatible the factored form may not typecheck.

Inlining does not hold. If $\Gamma \vdash \mathsf{let}\ x = e_0\ \mathsf{in}\ e : \tau$, we do not necessarily have $\Gamma \vdash e[x := e_0] : \tau$. This is specific to our support for *backpropagation*: the let-form will use information from all occurrences of x in e to resolve fragile constructs in e_0, but in the inlined form each copy of e must resolve its implicit constructs independently, and it has access to less information to establish unicity. As a result, the implicit OmniML calculus does not preserve typability in its operational semantics.

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.

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) head variables do not occur in multiple equations; (3) the constraint is acyclic.

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.

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

Fig. 5. Syntax and semantics of unification problems.

Unification problems U (Figure 5) 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.

Our algorithm is largely standard, with it 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]. For a detailed discussion of the unification rules, see §B (appendix).

5.2 Solving rules

We now gradually introduce the rules of the constraint solver itself (Figures 6, 8 and 10). 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.

$$\textsc{S-Unif}\quad \frac{U_1 \qquad U_1 \longrightarrow U_2}{U_2}\longrightarrow \qquad \textsc{S-False}\quad \frac{\mathscr{C}[\mathsf{false}] \qquad \mathscr{C} \neq \square}{\mathsf{false}}\longrightarrow$$

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

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

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

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

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

Fig. 6. Basic rewriting rules $C_1 \longrightarrow C_2$

Basic rules. S-Unif 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.

$$C \quad ::= \quad \ldots \mid \mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = C_1\ \mathsf{in}\ C_2 \qquad\qquad \text{Constraints}$$

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

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

Fig. 7. Syntax and semantics of region-based let constraints.

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 7. 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:

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

Rule 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 instantiations from §2.1 and §2.2, 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.

S-Match-Type

$$\frac{\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi} \qquad \tau \notin \mathcal{V}}{\mathsf{match}\ \tau := \mathsf{shape}\ (\tau)\ \mathsf{with}\ \bar{\chi}}$$

S-Match-Var

$$\frac{\mathscr{C}[\mathsf{match}\ \alpha\ \mathsf{with}\ \bar{\chi}] \qquad \alpha = \tau = \epsilon \in \mathscr{C}}{\mathscr{C}[\mathsf{match}\ \alpha := \mathsf{shape}\ (\tau)\ \mathsf{with}\ \bar{\chi}]}$$

Fig. 8. Rewriting rules for suspended match constraints.

Suspended match constraints. S-Match-Type solves suspended match constraints whose scrutinee is a non-variable type τ by rewriting them using the sugar ($\mathsf{match}\ \tau := \mathsf{shape}\ (\tau)\ \mathsf{with}\ \bar{\chi}$), introduced in §2.

S-Match-Var applies when the scrutinee is a variable α and the context $\mathscr{C}$ proves that α is equal to some non-variable type τ, which establishes the unicity property $\mathscr{C}[\tau\ !\ \mathsf{shape}\ (\tau)]$. To check whether a context $\mathscr{C}$ proves an equality—or more generally, a multi-equation ϵ—we search for a decomposition $\mathscr{C} = \mathscr{C}_1[\epsilon \wedge \mathscr{C}_2]$ where $\mathsf{fv}(\epsilon)$ is disjoint from the binders of $\mathscr{C}_2$.

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

S-Let-App-Beta

$$\frac{\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = C_1\ \mathsf{in}\ \mathscr{C}[x\ \tau] \qquad \alpha, \bar{\alpha} \# \tau \qquad x \# \mathsf{bv}(\mathscr{C})}{\mathsf{let}\ x = \lambda\alpha.\, C_1\ \mathsf{in}\ \mathscr{C}[\exists \alpha, \bar{\alpha}.\, \alpha = \tau \wedge C_1]}$$

This 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 ML generalization and instantiation, a connection formalized by Pottier and Rémy [2005], where $\lambda\alpha[\bar{\alpha}].\, \bar{\epsilon}$ corresponds to the type scheme $\forall\bar{\alpha}.\, \vartheta(\alpha)$ and ϑ is the *mgu* of $\bar{\epsilon}$. This optimization underlies efficient implementations of HM inference, such as OCaml's.

However, this approach *does not* extend to suspended constraints. To illustrate this, let us examine ex_7 (from §1):

```
let ex₇ r = let x = r.x in x + r.y
```
OCaml OmniML

The generated typing constraint contains:

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

where $\mathscr{C}[t, \alpha, \beta]$ is $\Omega(\ell/t) \leq \alpha \rightarrow \beta$. Here, α stands for `r`'s type. The constraint remains suspended until `r.y` forces `r`'s type to be one. Crucially, the variable β (introduced inside the abstraction for

$$C \quad ::= \quad \ldots \mid \exists i^x.\, C \mid i[\alpha \rightsquigarrow \tau] \qquad \text{Constraints}$$
$$\phi \quad ::= \quad \ldots \mid \phi[i := \phi'] \qquad \text{Semantic environments}$$

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

Fig. 9. The syntax and semantics of partial instantiations.

the type of y) 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 naïvely 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 *partial instantiation constraints*. 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 partial instantiations $i[\alpha \rightsquigarrow \tau]$ are solved uniformly. Within the solver, we view partial instantiations as markers indicating which parts of the abstraction still need to be copied.

Partial instantiations enables efficient incremental instantiation of constraint abstractions: solved parts are reused immediately, while suspended constraints can be solved later, further refining the abstraction and propagation additional equations to the application sites.

The semantics of the existential constraint $\exists i^x.\, C$ (Exists-Inst) introduces the fresh instantiation i by "guessing" a region ϕ' that satisfies the regional constraint abstraction bound to x. Partial instantiations (Partial-Inst) equate the copy of α in i with τ. The domain of partial 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 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 partial instantiations of x via renamings—and only there. Hence, they cannot appear in C_2 when the corresponding variable x does not itself appear in C_2.

Partial instantiation constraints are reduced using the following rules:

(1) S-Inst-Copy copies the shape of a type to the instantiation site, introducing fresh variables for each subcomponents and marking them with corresponding instantiation constraints. We write $i^x[\alpha \rightsquigarrow \tau]$ as a shorthand for $i[\alpha \rightsquigarrow \tau]$ 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 refer to the same source variable.

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 remove 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

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

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

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

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

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

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

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

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

$$\textsc{S-BackProp}\quad \frac{\begin{array}{c}\mathscr{C}[\text{let } x\ \alpha\ [\bar{\alpha}] = \mathscr{C}_1[\text{match } \alpha' \text{ with } \bar{\chi}] \text{ in } \mathscr{C}_2[i^x[\alpha' \leadsto \gamma]]] \\ \alpha' \in \alpha, \bar{\alpha} \qquad \alpha' = \tau = \epsilon \in \mathscr{C}[\mathscr{C}_2] \qquad x \# \text{bv}(\mathscr{C}_2)\end{array}}{\mathscr{C}[\text{let } x\ \alpha\ [\bar{\alpha}] = \mathscr{C}_1[\text{match } \alpha' := \text{shape }(\tau) \text{ with } \bar{\chi}] \text{ in } \mathscr{C}_2[i^x[\alpha' \leadsto \gamma]]]}$$

Fig. 10. Select rewriting rules for let-bindings and applications.

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.

To decide when $\exists \bar{\alpha}.\, C$ determines $\bar{\beta}$ holds for $\bar{\beta} \# \bar{\alpha}$, we search for a multi-equation ϵ in C of the form: (1) $\gamma = \epsilon'$ where $\gamma \# \bar{\alpha}, \bar{\beta}$ and $\bar{\beta} \subseteq \text{fv}(\epsilon')$, or (2) $\bar{\beta} = \tau = \epsilon'$ where $\text{fv}(\tau) \# \bar{\alpha}, \bar{\beta}$. For instance, $\exists \beta_1.\, \alpha = \beta_1 \rightarrow \beta_2$ determines β_2, as β_2 is free.

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.

Backpropagation. Finally, S-BackProp expresses *backpropagation*, previously illustrated in Example 3.11. In particular, the shape of a regional variable can sometimes be determined from its instantiations. If an abstraction contains a suspended match constraint on a regional variable α', and the constraint context includes a partial instantiation $i^x[\alpha' \rightsquigarrow \gamma]$ together with a multi-equation constraining the copy of α' (γ) to a non-variable type τ, then shape (τ) must be the unique shape of α'. Any other shape would render the instantiation unsatisfiable.

Theorem 5.3 (Progress). *If constraint C cannot take a step $C \longrightarrow C'$, then either:*

(i) C is solved.

(ii) C is stuck, it is either: (a) false*; (b) $\hat{\mathscr{C}}[x\ \tau]$ where $x \# \hat{\mathscr{C}}$; (c) $\hat{\mathscr{C}}[i^x[\alpha \rightsquigarrow \gamma]]$ where $x \# \hat{\mathscr{C}}$ and $i.\alpha \# \mathsf{insts}(\hat{\mathscr{C}})$; (d) for every match constraint $\hat{\mathscr{C}}[\mathsf{match}\ \alpha\ \mathsf{with}\ \bar{\chi}]$ 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.*

Theorem 5.4 (Termination). *The constraint solver terminates on all inputs.*

Theorem 5.5 (Preservation). *If $C_1 \longrightarrow C_2$, then $C_1 \equiv C_2$.*

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[5]. It is public and open-source (link omitted for anonymity). 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.

[5]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.

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 partial 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.

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

Principality tracking in OCaml. Garrigue and Rémy [1999] introduced an approach to principality tracking for polytypes—what we now call π-directional inference—in which generalization and instantiation govern the flow of known type information. This approach has since been extended to other features of the OCaml language: whenever the typechecker need to know if a type is known in a robust way, it checks whether the type is generalizable. We compared their approach to ours in §1 and §2.1.

Bidirectional type inference. At the level of simply-typed terms or ML, we believe that omnidirectional inference works better than bidirectional type inference: it can type more programs than a given fixed bidirectional system, and has a more declarative specification—we would say that it is "more principal". In fact, a direct inspiration for the present work was a user complaint in Rossberg [2016] on the type-based disambiguation of OCaml: its bidirectional logic propagates type information from patterns to definitions in `let`-bindings, when the WebAssembly reference implementation would sometimes prefer the other direction. On the other hand, bidirectional typing is known to scale to powerful systems such as fully-implicit predicative polymorphism [Dunfield and Krishnaswami 2013] and we have not considered scaling our approach to those systems yet.

Qualified types. Qualified types [Jones 1995], most well-known via their usage in Haskell type-classes, are related to our suspended match constraints as they represent constraints on types or type variables. At generalization time, the constraints on generalizable variables are kept as part of the generalized type scheme, and they get copied during instantiation. This is much simpler to implement than our partial type schemes, but it provides a different behavior where each instance can choose independently how to resolve the constraint. Qualified types are an excellent choice when this is the desired behavior, typically for dynamic overloading. To handle cases that require a unique resolution of the constraint across all instances—such as static overloading—we require the more complex mechanism of partial generalization.

Suspended constraints in dependent-type systems. Suspending the 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. The novelty of our work lies in combining constraint suspension with ML-style implicit polymorphism—absent from most dependently-typed systems—and in the design of a declarative constraint semantics used to establish principality.

OutsideIn. 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. They do not give a declarative semantics for delayed implication that their solver preserves. Moreover, 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. We believe that we have solved the troubling interactions between let-generalization and suspended constraints in this work, and would be interested in studying applications to GADT typing, which was also one of our original motivations.

8 Conclusions and future work

In this work, we developed a constraint-based framework for omnidirectional type inference, capable of supporting fragile features that would otherwise break principality. Central to our approach is a new declarative account of when type inference is *known* from the context, rather than *guessed*.

Our constraint solver is omnidirectional: constraints may be solved in any order, made possible by our introduction of *partial type schemes*. We formalized the solver as a non-deterministic, terminating rewrite system, and implemented an efficient prototype to demonstrate its practicality.

Through three instantiations of our framework—static overloading of tuples, nominal records and variant constructors, and semi-explicit first-class polymorphism—we showed that our framework yields a sound and complete inference algorithm and a principal type system. In short, it appears principality holds *anyway* from our approach. Naturally, all this begs the question: what else can be done with omnidirectional inference beyond the features of OmniML?

Static overloading. Our nominal records use a restricted form of overloading. We have also experimented with a more general overloading mechanism in which several definitions may be bound to the same identifier $M.x$, but prefixed with a namespace[6] M used for disambiguation: an

[6] Reusing the notation of Leijen and Ye [2025].

implicit form x in the source is elaborated to an explicit form $M.x$. Although implemented in a prototype, we have not yet formalized this feature. Nevertheless, we conjecture that it should be typable with our framework.

Modular implicits [White, Bour and Yallop 2014] are a proposed extension to OCaml's module system, intended to support ad-hoc polymorphism through type-directed implicit parameters. We believe omnidirectional type inference could serve as a principled, constraint-based approach foundation for resolving implicits in the presence of let-generalization. As future work, we aim to extend our constraint language to typecheck an implicit-parameters calculus, similar to COCHIS [Schrijvers, d. S. Oliveira, Wadler and Marntirosian 2019], but with ML polymorphism.

Higher-rank polymorphism. In §1 and §7, we compared omnidirectional and bidirectional type inference in the context of static overloading. While overloading is non-trivial, it poses little challenge for the bidirectional framework, making the comparison somewhat limited. Bidirectional typing is best known for its scalability to more complex settings, such as higher-rank polymorphism. We are therefore interested in extending our framework to support higher-rank polymorphism, in the style of Dunfield and Krishnaswami [2013]. This would provide a more meaningful basis for understanding the trade-offs of omnidirectional and bidirectional inference.

ML^F is an extension of ML that support first-class polymorphism that goes beyond the power of System F, while retaining type inference. It is a generalization of OCaml's polytypes, relying on π-directionality. It would therefore be worth exploring whether omnidirectional type inference could further empower ML^F.

Default rules. Some type systems disambiguate fragile constructs using known type information, but fall back on default, non-principal choices when none is available. OCaml selects the most recent matching record type in scope for ambiguous field names; SML assigns default types to overloaded numeric literals [Rossberg 2008, Section 5.8].

We explored adding such *default rules* to suspended match constraints, allowing unresolved constraints to discharge with a default shape rather than fail. While pragmatic, such rules are inherently non-principal and difficult to reconcile with our framework.

In particular, they introduce subtle semantic complexities: if two suspended constraints could unlock each other, then defaulting one over the other may force an unsatisfiable branch to be taken. The optimal or principled strategy for applying defaults in such cases remains unclear. Should we fire all default rules of all suspended constraints that remain after the solver terminates, or in batches, restricted to connected components of mutually dependent suspended constraints? Our prototype opts for the latter, but this warrants further study.

Equi-recursive types. OCaml allows equi-recursive types to express recursive polymorphic variants and objects types. Supporting such types is a necessary step towards integrating our approach into OCaml's typechecker.

In this work, for the sake of simplicity, we treat ground types as finite trees. Supporting equi-recursive types amounts to using regular trees instead [Pottier and Rémy 2005]. Our prototype already supports them, but the formalization of our solver relies on acyclicity to ensure termination. Extending the formalization to accommodate cycles would require some changes. Following the implementation, incremental instantiation might require to instantiate cycles atomically.

Shapes may also be equi-recursive, though only minimal shapes of polytypes can be recursive. In the acyclic setting, shape equality is syntactic; with cycles, this no longer holds—but we do not anticipate any fundamental issues.

References

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

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

Jana Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ICFP*.

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.

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

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

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, 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

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

FranCois 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/.

FranCcois 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. 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.

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

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

Tom Schrijvers, Bruno C. d. S. Oliveira, Philip Wadler, and Koar Marntirosian. 2019. COCHIS: Stable and coherent implicits. *J. Funct. Program.* 29 (2019), e3. https://doi.org/10.1017/S0956796818000242

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

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

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

Organization of our appendices

Content appendices. These appendices are intended to be readable prose in the style of the rest of the paper.

- §A presents and explains the typing rules for OmniML; this is the long version of §4.
- §B presents a relatively standard part of our constraint solver (§5), namely the unification rules.

Reference appendix.

- §C 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.

- §D proves properties of the constraint language and its semantics. The main result is canonicalization, which morally establishes that uses of the contextual rule Susp-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.
- §E proves the correctness of the constraint solver with respect to the semantics.
- §F proves the properties about the OmniML type system, in particular the correctness of constraint generation.

A The OmniML calculus: typing rules and constraint generation

A.1 Typing rules

As usual, the main typing judgment $\Gamma \vdash e : \sigma$ states that in context Γ, expression e has type scheme σ. Typing rules are given on Figure 11. They use auxiliary typing judgments $\Gamma \vdash \ell = e : \tau$ and $l : \tau \to \tau'$ for the typing of record assignments and label instantiations respectively.

Var retrieves the type scheme $x : \sigma$ from the context Γ. Function types are introduced via lambda abstractions: in Fun, the system guesses a well-formed type τ_1 for the type of x, typechecks the body e is under the extended context $\Gamma, x : \tau_1$ producing the return type τ_2, and assigns the abstraction the function type $\tau_1 \to \tau_2$. Conversely, function types are eliminated by applications; in Rule App, the type of the argument must match the function's parameter type τ_1 and application returns the type τ_2. Unit asserts that () has the unit type 1.

Gen and Inst correspond to implicit *generalization* and *instantiation* respectively. Generalization universally quantifies a type variable α, introducing it as a fresh polymorphic variable in the typing context. In Inst, we specialize a type scheme $\forall\alpha.\,\sigma$ to $\sigma[\alpha := \tau]$, substituting α for an arbitrary monotype τ.

Let-polymorphism is handled by the Let rule, where a *polymorphic* term can be bound. This allows a single definition to be instantiated differently at each use site—an essential feature of ML. In this rule, the term e_1 has a polymorphic type scheme σ, adds $x : \sigma$ into the context Γ to typecheck e_2.

Annotations $(e : \exists\bar{\alpha}.\,\tau)$ 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 that $\bar{\alpha}$ may be unified with some types $\bar{\tau}$ to produce a well-typed term. For instance, the term $(\lambda x.\,x+1 : \exists\alpha.\,\alpha \to \alpha)$ is well-typed with $\alpha := \mathsf{int}$ in Annot.

Polytypes and overloaded tuples. The typing rules for fully annotated terms (e^x) are unsurprising. However, typing rules for terms with omitted type annotations are non-compositional as they depend on a surrounding one-hole context $\mathscr{E}$. Hence, they assert that the typability of the expression

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

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

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

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

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

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

$$\frac{(\Gamma \vdash l_i = e_i : \tau)_{i=1}^{n} \qquad \bar{l} \,!\, \tau}{\Gamma \vdash \{l_1 = e_1; \ldots; l_n = e_n\} : \tau}\ \textsc{Rcd} \qquad \frac{\Gamma \vdash e : \tau' \qquad l : \tau \longrightarrow \tau' \qquad l \,!\, \tau}{\Gamma \vdash e.l : \tau}\ \textsc{Rcd-Proj} \qquad \frac{\mathscr{L}[\ell \,!\, \mathrm{t}] \qquad \Gamma \vdash \mathscr{L}[\ell/\mathrm{t}] : \tau}{\Gamma \vdash \mathscr{L}[\ell] : \tau}\ \textsc{Lab-I}$$

$$\frac{\Omega(\ell/\mathrm{t}) = \forall \bar{\alpha}.\, \tau \to \mathrm{t}\ \bar{\alpha}}{\ell/\mathrm{t} \colon \bar{\tau}[\bar{\alpha} := \bar{\tau}] \to \mathrm{t}\ \bar{\tau}}\ \textsc{Lab-X} \qquad \frac{\mathscr{L}[\ell \,!\, \mathrm{t}] \qquad \Gamma \vdash \mathscr{L}[\ell/\mathrm{t}] : \tau}{\Gamma \vdash \mathscr{L}[\ell] : \tau}\ \textsc{Lab-I} \qquad \frac{\bar{l} \,!\, \mathrm{t} \in \Omega}{\bar{l} \,!\, \mathrm{t}\bar{\tau}}\ \textsc{Lab-!} \qquad \textsc{Lab-?}\ \ \bar{l} \,!\, \tau$$

Fig. 11. Typing rules of OmniML.

$\Gamma \vdash \mathscr{E}[e^i] : \tau$ where e^i is an expression with an implicit type annotation. We first request a typing for the expression with an explicit annotation $\Gamma \vdash \mathscr{E}[e^x] : \tau$ where e^x is a fully annotated variant of e^i. We then request that (the shape of) the annotation is fully determined from context, either from the type of the expression, which we write $\mathscr{E}[e \triangleright \varsigma]$, or from the type of the hole, which we write $\mathscr{E}[e \triangleleft \varsigma]$.

In order to describe the judgments $\mathscr{E}[e \triangleright \varsigma]$ and $\mathscr{E}[e \triangleleft \varsigma]$, we introduce a *typed hole* construct $\{e\}$ that allows any well-typed expression e to be treated as if it had any type. That is the typing rule for holes is:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \{e\} : \tau'}\ \textsc{Magic}$$

Typed holes are not allowed on source terms and are just a device for the definition of non-ambiguous shapes. Finally, we define what it means for a shape to be determined from the type of a context or an expression:

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

These states that the shape ς of expression e in context $\mathscr{E}$ is determined by the expression e, in the former case, or by the context $\mathscr{E}$ in the latter case. Just like constraints, we must erase implicit constructs in the term that have not yet been elaborated, written $\lfloor e \rfloor$ (defined in §C).

The implicit rule Proj-I types the projection $e.j$ provided the context $\mathscr{E}$ *infers* that the shape of e must be a tuple with arity n. Similarly, Use-I permits instantiating a polytype in $\langle e \rangle$ if the context $\mathscr{E}$ infers that the type of e must be a polytype with shape $\nu\bar{\gamma}.\,[\sigma]$. The rule Poly-I types the implicit boxing construct $[e]$ by *checking* the expected type of $[e]$ in the context $\mathscr{E}$ is a polytype with the shape $\nu\bar{\gamma}.\,[\sigma]$. This rule differs from the previous two as the shape is determined by the expected type within the context as opposed to the inferred type of e.

Overloaded record labels. We adopt a similar non-compositional approach for elaborating overloaded labels, whether in record projection ($e.\ell$) or record construction ($\{\overline{\ell = e}\}$), although the definitions are slightly more involved. Here, a one-hole label context $\mathscr{L}$ provides the surrounding context in which a label ℓ may appear:

$$\mathscr{L} \quad ::= \quad \mathscr{E}[e.\square] \mid \mathscr{E}[\{l_1 = e_1; \ldots; \square = e_i; \ldots; l_n = e_n\}] \qquad \text{Label contexts}$$

As with our contextual rules for expressions, we define two rules for labels. Lab-X handles explicitly annotated labels ℓ/t by instantiating the type scheme $\forall\bar{\alpha}.\,\tau \to \mathrm{t}\ \bar{\alpha}$ associated with ℓ in label environment Ω. Lab-I handles unannotated labels by elaborating ℓ to ℓ/t if the context $\mathscr{L}$ uniquely infers the record type t for ℓ and the resulting elaboration is well-typed.

The unicity of the inferred record type is captured by the judgment $\mathscr{L}[\ell\,!\,\mathrm{t}]$. The definition fits into the framework we established for expressions above by introducing analogous annotation and hole constructs for labels.

$$\textsc{Lab-Magic}\ \frac{}{\Gamma \vdash \{\ell\} : \tau' \to \tau} \qquad\qquad \textsc{Lab-Annot}\ \frac{\Gamma \vdash l : \tau' \to \tau[\bar{\alpha} := \bar{\tau}]}{\Gamma \vdash (l : \exists\bar{\alpha}.\,\tau) : \tau' \to \tau[\bar{\alpha} := \bar{\tau}]}$$

$$\mathscr{L}[\ell\,!\,\mathrm{t}] \ \triangleq\ \forall \Gamma, \tau, \mathfrak{g},\ \Gamma \vdash \lfloor \mathscr{L}[(\{\ell\} : \mathfrak{g})] \rfloor : \tau \implies \text{shape}\ (\mathfrak{g}) = \nu\bar{\gamma}.\,\bar{\gamma}\mathrm{t}$$

Rcd types a record $\{\overline{l = e}\}$ as a record type τ provided that each field assignment $l = e$ can be assigned the record type τ. Rcd-Assn checks that e has the appropriate field type in $l = e$ and returns the instantiated record type τ' for the label l. Rcd-Proj types the projection $e.l$ by checking that the type of e matches the record type associated with label l, returning the field type τ.

Both Rcd and Rcd-Proj impose additional constraints on their record types to support *closed-world* reasoning. These constraints exploit the uniqueness of type definitions in the global label environment Ω to resolve overloaded labels: (1) in a record projection $e.\ell$, if the label ℓ is not overloaded, then the global record typing context Ω assigns a unique record type t to ℓ; (2) in a record expression $\{l_1 = e_1; \ldots; l_n = e_n\}$, if the set of labels $l_1, \ldots, l_n$ uniquely identifies a record type t in the typing context Ω, then we can assign this type to the record expression.

We formalize this with the judgment $\bar{l}\,!\,\tau$, which either: (1) enforces τ to be of the form $\mathrm{t}\ \bar{\tau}$ if the labels $\bar{l}$ uniquely identify a nominal record type t in Ω (Lab-!), or (2) imposes no constraint on τ in the ambiguous case (Lab-?).

Label declarations in Ω have the form $\ell : \forall \bar{\alpha}.\, \tau' \to \mathsf{t}\ \bar{\alpha}$, assigning labels to field types and record types[7]. We write $\ell/\mathsf{t} \in \Omega$ if such a declaration of ℓ exists for the record type t. This membership relation extends to explicitly annotated and casted labels:

$$\textsc{Lab-inX}\ \frac{\ell/\mathsf{t} \in \Omega}{(\ell/\mathsf{t})/\mathsf{t} \in \Omega} \qquad \textsc{Lab-inMagic}\ \frac{\ell/\mathsf{t} \in \Omega}{\{\ell\}/\mathsf{t} \in \Omega} \qquad \textsc{Lab-inAnnot}\ \frac{l/\mathsf{t} \in \Omega}{(l : \exists \bar{\alpha}.\, \tau)/\tau \in \Omega}$$

We then define the uniqueness predicate $\bar{l}\,!\,\mathsf{t} \in \Omega$ as:

$$\textsc{Lab-U}\ \frac{\bar{l}/\mathsf{t} \in \Omega \qquad \forall \mathsf{t}'\,.\ \bar{l}/\mathsf{t}' \in \Omega \implies \mathsf{t} = \mathsf{t}'}{\bar{l}\,!\,\mathsf{t} \in \Omega}$$

This states that the set of labels $\bar{l}$ determines a unique nominal type t in Ω if no other type t' can be associated with the same label set.

A.2 Examples of typings

The following lemma shows that we can always take a larger context $\mathscr{E}$ or $\mathscr{L}$ for implicit rules Proj-I, Use-I, Poly-I and Lab-I. That is, there is always a derivation using only toplevel contexts.

Lemma A.1. *If $\mathscr{E}_2[e \triangleright \varsigma]$, then $(\mathscr{E}_1[\mathscr{E}_2])[e \triangleright \varsigma]$. Similarly, for label contexts, if $\mathscr{L}[\ell\,!\,\mathsf{t}]$, then $(\mathscr{E}[\mathscr{L}])[\ell\,!\,\mathsf{t}]$.*

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

Example A.2. To illustrate a simple case of non-typability, we show that the expression e equal to $\lambda x.\, x.k$ is ambiguous, *i.e.*, that it does not typecheck. If there is a derivation of $\lambda x.\, x.k$ then there must be one of the form:

$$\frac{\mathscr{E}[x \triangleright \nu \bar{\gamma}.\, \Pi_{i=1}^{n} \bar{\gamma}] \qquad \emptyset \vdash \mathscr{E}[x.k/n] : \tau}{\emptyset \vdash \mathscr{E}[x.k] : \tau}\ \textsc{Proj-I}$$

where E is the term $\lambda x.\, \square$, which is the largest possible context, thanks to Lemma A.1. Let τ be $\Pi_{i=1}^{n} \tau_i \to \tau_k$ for some $n \geq k$. We have the following derivation:

$$\frac{\dfrac{x : \Pi_{i=1}^{n} \tau_i \vdash x : \Pi_{i=1}^{n} \tau_i}{x : \Pi_{i=1}^{n} \tau_i \vdash x.k/n : \tau_k}\ \textsc{Proj-X}}{\emptyset \vdash \mathscr{E}[x.k/n] : \tau}\ \textsc{Fun}$$

Unfortunately, $\mathscr{E}[x \triangleright \nu \bar{\gamma}.\, \Pi_{i=1}^{n} \bar{\gamma}]$ does not hold. Indeed, we have $\emptyset \vdash \mathscr{E}[\{(x : \mathfrak{g})\}] : \tau$ for any $\mathfrak{g}$ assuming τ is of the form $\mathfrak{g} \to \tau'$. Hence, $\nu \bar{\gamma}.\, \Pi_{i=1}^{n} \bar{\gamma}$ and $\nu \bar{\gamma}.\, \Pi_{i=1}^{n+1} \bar{\gamma}$ are two possible shapes for the type of x.

Example A.3. We now illustrate a non-ambiguous example, showing that the expression e equal to $\vdash (\lambda x.\, x.1)\ (1, 2) : \mathsf{int}$. Let $\mathscr{E}$ be the context $(\lambda x.\, \square)\ (1, 2)$. We have the derivation:

$$\frac{\mathscr{E}[x \triangleright \nu \gamma_1, \gamma_2.\, \gamma_1 * \gamma_2] \qquad \emptyset \vdash \mathscr{E}[x.1/2] : \mathsf{int}}{\emptyset \vdash \mathscr{E}[x.1] : \mathsf{int}}\ \textsc{Proj-I}$$

We have $\emptyset \vdash \mathscr{E}[x.1/2] : \mathsf{int}$, indeed. Therefore, it just remains to show $\mathscr{E}[x \triangleright \nu \gamma_1, \gamma_2.\, \gamma_1 * \gamma_2]$ (**1**). Assume $\emptyset \vdash \mathscr{E}[\{(x : \mathfrak{g})\}] : \tau$. Since $x : \mathsf{int} * \mathsf{int}$ is bound in the context at the hole in $\mathscr{E}$, there is no other choice but take $\mathfrak{g}$ equal to $\mathsf{int} * \mathsf{int}$, hence shape $(\mathfrak{g}) = \nu \gamma_1, \gamma_2.\, \gamma_1 * \gamma_2$, which proves (1).

[7] For a given record type t, we assume each label associated with it is unique.

The following example of non-typability illustrates how the typing rules still forces to reject typing of some expressions whose elaboration would be unambiguous. This is intended, to prevent us from having to focus at several terms simultaneously. Our typing rules enforce the resolution of shape inference, locally, one at a time.

Example A.4. Let τ_{id} be $[\forall\alpha.\,\alpha \to \alpha]$. We show that the expression e equal to $\mathsf{let}\ x = [\lambda z.\,z]\ \mathsf{in}\ (\langle x\rangle\ 1, \langle x\rangle\ ())$ is rejected as ambiguous. Let τ_{id} be $[\forall\alpha.\,\alpha \to \alpha]$. Clearly, we have $\mathsf{let}\ x = [\lambda z.\,z : \tau_{\mathrm{id}}]\ \mathsf{in}\ (\langle x : \tau_{\mathrm{id}}\rangle\ 1, \langle x : \tau_{\mathrm{id}}\rangle\ ())$. This is actually the only possible fully annotated derivation. To show that e is typable, we must be able to make all annotations optional, sequentially. Therefore, the final step, which will eliminate the last annotation has a single point of focus of the form $\mathscr{E}[e^i]$, where e^i can be any of the three positions with a missing annotation. We consider each case independently, and show that it is actually not typable.

Case $\mathscr{E}$ *is* $\mathsf{let}\ x = \square\ \mathsf{in}\ (\langle x\rangle\ 1, \langle x\rangle\ ())$. If this holds, we should have a derivation that ends with

$$\frac{\mathscr{E}[\lambda z.\,z \triangleleft [\tau_{\mathrm{id}}]] \qquad \emptyset \vdash \mathscr{E}[[\lambda z.\,z : \tau_{\mathrm{id}}]] : \tau}{\emptyset \vdash \mathscr{E}[[\lambda z.\,z]] : \tau}\ \textsc{Poly-I}$$

However, $\mathscr{E}[\lambda z.\,z \triangleleft [\tau_{\mathrm{id}}]]$ does not hold. Indeed, the following judgment $\emptyset \vdash \mathscr{E}[(\{\lambda z.\,z\} : [\sigma])] : \tau$ holds, where σ is either $\forall\alpha.\,\alpha \to \alpha$ or $\forall\alpha.\,\alpha \to \alpha \to \alpha$. Hence, the shape of the type of $\lambda z.\,z$ is not uniquely determined and this case cannot occur.

Case $\mathscr{E}$ *is* $\mathsf{let}\ x = [\lambda z.\,z]\ \mathsf{in}\ \langle\square\rangle\ 1, \langle x\rangle\ ()$. The derivation must end with:

$$\frac{\mathscr{E}[x \triangleright [\tau_{\mathrm{id}}]] \qquad \emptyset \vdash \mathscr{E}[\langle x : \tau_{\mathrm{id}}\rangle] : \tau}{\emptyset \vdash \mathscr{E}[\langle x\rangle] : \tau}\ \textsc{Proj-X}$$

However, $\mathscr{E}[x \triangleright \tau_{\mathrm{id}}]$ does not hold (the proof is similar to the previous case).

Case $\mathscr{E}$ *is* $\mathsf{let}\ x = [\lambda z.\,z]\ \mathsf{in}\ (\langle x\rangle\ 1, \langle\square\rangle\ ())$. This is symmetric to the previous case, which cannot hold either.

Example A.5. Let e be $\mathsf{let}\ f = \lambda x.\,x.1\ \mathsf{in}\ f\ (1, 2)$. e is well-typed using *backpropagation*. e is of the form $\mathscr{E}[x]$ where $\mathscr{E}$ is the context $\mathsf{let}\ f = \lambda x.\,\square\ \mathsf{in}\ f\ (1, 2)$. We have $\emptyset \vdash \mathscr{E}[x.1/2] : \mathsf{int}$. Let us show that $\mathscr{E}[x \triangleright \nu\gamma_1, \gamma_2.\,\gamma_1 * \gamma_2]$. Assume $\emptyset \vdash \mathscr{E}[\{(x : \mathfrak{g})\}] : \tau$. As $\mathfrak{g}$ is a ground type, the type $\mathfrak{g}$ of x is not a variable. Then, $\mathfrak{g}$ cannot be that of an arbitrary sized tuple, since there is no such type for a tuple of arbitrary size. Hence, $\mathfrak{g}$ must be a tuple $\Pi_{i=1}^{n}\bar{\tau}$ for some size n. Since the codomain of f must be a tuple of size 2 (for $f\ (1, 2)$ to be well-typed), then n must also be 2. This shows that $\mathscr{E}[x \triangleright \nu\gamma_1, \gamma_2.\,\gamma_1 * \gamma_2]$.

A.3 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 concertize this by specifying the patterns used in OmniML (in Figure 12), 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 $\mathsf{t}\ \bar{\tau}$ 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.*, $\forall \mathrm{fv}(\tau).\, \tau$).

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

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

$$
\textsc{Lab-Inst}\ \frac{\phi \vdash \Omega(\ell/\phi(t)) \leq \tau_1 \to \tau_2}{\phi \vdash \Omega(\ell/t) \leq \tau_1 \to \tau_2}
\qquad
\textsc{Scm-Inst}\ \frac{\phi \vdash \phi(s) \leq \tau}{\phi \vdash s \leq \tau}
\qquad
\textsc{Abs-Inst}\ \frac{\phi \vdash x \leq \phi(s)}{\phi \vdash x \leq s}
$$

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

Fig. 12. Patterns for OmniML.

Constraint generation. The function $[\![- : =]\!]$ is defined in Figure 13. 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 generates 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. Tuples introduce fresh variables for each component and unify their product with α. Explicit projections ensure e has a tuple type $\Pi_{i=1}^{n}\bar{\beta}$ and extract the j-th component β_j, unifying it with α. Implicit projections defer this via a suspended match constraint, until the shape of e's expected type is known to be a tuple, extracting the j-th component with the pattern $\Pi\ \beta_j$,

For polytypes, boxing asserts that e has the polymorphic type σ (using universal quantification) and that the expected type is the polytype $[\sigma]$. 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. Explicit unboxing is analogous, but uses an explicit scheme σ and therefore does not require a suspended match constraint. Implicit boxing infers the principal type for e using a let constraint

$$
\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_1, \ldots, e_n) : \alpha \rrbracket & \triangleq & \exists \bar{\alpha}.\, \alpha = \Pi_{i=1}^{n} \bar{\alpha} \wedge \bigwedge_{i=1}^{n} \llbracket e_i : \alpha_i \rrbracket \\
\llbracket e.j/n : \alpha \rrbracket & \triangleq & \exists \beta, \bar{\beta}.\, \llbracket e : \beta \rrbracket \wedge \beta = \Pi_{i=1}^{n} \bar{\beta} \wedge \alpha = \beta_j \\
\llbracket e.j : \alpha \rrbracket & \triangleq & \exists \beta.\, \llbracket e : \beta \rrbracket \wedge \mathsf{match}\ \beta\ \mathsf{with}\ \Pi\ \gamma_j \to \alpha = \gamma \\
\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 \alpha.\, \llbracket e : \alpha \rrbracket \wedge \mathsf{match}\ \alpha\ \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.l : \alpha \rrbracket & \triangleq & \exists \beta.\, \llbracket e : \beta \rrbracket \wedge \llbracket l \,!\, \beta \rrbracket \wedge \llbracket \ell : \alpha \to \beta \rrbracket \\
\llbracket \{\overline{l = e}\} : \alpha \rrbracket & \triangleq & \llbracket \bar{l} \,!\, \alpha \rrbracket \wedge \bigwedge_{i=1}^{n} \llbracket l_i = e_i : \alpha \rrbracket \\
\llbracket \{e\} : \alpha \rrbracket & \triangleq & \exists \beta.\, \llbracket e : \beta \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 l = e : \alpha \rrbracket & \triangleq & \exists \beta.\, \llbracket e : \beta \rrbracket \wedge \llbracket l : \beta \to \alpha \rrbracket \\
\\
\llbracket \ell : \alpha \to \beta \rrbracket & \triangleq & \mathsf{match}\ \tau_2\ \mathsf{with}\ t\ \_ \to \Omega(\ell/t) \leq \alpha \to \beta \\
\llbracket \ell/\mathrm{t}\!: \alpha \to \beta \rrbracket & \triangleq & \Omega(\ell/\mathrm{t}) \leq \alpha \to \beta \\
\llbracket \{\ell\} : \alpha \to \beta \rrbracket & \triangleq & \mathsf{true} \\
\llbracket (l : \exists \bar{\alpha}.\, \tau) : \alpha \to \beta \rrbracket & \triangleq & \exists \bar{\alpha}.\, \beta = \tau \wedge \llbracket l : \alpha \to \beta \rrbracket \\
\llbracket \bar{l} \,!\, \alpha \rrbracket & \triangleq & \begin{cases} \exists \bar{\alpha}.\, \alpha = \mathrm{t}\ \bar{\alpha} & \text{if } \bar{l} \,!\, \mathrm{t} \in \Omega \\ \mathsf{true} & \text{otherwise} \end{cases}
\end{array}
$$

Fig. 13. The constraint generation translation for OmniML.

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, constraining e to this type, and use the auxiliary function $\llbracket l : \alpha \to \beta \rrbracket$ to instantiate the label. The function $\llbracket \bar{l} \,!\, \alpha \rrbracket$ checks whether a label sequence $\bar{l}$ uniquely determines a record type, unifying α with $\mathrm{t}\ \bar{\alpha}$ if so, or leaving it unconstrained if ambiguous. This function enables closed-world reasoning for both projections and constructions, and corresponds to the judgment $\bar{l} \,!\, \tau$ judgment defined in §A.1.

Record construction checks label uniqueness and generates a per-field constraint $l_i = e_i$, introducing a fresh variable β for each field's type and ensuring that e has this type and the label l instantiates to $\beta \to \alpha$. Label instantiation constraints $\llbracket \ell : \alpha \to \beta \rrbracket$ suspend until β is known to be a record type; once resolved, the label type is looked up in Ω and instantiated. Explicit instantiations bypass suspension and directly instantiate the label's type.

B Unification

The unification rules are listed in Figure 14. 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] but replaces type constructors with *canonical principal shapes*, enabling a uniform treatment of monotypes and polytypes within unification compared to prior formulations [Garrigue and Rémy 1999].

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

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

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

Fig. 14. Unification algorithm as a series of rewriting rules $U_1 \longrightarrow U_2$. All shapes are principal.

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[8]. 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 $\mathrm{cyclic}\,(U)$, if $\alpha \prec_U^* \alpha$ for some α.

[8]We discuss relaxing this constraint in §8.

C Full technical reference

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

$$
\begin{array}{rcll}
\alpha, \beta, \gamma & \in & \mathcal{V} & \text{Type variables} \\
\tau & ::= & \alpha \mid 1 \mid \tau_1 \to \tau_2 \mid \Pi_{i=1}^{n} \tau_i \mid \mathsf{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{R} & & & \text{Sets of ground types} \\
\mathfrak{R} \subseteq \mathcal{G} & & & \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 & \bar{\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(\ell/t) \leq \tau_1 \to \tau_2 \mid \Omega(\ell/\mathsf{t}) \leq \tau_1 \to \tau_2 & \\
& \mid & s \leq \tau \mid \sigma \leq \tau \mid x \leq s \mid x \leq \sigma & \\
\chi & ::= & \rho \to 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{l = e}\} \mid e.l & \\
& \mid & (e_1, \ldots, e_n) \mid e.j \mid e.j/n & \\
& \mid & [e] \mid [e : \exists \bar{\alpha}.\, \sigma] \mid \langle e \rangle \mid \langle e : \exists \bar{\alpha}.\, \sigma \rangle & \\
& \mid & \{e\} & \\
l & ::= & \ell \mid \ell/\mathsf{t} \mid \{\ell\} \mid (l : \exists \bar{\alpha}.\, \tau) & \text{Labels} \\
\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 & \{l_1 = e_1\ \ldots\ l_i = \mathscr{E}\ \ldots\ l_n = e_n\} \mid \mathscr{E}.l & \\
& \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 & \{\mathscr{E}\} & \\
\mathscr{L} & ::= & \mathscr{E}[e.\square] \mid \mathscr{E}[\{l_1 = e_1; \ldots; \square = e_i; \ldots; l_n = e_n\}] & \text{Label contexts} \\
\Gamma & ::= & \emptyset \mid \Gamma, x : \sigma & \text{Typing contexts} \\
\Omega & ::= & \emptyset \mid \Omega, \ell : \forall \bar{\alpha}.\, \tau \to \mathsf{t}\ \bar{\alpha} & \text{Label environment}
\end{array}
$$

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

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

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

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

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

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

$$\textsc{Scm-Inst}\ \frac{\phi \vdash \phi(s) \leq \tau}{\phi \vdash s \leq \tau} \qquad \textsc{Abs-Inst}\ \frac{\phi \vdash x \leq \phi(s)}{\phi \vdash x \leq s}$$

$$\begin{array}{rcll}
\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \rho \to \bar{C} & \triangleq & \exists \bar{\alpha}.\, \tau = \varsigma\ \bar{\alpha} \wedge \theta(C_i) & \text{if } \rho_i \text{ matches } \varsigma\ \bar{\alpha} = \theta \\
\Omega(\ell/\mathsf{t}) \leq \tau_1 \to \tau_2 & \triangleq & \exists \bar{\alpha}.\, \tau_1 = \tau \wedge \tau_2 = \mathsf{t}\ \bar{\alpha} & \text{if } \Omega(\ell/\mathsf{t}) = \forall \bar{\alpha}.\, \tau \to \mathsf{t}\ \bar{\alpha} \\
(\forall \bar{\alpha}.\, \tau') \leq \tau & \triangleq & \exists \bar{\alpha}.\, \tau' = \tau & \\
x \leq (\forall \bar{\alpha}.\, \tau) & \triangleq & \forall \bar{\alpha}.\, x\ \tau &
\end{array}$$

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

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

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

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 C.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. shape (τ) 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.

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

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

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

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

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

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

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

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

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

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

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

$\boxed{l : \tau_1 \to \tau_2}$ The label l has the field type τ_1 and record type τ_2.

$$
\textsc{Lab-Magic}\ \frac{}{\Gamma \vdash \{\ell\} : \tau' \to \tau}
\qquad
\textsc{Lab-Annot}\ \frac{\Gamma \vdash l : \tau' \to \tau[\bar{\alpha} := \bar{\tau}]}{\Gamma \vdash (l : \exists \bar{\alpha}.\, \tau) : \tau' \to \tau[\bar{\alpha} := \bar{\tau}]}
\qquad
\textsc{Lab-X}\ \frac{\Omega(\ell/\mathsf{t}) = \forall \bar{\alpha}.\, \tau \to \mathsf{t}\ \bar{\alpha}}{\Gamma \vdash \ell/\mathsf{t}\colon \bar{\tau}[\bar{\alpha} := \bar{\tau}] \to \mathsf{t}\ \bar{\tau}}
$$

$\boxed{l/\mathsf{t} \in \Omega}$ The label l belongs to the nominal record type t in Ω.

$\boxed{\bar{l}\,!\,\mathsf{t} \in \Omega}$ The set of labels $\bar{l}$ belong to a unique nominal record type t in Ω.

$\boxed{\bar{l}\,!\,\tau}$ The set of labels $\bar{l}$ infer the possibly unique type τ.

$$
\textsc{Lab-inI}\ \frac{\ell : \forall \bar{\alpha}.\, \tau \to \mathsf{t}\ \bar{\alpha} \in \Omega}{\ell/\mathsf{t} \in \Omega}
\qquad
\textsc{Lab-inX}\ \frac{\ell/\mathsf{t} \in \Omega}{(\ell/\mathsf{t})/\mathsf{t} \in \Omega}
\qquad
\textsc{Lab-inMagic}\ \frac{\ell/\mathsf{t} \in \Omega}{\{\ell\}/\mathsf{t} \in \Omega}
\qquad
\textsc{Lab-inAnnot}\ \frac{l/\mathsf{t} \in \Omega}{(l : \exists \bar{\alpha}.\, \tau)/\mathsf{t} \in \Omega}
$$

$$
\textsc{Lab-U}\ \frac{\bar{l}/\mathsf{t} \in \Omega \qquad \forall \mathsf{t}',\ \bar{l}/\mathsf{t}' \in \Omega \implies \mathsf{t} = \mathsf{t}'}{\bar{l}\,!\,\mathsf{t} \in \Omega}
\qquad
\textsc{Lab-!}\ \frac{\bar{l}\,!\,\mathsf{t} \in \Omega}{l\,!\,\mathsf{t}\ \bar{\tau}}
\qquad
\textsc{Lab-?}\ \frac{}{\bar{l}\,!\,\tau}
$$

$\boxed{\Gamma \vdash l = e : \tau}$ Under the typing context Γ, the record assignment $l = e$ has the record type τ.

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

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

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

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

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

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

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

$$\textsc{Rcd}\ \frac{(\Gamma \vdash l_i = e_i : \tau)_{i=1}^{n} \qquad \bar l\ !\ \tau}{\Gamma \vdash \{l_1 = e_1; \ldots; l_n = e_n\} : \tau} \qquad \textsc{Rcd-Proj}\ \frac{\Gamma \vdash e : \tau' \qquad l : \tau \to \tau' \qquad l\ !\ \tau}{\Gamma \vdash e.l : \tau} \qquad \textsc{Lab-I}\ \frac{\mathscr{L}[\ell\ !\ \mathsf{t}] \qquad \Gamma \vdash \mathscr{L}[\ell/\mathsf{t}] : \tau}{\Gamma \vdash \mathscr{L}[\ell] : \tau}$$

$$\textsc{Magic}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \{e\} : \tau'}$$

$$\begin{array}{rcl} E[e \triangleright \varsigma] & \triangleq & \forall \Gamma, \tau, \mathfrak{g},\ \Gamma \vdash \lfloor \mathscr{E}[\{(e : \mathfrak{g})\}] \rfloor : \tau \implies \mathsf{shape}\ (\mathfrak{g}) = \varsigma \\ E[e \triangleleft \varsigma] & \triangleq & \forall \Gamma, \tau, \mathfrak{g},\ \Gamma \vdash \lfloor \mathscr{E}[(\{e\} : \mathfrak{g})] \rfloor : \tau \implies \mathsf{shape}\ (\mathfrak{g}) = \varsigma \\ \mathscr{L}[\ell\ !\ \mathsf{t}] & \triangleq & \forall \Gamma, \tau, \mathfrak{g},\ \Gamma \vdash \lfloor \mathscr{L}[(\{\ell\} : \mathfrak{g})] \rfloor : \tau \implies \mathsf{shape}\ (\mathfrak{g}) = \nu\bar\gamma.\, \bar\gamma\mathsf{t} \end{array}$$

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

$\boxed{[\![\bar l\ !\ \alpha]\!]}$ $[\![\bar l\ !\ \alpha]\!]$ is satisfiable iff $\bar l$ has the possibly unique type α.

$\boxed{[\![l : \alpha \to \beta]\!]}$ $[\![l : \alpha \to \beta]\!]$ is satisfiable iff l has the record type β and field type α.

$\boxed{[\![l = e : \alpha]\!]}$ $[\![l = e : \alpha]\!]$ is satisfiable iff the record assignment $l = e$ has the record type α.

$\boxed{[\![e : \sigma]\!]}$ $[\![e : \sigma]\!]$ 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 \text{let } x = \lambda \beta'.\, \beta' = \beta \text{ in } [\![e : \gamma]\!] \\
[\![e_1\ e_2 : \alpha]\!] & \triangleq & \exists \beta \gamma.\, \gamma = \beta \to \alpha \wedge [\![e_1 : \gamma]\!] \wedge [\![e_2 : \beta]\!] \\
[\![\text{let } x = e_1 \text{ in } e_2 : \alpha]\!] & \triangleq & \text{let } x = \lambda \beta.\, [\![e_1 : \beta]\!] \text{ 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 \text{match } \beta \text{ 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 \leq \alpha \\
[\![\langle e \rangle : \alpha]\!] & \triangleq & \exists \alpha.\, [\![e : \alpha]\!] \wedge \text{match } \alpha \text{ with } [s] \to s \leq \alpha \\
[\![[e] : \alpha]\!] & \triangleq & \text{let } x = \lambda \beta.\, [\![e : \beta]\!] \text{ in match } \alpha \text{ with } [s] \to x \leq s \\
[\![e.l : \alpha]\!] & \triangleq & \exists \beta.\, [\![e : \beta]\!] \wedge [\![l \,!\, \beta]\!] \wedge [\![\ell : \alpha \to \beta]\!] \\
[\![\{\overline{l = e}\} : \alpha]\!] & \triangleq & [\![\bar{l} \,!\, \alpha]\!] \wedge \bigwedge_{i=1}^{n} [\![l_i = e_i : \alpha]\!] \\
[\![\{e\} : \alpha]\!] & \triangleq & \exists \beta.\, [\![e : \beta]\!]
\end{array}
$$

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

$$
[\![l = e : \alpha]\!] \triangleq \exists \beta.\, [\![e : \beta]\!] \wedge [\![l : \beta \to \alpha]\!]
$$

$$
\begin{array}{lcl}
[\![\ell : \alpha \to \beta]\!] & \triangleq & \text{match } \tau_2 \text{ with } t\ \_ \to \Omega(\ell/t) \leq \alpha \to \beta \\
[\![\ell/\mathrm{t} \colon \alpha \to \beta]\!] & \triangleq & \Omega(\ell/\mathrm{t}) \leq \alpha \to \beta \\
[\![\{\ell\} : \alpha \to \beta]\!] & \triangleq & \text{true} \\
[\![(l : \exists \bar{\alpha}.\, \tau) : \alpha \to \beta]\!] & \triangleq & \exists \bar{\alpha}.\, \beta = \tau \wedge [\![l : \alpha \to \beta]\!] \\
[\![\bar{l} \,!\, \alpha]\!] & \triangleq & \begin{cases} \exists \bar{\alpha}.\, \alpha = \mathrm{t}\ \bar{\alpha} & \text{if } \bar{l} \,!\, \mathrm{t} \in \Omega \\ \text{true} & \text{otherwise} \end{cases}
\end{array}
$$

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

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

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

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

$$\textsc{Simple-PolyX}\ \frac{e \text{ simple}}{[e : \exists \bar{\alpha}.\, \sigma] \text{ simple}} \qquad \textsc{Simple-UseX}\ \frac{e \text{ simple}}{\langle e : \exists \bar{\alpha}.\, \sigma \rangle \text{ simple}} \qquad \textsc{Simple-Rcd}\ \frac{(l_i = e_i \text{ simple})_{i=1}^{n}}{\{l_1 = e_1\ \ldots\ l_n = e_n\}} \qquad \textsc{Simple-Rcd-Assn}\ \frac{l \text{ simple} \qquad e \text{ simple}}{l = e \text{ simple}}$$

$$\textsc{Simple-Rcd-Proj}\ \frac{e \text{ simple} \qquad l \text{ simple}}{e.l \text{ simple}} \qquad \textsc{Simple-Magic}\ \frac{e \text{ simple}}{\{e\} \text{ simple}} \qquad \textsc{Simple-Lab}\ \frac{}{\ell/\mathrm{t} \text{ simple}} \qquad \textsc{Simple-Lab-Magic}\ \frac{}{\{\ell\} \text{ simple}} \qquad \textsc{Simple-Lab-Annot}\ \frac{l \text{ simple}}{(l : \exists \bar{\alpha}.\, \tau) \text{ simple}}$$

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

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

$$\begin{aligned}
\lfloor x \rfloor &\triangleq x \\
\lfloor \lambda x.\, e \rfloor &\triangleq \lambda x.\, \lfloor e \rfloor \\
\lfloor e_1\ e_2 \rfloor &\triangleq \lfloor e_1 \rfloor\ \lfloor e_2 \rfloor \\
\lfloor () \rfloor &\triangleq () \\
\lfloor \text{let } x = e_1 \text{ in } e_2 \rfloor &\triangleq \text{let } x = \lfloor e_1 \rfloor \text{ in } \lfloor e_2 \rfloor \\
\lfloor (e : \exists \bar{\alpha}.\, \tau) \rfloor &\triangleq (\lfloor e \rfloor : \exists \bar{\alpha}.\, \tau) \\
\lfloor (e_1, \ldots, e_n) \rfloor &\triangleq (\lfloor e_1 \rfloor, \ldots, \lfloor e_n \rfloor) \\
\lfloor e.j \rfloor &\triangleq \{\lfloor e \rfloor\} \\
\lfloor e.j/n \rfloor &\triangleq \lfloor e \rfloor.j/n \\
\lfloor [e : \exists \bar{\alpha}.\, \sigma] \rfloor &\triangleq [\lfloor e \rfloor : \exists \bar{\alpha}.\, \sigma] \\
\lfloor [e] \rfloor &\triangleq \{\lfloor e \rfloor\} \\
\lfloor \langle e \rangle \rfloor &\triangleq \{\lfloor e \rfloor\} \\
\lfloor \langle e : \exists \bar{\alpha}.\, \sigma \rangle \rfloor &\triangleq \langle \lfloor e \rfloor : \exists \bar{\alpha}.\, \sigma \rangle \\
\lfloor \{l_1 = e_1\ \ldots\ l_n = e_n\} \rfloor &\triangleq \{\lfloor l_1 \rfloor = \lfloor e_1 \rfloor\ \ldots\ \lfloor l_n \rfloor = \lfloor e_n \rfloor\} \\
\lfloor e.l \rfloor &\triangleq \lfloor e \rfloor.\lfloor l \rfloor \\
\lfloor \ell/\mathrm{t} \rfloor &\triangleq \ell/\mathrm{t} \\
\lfloor \ell \rfloor &\triangleq \{\ell\} \\
\lfloor (l : \exists \bar{\alpha}.\, \tau) \rfloor &\triangleq (\lfloor l \rfloor : \exists \bar{\alpha}.\, \tau) \\
\lfloor \{\ell\} \rfloor &\triangleq \{\ell\}
\end{aligned}$$

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

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

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

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

$$\textsc{Can-Use-I}\ \frac{\mathscr{E}[e \triangleright \nu\bar{\gamma}.[\sigma]] \qquad \Vdash \mathscr{E}[\langle e : \exists\bar{\gamma}.\sigma\rangle] : \tau}{\Vdash \mathscr{E}[\langle e\rangle] : \tau} \qquad \textsc{Can-Lab-I}\ \frac{\mathscr{L}[\ell\,!\,\mathrm{t}] \qquad \Vdash \mathscr{L}[\ell/\mathrm{t}] : \tau}{\Vdash \mathscr{L}[\ell] : \tau}$$

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

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

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

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

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

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

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

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

$$\textsc{S-Let-ConjRight}\ \frac{\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = C_1\ \mathsf{in}\ (C_2 \wedge C_3) \qquad x \# C_3}{C_3 \wedge \mathsf{let}\ x\ \alpha = C_1\ \mathsf{in}\ C_2} \qquad \textsc{S-Match-Type}\ \frac{\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi} \qquad \tau \notin \mathcal{V}}{\mathsf{match}\ \tau := \mathsf{shape}\ (\tau)\ \mathsf{with}\ \bar{\chi}}$$

$$\textsc{S-Match-Var}\quad \frac{\mathscr{C}[\mathsf{match}\ \alpha\ \mathsf{with}\ \bar{\chi}] \qquad \alpha = \tau = \epsilon \in \mathscr{C}}{\mathscr{C}[\mathsf{match}\ \alpha := \mathsf{shape}\ (\tau)\ \mathsf{with}\ \bar{\chi}]}
\qquad
\textsc{S-Inst-Name}\quad \frac{i[\alpha \rightsquigarrow \tau] \qquad \tau \notin \mathcal{V}}{\exists \gamma.\, \gamma = \tau \wedge i[\alpha \rightsquigarrow \gamma]}$$

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

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

$$\textsc{S-Inst-Unify}\quad \frac{i[\alpha \rightsquigarrow \gamma_1] \wedge i[\alpha \rightsquigarrow \gamma_2]}{i[\alpha \rightsquigarrow \gamma_1] \wedge \gamma_1 = \gamma_2}
\qquad
\textsc{S-Inst-Poly}\quad \frac{\begin{array}{c}\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = \bar{\epsilon} \wedge C\ \mathsf{in}\ \mathscr{C}[i^x[\alpha' \rightsquigarrow \gamma]] \qquad \forall \alpha'.\, \exists \alpha.\, \bar{\epsilon} \equiv \mathsf{true} \\ \alpha' \in \alpha, \bar{\alpha} \qquad \alpha' \# C \qquad i.\alpha' \# \mathrm{insts}(\mathscr{C}) \qquad x \# \mathrm{bv}(\mathscr{C})\end{array}}{\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = \bar{\epsilon} \wedge C\ \mathsf{in}\ \mathscr{C}[\mathsf{true}]}$$

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

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

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

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

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

$$\textsc{S-BackProp}\quad \frac{\begin{array}{c}\mathscr{C}[\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = \mathscr{C}_1[\mathsf{match}\ \alpha'\ \mathsf{with}\ \bar{\chi}]\ \mathsf{in}\ \mathscr{C}_2[i^x[\alpha' \rightsquigarrow \gamma]]] \\ \alpha' \in \alpha, \bar{\alpha} \qquad \gamma = \tau = \epsilon \in \mathscr{C}[\mathscr{C}_2] \qquad x \# \mathrm{bv}(\mathscr{C}_2)\end{array}}{\mathscr{C}[\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = \mathscr{C}_1[\mathsf{match}\ \alpha' := \mathsf{shape}\ (\tau)\ \mathsf{with}\ \bar{\chi}]\ \mathsf{in}\ \mathscr{C}_2[i^x[\alpha' \rightsquigarrow \gamma]]]}
\qquad
\textsc{S-Exists-Exists-Inst}\quad \frac{\exists i^x.\, \exists \alpha.\, C}{\exists \alpha.\, \exists i^x.\, C}$$

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

$$\textsc{S-All-Conj}\qquad \begin{array}{c} \forall\bar{\alpha}.\,\exists\bar{\beta}.\,C_1 \wedge C_2 \qquad \bar{\alpha},\bar{\beta} \# C_1 \\ \hline \longrightarrow \\ C_1 \wedge \forall\bar{\alpha}.\,\exists\bar{\beta}.\,C_2 \end{array}$$

$$\textsc{S-Exists-All}\qquad \begin{array}{c} \forall\bar{\alpha}.\,\exists\bar{\beta},\bar{\gamma}.\,C \qquad \exists\bar{\alpha},\bar{\beta}.\,C \text{ determines } \bar{\gamma} \\ \hline \longrightarrow \\ \exists\bar{\gamma}.\,\forall\bar{\alpha}.\,\exists\bar{\beta}.\,C \end{array}$$

$$\textsc{S-All-Escape}\qquad \begin{array}{c} \forall\bar{\alpha},\alpha.\,\exists\bar{\beta}.\,C \wedge \bar{\epsilon} \qquad \alpha \prec^{*}_{\bar{\epsilon}} \gamma \qquad \gamma \# \alpha,\bar{\beta} \qquad \alpha \# \bar{\beta} \\ \hline \longrightarrow \\ \text{false} \end{array}$$

$$\textsc{S-All-Rigid}\qquad \begin{array}{c} \forall\bar{\alpha},\alpha.\,\exists\bar{\beta}.\,C \wedge \alpha = \tau = \epsilon \qquad \tau \notin \mathcal{V} \qquad \alpha \# \bar{\beta} \\ \hline \longrightarrow \\ \text{false} \end{array}$$

$$\textsc{S-All-Solve}\qquad \begin{array}{c} \forall\bar{\alpha}.\,\exists\bar{\beta}.\,\bar{\epsilon} \qquad \exists\bar{\beta}.\,\bar{\epsilon} \equiv \text{true} \\ \hline \longrightarrow \\ \text{true} \end{array}$$

Definition C.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.

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

Definition C.3. A context $\mathscr{C}$ proves a multi-equation ϵ, written $\epsilon \in \mathscr{C}$, if there exists a decomposition $\mathscr{C} = \mathscr{C}_1[\epsilon \wedge \mathscr{C}_2]$ such that $\mathrm{fv}(\epsilon) \# \mathrm{bv}(\mathscr{C}_2)$

Definition C.4 (Measure). For the relation $\phi \vdash C$, the following measure enables a useful induction principle:

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

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

(1) $\#\text{match } C$ 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.

D Properties of the constraint language

This appendix establishes key properties of the constraint language. The first is the principality of shapes Theorem D.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 §F.

D.1 Principality of shapes

Theorem D.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).

□

D.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 Susp-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 §E.

We define in §C 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 §C a "canonical" semantic judgment $\phi \Vdash C$ that enforces the structure we mentioned: its derivation starts by discharging suspended constraints, until eventually we reach a simple constraint C. Below we prove that any semantic derivation $\phi \vdash C$ can be turned into a canonical semantic derivation $\phi \Vdash C$.

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

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

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

Lemma D.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 $\mathscr{C}[\tau \,!\, \varsigma]$, 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 D.3 (Composability of unicity). *If $\mathscr{C}_1[\tau \,!\, \varsigma]$, then $\mathscr{C}_2[\mathscr{C}_1][\tau \,!\, \varsigma]$.*

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

Case $\square$. immediate.

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

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

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

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

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

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

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

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

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

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

Case let $x = \lambda\alpha.\,\mathscr{C}_3$ 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 (let $x \ldots)[\tau \,!\, \varsigma]$
	$\phi \vdash$ let $x = \lambda\alpha.\,\lfloor\mathscr{C}_3[\mathscr{C}_1][\tau = \mathfrak{g}]\rfloor$ in $\lfloor C\rfloor$	$\Longrightarrow$ I
	$\phi \vdash \exists\alpha.\,\lfloor\mathscr{C}_3[\mathscr{C}_1][\tau = \mathfrak{g}]\rfloor$	Simple inversion
	$\phi[\alpha := \mathfrak{g}'] \vdash \lfloor\mathscr{C}_3[\mathscr{C}_1][\tau = \mathfrak{g}]\rfloor$	Simple inversion
	shape $(\mathfrak{g}) = \varsigma$	$\Longrightarrow$ E on $\mathscr{C}_3[\mathscr{C}_1][\tau \,!\, \varsigma]$
☞	(let $x = \lambda\alpha.\,\mathscr{C}_3[\mathscr{C}_1]$ in $C)[\tau \,!\, \varsigma]$	Above

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

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

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

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

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

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

□

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

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

Theorem D.6 (Canonicalization). *If $\phi \vdash C$, then $\phi \Vdash C$.*

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

Case

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

☞ $\phi \Vdash$ 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

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

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

☞ $\phi \Vdash C_1 \wedge C_2$ immediate by CAN-BASE

Subcase

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

$\phi \Vdash C_2$

$\phi \Vdash \mathscr{C}[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}]$	Premise
$\phi \vdash \mathscr{C}[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}]$	Lemma D.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 D.3
☞ $\phi \Vdash \mathscr{C}[\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}]$	By CAN-SUSP-CTX

Subcase

$\phi \Vdash C_1$

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

Symmetric to the above case.

Case

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

$$\begin{array}{rl} \phi[\alpha := \mathfrak{g}] \vdash C & \text{Premise} \\ \phi[\alpha := \mathfrak{g}] \Vdash C & \text{By } \textit{i.h.} \end{array}$$

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

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

$$\begin{array}{rl} \phi[\alpha := \mathfrak{g}] \Vdash \mathscr{C}[\text{match } \tau := \varsigma \text{ with } \bar{\chi}] & \text{Premise} \\ \phi[\alpha := \mathfrak{g}] \vdash \mathscr{C}[\text{match } \tau := \varsigma \text{ with } \bar{\chi}] & \text{Lemma D.5} \end{array}$$

$$\begin{array}{rl} \phi \vdash \exists \alpha.\, \mathscr{C}[\text{match } \tau := \varsigma \text{ with } \bar{\chi}] & \text{By } \textsc{Exists} \\ \phi \Vdash \exists \alpha.\, \mathscr{C}[\text{match } \tau := \varsigma \text{ with } \bar{\chi}] & \text{By } \textit{i.h.} \\ \mathscr{C}[\tau \,!\, \varsigma] & \text{Premise} \\ (\exists \alpha.\, \mathscr{C})[\tau \,!\, \varsigma] & \text{Lemma D.3} \\ ☞\ \phi \Vdash \exists \alpha.\, \mathscr{C}[\text{match } \tau \text{ with } \bar{\chi}] & \text{By } \textsc{Can-Susp-Ctx} \end{array}$$

Case

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

Similar to the Exists case.

Case

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

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

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

Subcase

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

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

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

Subcase

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

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

$(\exists \alpha.\, \mathscr{C})[\tau \,!\, \varsigma]$	Premise
$\mathscr{C}[\tau \,!\, \varsigma]$	Lemma D.4
$\phi \Vdash \exists \alpha.\, \mathscr{C}[\text{match } \tau := \varsigma \text{ with } \bar{\chi}]$	Premise
$\phi \vdash \exists \alpha.\, \mathscr{C}[\text{match } \tau := \varsigma \text{ with } \bar{\chi}]$	Lemma D.5

	$\phi(\lambda \alpha.\, C_1) = \phi(\lambda \alpha.\, \mathscr{C}[\text{match } \tau := \varsigma \text{ with } \bar{\chi}])$	Corollary D.8
	$\phi \vdash \text{let } x = \lambda \alpha.\, \mathscr{C}[\text{match } \tau := \varsigma \text{ with } \bar{\chi}] \text{ in } C_2$	By Let
	$\phi \Vdash \text{let } x = \lambda \alpha.\, \mathscr{C}[\text{match } \tau := \varsigma \text{ with } \bar{\chi}] \text{ in } C_2$	By *i.h.*
	$(\text{let } x = \lambda \alpha.\, \mathscr{C} \text{ in } C_2)[\tau \,!\, \varsigma]$	Lemma D.3
☞	$\phi \Vdash \text{let } x = \lambda \alpha.\, \mathscr{C}[\text{match } \tau \text{ with } \bar{\chi}] \text{ in } C_2$	By Can-Susp-Ctx

Subcase

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

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

	$\mathscr{C}[\tau \,!\, \varsigma]$	Premise
	$(\text{let } x = \lambda \alpha.\, C_1 \text{ in } \mathscr{C})[\tau \,!\, \varsigma]$	Lemma D.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 D.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-Susp-Ctx

Case

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

Similar to the True case.

Case

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

Similar to the Let case.

Case

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

Similar to the App case.

Case

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

Similar to the Exists case.

Case

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

Similar to the Unif case.

Case

$$\dfrac{\phi(i)(\alpha) = \phi(\tau)}{\phi \vdash i[\alpha \rightsquigarrow \tau]}\ \textsc{Partial-Inst}$$

Similar to the App case.

□

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

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

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

Case

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

The second premise is a contradiction.

Case

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

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

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

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

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

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

$\phi \Vdash \mathscr{C}_2[\text{match } \tau \text{ with } \bar{\chi}, \text{match } \tau' := \varsigma' \text{ with } \bar{\chi}']$	Premise
For all $\phi', \mathfrak{g}'$	Defn. of $\mathscr{C}_2[\square, \text{match } \tau' := \varsigma' \text{ with } \bar{\chi}'][\tau \,!\, \varsigma]$
$\phi' \vdash \lfloor \mathscr{C}_2[\tau = \mathfrak{g}', \text{match } \tau' := \varsigma' \text{ with } \bar{\chi}'] \rfloor$	$\Longrightarrow$ I
$\phi' \vdash \lfloor \mathscr{C}_2[\tau = \mathfrak{g}', \text{true}] \rfloor$	Lemma D.2
$\lfloor \mathscr{C}_2[\tau = \mathfrak{g}', \text{true}] \rfloor = \lfloor \mathscr{C}_2[\tau = \mathfrak{g}', \lfloor \text{match } \tau' \text{ with } \bar{\chi}' \rfloor] \rfloor$	By definition
$= \lfloor \mathscr{C}[\tau = \mathfrak{g}'] \rfloor$	By definition
$\phi' \vdash \lfloor \mathscr{C}[\tau = \mathfrak{g}'] \rfloor$	Above
$\text{shape } (\mathfrak{g}') = \varsigma$	$\Longrightarrow$ E on $\mathscr{C}[\tau \,!\, \varsigma]$
$\mathscr{C}_2[\square, \text{match } \tau' := \varsigma' \text{ with } \bar{\chi}'][\tau \,!\, \varsigma]$	Above
$\phi \Vdash \mathscr{C}_2[\text{match } \tau := \varsigma \text{ with } \bar{\chi}, \text{match } \tau' := \varsigma' \text{ with } \bar{\chi}']$	By *i.h.*
For all $\phi', \mathfrak{g}'$	Defn. of $\mathscr{C}_2[\text{match } \tau := \varsigma \text{ with } \bar{\chi}, \square][\tau' \,!\, \varsigma']$
$\phi' \vdash \lfloor \mathscr{C}_2[\text{match } \tau := \varsigma \text{ with } \bar{\chi}, \tau' = \mathfrak{g}'] \rfloor$	$\Longrightarrow$ I
$\phi' \vdash \lfloor \mathscr{C}_2[\text{true}, \tau' = \mathfrak{g}'] \rfloor$	Lemma D.2
$\lfloor \mathscr{C}_2[\text{true}, \tau' = \mathfrak{g}'] \rfloor = \lfloor \mathscr{C}_2[\lfloor \text{match } \tau \text{ with } \bar{\chi} \rfloor, \tau' = \mathfrak{g}'] \rfloor$	By definition
$= \lfloor \mathscr{C}'[\tau' = \mathfrak{g}'] \rfloor$	By definition
$\phi' \vdash \lfloor \mathscr{C}[\tau = \mathfrak{g}'] \rfloor$	Above
$\mathscr{C}'[\tau' \,!\, \varsigma']$	Premise
$\text{shape } (\mathfrak{g}') = \varsigma'$	$\Longrightarrow$ E on $\mathscr{C}'[\tau' \,!\, \varsigma']$
$\mathscr{C}_2[\text{match } \tau := \varsigma \text{ with } \bar{\chi}, \square][\tau' \,!\, \varsigma']$	Above
☞ $\phi \Vdash \mathscr{C}_2[\text{match } \tau := \varsigma \text{ with } \bar{\chi}, \text{match } \tau' \text{ with } \bar{\chi}']$	By Con-Susp-Ctx

□

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

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

Case $\Longrightarrow$.

$\mathscr{C}[\tau \,!\, \varsigma]$	Premise
$\phi[\alpha := \mathfrak{g}] \vdash \mathscr{C}[\text{match } \tau \text{ with } \bar{\chi}]$	Premise
☞ $\phi[\alpha := \mathfrak{g}] \vdash \mathscr{C}[\text{match } \tau := \varsigma \text{ with } \bar{\chi}]$	Lemma D.7

Case $\Longleftarrow$.

$\mathscr{C}[\tau\,!\,\varsigma]$	Premise
$\phi[\alpha := \mathfrak{g}] \vdash \mathscr{C}[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}]$	Premise
☞ $\phi[\alpha := \mathfrak{g}] \vdash \mathscr{C}[\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}]$	By Susp-Ctx

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.

□

E 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.

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

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

Corollary E.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 D.2. □

Lemma E.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 E.2.

Case *n is $k+1$.*

Subcase $\Longrightarrow$.

$$\begin{array}{rll}
 & \phi \vdash \mathscr{C}[C_1] & \text{Premise} \\
 & \phi \Vdash \mathscr{C}[C_1] & \text{Theorem D.6} \\
 & \mathscr{C}'[\tau \,!\, \varsigma] & \text{Inversion of Can-Susp-Ctx} \\
 & \phi \Vdash \mathscr{C}'[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}] & '' \\
 & \mathscr{C}[C_1] = \mathscr{C}'[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}] & '' \\
 & = \mathscr{C}_2[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}, C_1] & \text{For some two-hole context } \mathscr{C}_2 \\
 & \phi \vdash \mathscr{C}_2[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}, C_2] & \text{By } \textit{i.h.} \\
 \text{For all } \phi', \mathfrak{g} & & \text{Defn of } \mathscr{C}'[\tau \,!\, \varsigma] \\
 & \phi' \vdash \lfloor \mathscr{C}_2[\tau := \mathfrak{g}, C_2] \rfloor & \text{Premise} \\
 & \phi' \vdash \lfloor \mathscr{C}_2[\tau := \mathfrak{g}, C_1] \rfloor & \text{Corollary E.2} \\
 & \phi' \vdash \lfloor \mathscr{C}'[\tau := \mathfrak{g}] \rfloor & \text{Above} \\
 \mathsf{shape}\ (\mathfrak{g}) = \varsigma & & \Longrightarrow \text{E on } \mathscr{C}'[\tau \,!\, \varsigma]
\end{array}$$

$$\begin{array}{ll} \mathscr{C}_2[\square, C_2][\tau \,!\, \varsigma] & \text{Above} \\ ☞\ \phi \vdash \mathscr{C}_2[\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}, C_2] & \text{By Susp-Ctx} \end{array}$$

Subcase $\Longleftarrow$.

Symmetric argument.

□

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

Proof. By induction on the structure of U. □

Definition E.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:

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

Definition E.6 (Match-closed). A predicate P on constraints is *match-closed* if, for all constraints $\bar{C}, \bar{C}'$, matches $\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}$ and shapes ς,

$$P(\bar{C}, \mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}, \bar{C}') \implies P(\bar{C}, \mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}, \bar{C}')$$

Lemma E.7 (Determines is match-closed). *C* determines *$\bar{\beta}$ is match-closed.*

Proof. Follows from the definition of C determines $\bar{\beta}$ and Lemma D.2. □

Lemma E.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 $Q(n) := \forall \bar{C}, \mathscr{C}.\ \#\mathsf{match}\ \mathscr{C} + \#\mathsf{match}\ \bar{C} = n \implies P(\bar{C}) \implies \mathscr{C}[\mathscr{C}_1[\bar{C}]] \equiv \mathscr{C}[\mathscr{C}_2[\bar{C}]]$.

Case *n is 0.*

$$\begin{array}{ll} \mathscr{C}, \bar{C}\ \text{simple} & \text{Premise (n is 0)} \\ ☞\ P(\bar{C}) \implies \mathscr{C}[\mathscr{C}_1][\bar{C}] \equiv \mathscr{C}[\mathscr{C}_2][\bar{C}] & \dagger \end{array}$$

Case *n is $k+1$.*

Subcase $\Longrightarrow$.

$$\begin{array}{rl} P(\bar{C}) & \text{Premise} \\ \phi \vdash \mathscr{C}[\mathscr{C}_1][\bar{C}] & \text{Premise} \\ \phi \Vdash \mathscr{C}[\mathscr{C}_1][\bar{C}] & \text{Theorem D.6} \\ \phi \Vdash \mathscr{C}'[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}] & \text{Inversion of Can-Susp-Ctx} \\ \mathscr{C}'[\tau \,!\, \varsigma] & '' \\ \mathscr{C}[\mathscr{C}_1][\bar{C}] = \mathscr{C}'[\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}] & '' \end{array}$$

Cases on $\mathscr{C}, \bar{C}$.

Subsubcase *$\mathscr{C}$ contains $\mathscr{C}'$'s hole.*

$$\begin{array}{rl} \mathscr{C}[\mathscr{C}_1][\bar{C}] = \mathscr{C}_3[\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}, \mathscr{C}_1[\bar{C}]] & \text{For some 2-hole context } \mathscr{C}_3 \\ \phi \Vdash \mathscr{C}_3[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}, \mathscr{C}_1[\bar{C}]] & \\ k = \#\mathsf{match}\ \mathscr{C}_3[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}, \mathscr{C}_1[\bar{C}]] & \\ \phi \vdash \mathscr{C}_3[\mathsf{match}\ \tau := \varsigma\ \mathsf{with}\ \bar{\chi}, \mathscr{C}_2[\bar{C}]] & \text{By } \textit{i.h.} \end{array}$$

For all $\phi', \mathfrak{g}$

$$\begin{array}{lll}
 & \phi' \vdash \lfloor \mathscr{C}_3[\tau = \mathfrak{g}, \mathscr{C}_2[\bar{C}]] \rfloor & \text{Premise} \\
 & \phi' \vdash \lfloor \mathscr{C}_3[\tau = \mathfrak{g}, \mathscr{C}_1[\bar{C}]] \rfloor & \dagger \\
\text{shape } (\mathfrak{g}) = \varsigma & & \Longrightarrow \text{E on } \mathscr{C}'[\tau \,!\, \varsigma] \\
 & \mathscr{C}_3[\square, \mathscr{C}_2[\bar{C}]][\tau \,!\, \varsigma] & \text{Above} \\
☞ & \phi \vdash \mathscr{C}_3[\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}, \mathscr{C}_2[\bar{C}]] & \text{By Susp-Ctx}
\end{array}$$

Subsubcase *C_i contains $\mathscr{C}'$'s hole.*

Similar argument to the above case, but relies on the match-closure of P.

Subcase $\Longleftarrow$.

Symmetric argument.

□

Lemma E.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 ($\dagger$):

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

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 E.3) to establish the result.
We proceed by induction on the structure of $\mathscr{C}'$ with the statement ($\ddagger$):

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

This holds due to the compositionality of simple equivalence using $\dagger$ as a base case.

Subcase $\Longrightarrow$.

$$\begin{array}{rl}
\phi \vdash \mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = \mathscr{C}[\bar{C}]\ \mathsf{in}\ \mathscr{C}'[C_1] & \text{Premise} \\
\phi \vdash \exists \alpha, \bar{\alpha}. \mathscr{C}[\bar{C}] & \text{Simple inversion} \\
\phi[x := \phi(\lambda\alpha[\bar{\alpha}].\mathscr{C}[\bar{C}])] \vdash \mathscr{C}'[C_1] & '' \\
\phi[x := \phi(\lambda\alpha[\bar{\alpha}].\mathscr{C}[\bar{C}])] \vdash \mathscr{C}'[C_2] & \ddagger \\
\phi \vdash \mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = \mathscr{C}[\bar{C}]\ \mathsf{in}\ \mathscr{C}'[C_2] & \text{By LetR}
\end{array}$$

Subcase $\Longleftarrow$.

Symmetric argument.

Case *n is $k + 1$.*

Analogous to the inductive step in Lemma E.8.

□

Lemma E.10. *If $\alpha = \tau = \epsilon \in \mathscr{C}$ and $\tau \notin \mathcal{V}$, then $\mathscr{C}[\alpha \,!\, \text{shape } (\tau)]$.*

PROOF.

$\alpha = \tau = \epsilon \in \mathscr{C}$	Premise
$\tau \notin \mathcal{V}$	Premise
$\tau = \text{shape}\ (\tau)\ \bar{\tau}$	For some $\bar{\tau}$
$\mathscr{C} = \mathscr{C}_1[\alpha = \tau = \epsilon \wedge \mathscr{C}_2]$	By definition
$\text{fv}(\alpha, \tau, \epsilon) \# \text{bv}(\mathscr{C}_2)$	$''$
For all $\phi, \mathfrak{g}$	Defn. of $\mathscr{C}[\alpha\,!\,\text{shape}\ (\tau)]$
$\phi \vdash \lfloor \mathscr{C}_1[\alpha = \text{shape}\ (\tau)\ \bar{\tau} = \epsilon \wedge \mathscr{C}_2[\alpha = \mathfrak{g}]] \rfloor$	Premise
$\phi_1 \vdash \alpha = \text{shape}\ (\tau)\ \bar{\tau} = \epsilon$	Inversion of $\mathscr{C}_1$
$\phi_2 \vdash \alpha = \mathfrak{g}$	Inversion of $\mathscr{C}_2$
$\mathfrak{g} = \phi_2(\alpha)$	Simple inversion
$= \phi_1(\alpha)$	$\alpha \# \text{bv}(\mathscr{C}_2)$
$= \text{shape}\ (\tau)\ \phi_1(\bar{\tau})$	Simple inversion
☞ $\text{shape}\ (\mathfrak{g}) = \text{shape}\ (\tau)$	Applying shape to both sides

□

LEMMA E.11. *If* $\gamma = \tau = \epsilon \in \mathscr{C}[\mathscr{C}_2]$ *and* $\tau \notin \mathcal{V}$, *then*

$$\mathscr{C}[\text{let}\ x\ \alpha\ [\bar{\alpha}] = \mathscr{C}_1[\square]\ \text{in}\ \mathscr{C}_2[i^x[\alpha' \rightsquigarrow \gamma]]][\alpha'\,!\,\text{shape}\ (\tau)]$$

PROOF. Similar proof to Lemma E.10. □

LEMMA E.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 E.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 $C_1 \equiv_{\text{ctx}} C_2$.

Case

$$\frac{U_1 \qquad U_1 \longrightarrow U_2}{U_2}\ \text{S-UNIF}$$

$U_1 \longrightarrow U_2$	Premise
$U_1 \equiv U_2$	Lemma E.12
U_1, U_2 simple	Lemma E.4
☞ $U_1 \equiv_{\text{ctx}} U_2$	Lemma E.3

Case

$$\frac{(\exists \alpha. C_1) \wedge C_2 \qquad \alpha \# C_2}{\exists \alpha. C_1 \wedge C_2}\ \text{S-EXISTS-CONJ}$$

$\alpha \# C_2$ Premise

Sufficient to show equivalence for simple constraints.	Lemma E.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

$$\frac{\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi} \qquad \tau \notin \mathcal{V}}{\mathsf{match}\ \tau := \mathsf{shape}\ (\tau)\ \mathsf{with}\ \bar{\chi}} \longrightarrow \textsc{S-Match-Type}$$

$\tau \notin \mathcal{V}$	Premise
$\square[\tau\ !\ \mathsf{shape}\ (\tau)]$	By definition
Sufficient to show equivalences between constraints.	Lemma D.3

Subcase $\Longrightarrow$.

For all ϕ

$\phi \vdash \mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}$	Premise
☞ $\phi \vdash \mathsf{match}\ \tau := \mathsf{shape}\ (\tau)\ \mathsf{with}\ \bar{\chi}$	Lemma D.7

Subcase $\Longleftarrow$.

For all ϕ

$\phi \vdash \mathsf{match}\ \tau := \mathsf{shape}\ (\tau)\ \mathsf{with}\ \bar{\chi}$	Premise
☞ $\phi \vdash \mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}$	By Susp-Ctx

Case

$$\frac{\mathscr{C}[\mathsf{match}\ \alpha\ \mathsf{with}\ \bar{\chi}] \qquad \alpha = \tau = \epsilon \in \mathscr{C}}{\mathscr{C}[\mathsf{match}\ \alpha := \mathsf{shape}\ (\tau)\ \mathsf{with}\ \bar{\chi}]} \longrightarrow \textsc{S-Match-Var}$$

$\alpha = \tau = \epsilon \in \mathscr{C}$	Premise
$\mathscr{C}[\alpha\ !\ \mathsf{shape}\ (\tau)]$	Lemma E.10
Sufficient to show equivalences between constraints.	Lemma D.3

Subcase $\Longrightarrow$.

For all ϕ

$\phi \vdash \mathscr{C}[\mathsf{match}\ \alpha\ \mathsf{with}\ \bar{\chi}]$	Premise
☞ $\phi \vdash \mathscr{C}[\mathsf{match}\ \alpha := \mathsf{shape}\ (\tau)\ \mathsf{with}\ \bar{\chi}]$	Lemma D.7

Subcase $\Longleftarrow$.

For all ϕ

$\phi \vdash \mathscr{C}[\mathsf{match}\ \alpha := \mathsf{shape}\ (\tau)\ \mathsf{with}\ \bar{\chi}]$	Premise
☞ $\phi \vdash \mathscr{C}[\mathsf{match}\ \alpha\ \mathsf{with}\ \bar{\chi}]$	By Susp-Ctx

Case

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

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

Sufficient to show equivalence between $x\ \tau$ and $\exists \gamma, i^x.\, \gamma = \tau \wedge i[\alpha \rightsquigarrow \gamma]$. Lemma E.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 Partial-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

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

$x \# \mathrm{bv}(\mathscr{C})$ 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 E.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 Partial-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

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

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 E.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

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

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

Sufficient to show equivalence between $i^x[\alpha' \rightsquigarrow \gamma]$ and true. Lemma E.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
☞	$\phi' \vdash \mathsf{true}$	By True

Subcase $\Longleftarrow$.

$\phi' \vdash \mathsf{true}$	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 Partial-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 $\exists i^x$.
$\phi_3(i) = \phi_1$	″
$\phi_3[i := \phi_2] \vdash \mathscr{C}_2[\text{true}]$	$i.\alpha' \# \mathsf{insts}(\mathscr{C}_2)$
$\mathscr{D} :: \phi_3 \vdash \mathscr{C}_2[\text{true}]$	By Exists-Inst

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

☞ So this case degenerates to the former case.

Case

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

$\beta \# \alpha, \bar{\alpha}$	Premise
$x, \beta \# \mathsf{bv}(\mathscr{C})$	Premise

Sufficient to show equivalence between $i^x[\beta \rightsquigarrow \gamma]$ and $\beta = \gamma$.	Lemma E.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
$\phi_1(\beta) = \phi(\beta)$	$\beta \# \alpha, \bar{\alpha}$
$\phi'(\beta) = \phi(\beta)$	$\beta \# \mathsf{bv}(\mathscr{C})$
$\phi'(\gamma) = \phi'(\beta)$	Above
$\phi' \vdash \gamma = \beta$	By Unif

Subcase $\Longleftarrow$.

Symmetric argument.

Case

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

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

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

Subcase $\Longrightarrow$.

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

Subcase $\Longleftarrow$.

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

Case

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

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

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

$$
\frac{\begin{array}{c}\mathscr{C}[\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = \mathscr{C}_1[\mathsf{match}\ \alpha'\ \mathsf{with}\ \bar{\chi}]\ \mathsf{in}\ \mathscr{C}_2[i^x[\alpha' \rightsquigarrow \gamma]]] \\ \alpha' \in \alpha, \bar{\alpha} \qquad \gamma = \tau = \epsilon \in \mathscr{C}[\mathscr{C}_2] \qquad x \# \mathsf{bv}(\mathscr{C}_2)\end{array}}{\mathscr{C}[\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = \mathscr{C}_1[\mathsf{match}\ \alpha' := \mathsf{shape}\ (\tau)\ \mathsf{with}\ \bar{\chi}]\ \mathsf{in}\ \mathscr{C}_2[i^x[\alpha' \rightsquigarrow \gamma]]]} \text{ S-BackProp}
$$

Similar argument to S-Match-Var, using Lemma E.11.

Case *S-Compress*, *S-Gc*, *S-Exists-All*, *S-All-Escape*, *S-All-Rigid*, *S-All-Solve*.

Similar argument. Use Lemma E.8. The simple equivalences are standard, see Pottier and Rémy [2005].

□

E.2 Progress

Lemma E.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 E.15 (Progress). *If constraint C cannot take a step $C \longrightarrow C'$, then either:*

(i) C *is solved.*
(ii) C *is stuck, it is either: (a)* false*; (b)* $\hat{\mathscr{C}}[x\ \tau]$ *where* $x \# \hat{\mathscr{C}}$*; (c)* $\hat{\mathscr{C}}[i^x[\alpha \rightsquigarrow \gamma]]$ *where* $x \# \hat{\mathscr{C}}$ *and* $i.\alpha \# \mathsf{insts}(\hat{\mathscr{C}})$*; (d) for every match constraint* $\hat{\mathscr{C}}[\mathsf{match}\ \alpha\ \mathsf{with}\ \bar{\chi}]$ *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 $\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}$. We have two cases:

Subcase *τ is a non-variable type.* Apply S-Match-Type.

Subcase *τ is a type variable α.*

We have $\square[\alpha\, \xcancel{!}]$. It suffices that every match constraint in a context-reachable position $\hat{\mathscr{C}}[\mathsf{match}\ \alpha'\ \mathsf{with}\ \bar{\chi}]$ 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 $\mathsf{match}\ \tau\ \mathsf{with}\ \bar{\chi}$ is stuck.

Case $C_1 \wedge C_2$. We begin by inducting on C_1 and C_2. Then we consider cases:

Subcase *C_1 (or C_2) take a step.* Apply congruence rewriting rule.

Subcase *C_1 (or C_2) is* true. Apply S-True.

Subcase *C_1 (or C_2) is* false. Apply S-False.

Subcase *C_1 (or C_2) begins with $\exists$.* Apply S-Exists-Conj.

Subcase *C_1, C_2 are solved.*

We either apply the above $\exists$ case, or both C_1 and C_2 are solved multi-equations $\bar{\epsilon}_1, \bar{\epsilon}_2$. We perform cases on this:

Subsubcase *$\bar{\epsilon}_1$ and $\bar{\epsilon}_2$ are mergable.* Apply U-Merge.

Subsubcase $\mathsf{cyclic}\ (\bar{\epsilon}_1, \bar{\epsilon}_2)$. Apply U-Cycle.

Subsubcase *Otherwise.* The conjunction $\bar{\epsilon}_1 \wedge \bar{\epsilon}_2$ is solved.

Subcase *C_1 and C_2 are stuck (and not* false*).*

w.l.o.g., consider cases C_1.

Subsubcase $\hat{\mathscr{C}}_1[x\ \tau]$. We have $x \# \mathsf{bv}(\hat{\mathscr{C}}_1)$.

$\hat{\mathscr{C}}_1[x\ \tau] \wedge C_2$ is stuck as we do not bind x in $\hat{\mathscr{C}}_1 \wedge C_2$.

Subsubcase $\hat{\mathscr{C}}_1[i^x[\alpha \rightsquigarrow \gamma]]$. We have $x \# \mathsf{bv}(\hat{\mathscr{C}}_1)$ and $i.\alpha \# \mathsf{insts}(\hat{\mathscr{C}}_1)$.

If $i.\alpha \in \mathsf{insts}(C_2)$ and $i \# \mathsf{bv}(\hat{C}_1)$, then apply S-Inst-Unify. It must be the case that we can apply S-Inst-Unify, otherwise, we could lift these instantiation constraints using S-Exists-Lower and S-Let-ConjLeft, contradicting that $\hat{\mathscr{C}}_1$ is stuck.

Otherwise, $x \# \mathsf{bv}(\hat{\mathscr{C}}_1 \wedge C_2)$, thus $\hat{\mathscr{C}}_1[i^x[\alpha \rightsquigarrow \gamma]]$ is stuck.

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

Consider a match constraint $\mathsf{match}\ \alpha'\ \mathsf{with}\ \bar{\chi}$ in C_1.

If $\alpha' = \tau = \epsilon \in C_2$ and $\tau \notin \mathcal{V}$. By the above logic, it must be at the root (otherwise C_2 is not stuck). So we have $\alpha' = \tau = \epsilon \in \hat{\mathscr{C}}_1 \wedge C_2$. Thus we can apply S-Match-Type.

If $\gamma = \tau = \epsilon \in C_2$, $\tau \notin \mathcal{V}$, and $\hat{\mathscr{C}}_1$ contains $\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = \hat{\mathscr{C}}_3[\mathsf{match}\ \alpha'\ \mathsf{with}\ \bar{\chi}]\ \mathsf{in}\ \hat{\mathscr{C}}_4[i^x[\alpha' \rightsquigarrow \gamma]]$. Apply S-BackProp.

Otherwise, we are stuck and $(\mathscr{C}_1 \wedge C_2)[\alpha' \not\times]$.

Case $\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = C_1\ \mathsf{in}\ C_2$. We begin by inducting on C_1 and C_2. Then we consider cases:

Subcase C_1 *(or C_2) take a step.* Apply congruence rewriting rule.

Subcase C_1 *(or C_2) is* false. Apply S-False.

Subcase C_1 *begins with* $\exists$. Apply S-Let-ExistsLeft

Subcase C_2 *begins with* $\exists$. Apply S-Let-ExistsRight

Subcase C_2 *begins with* $\wedge$ *with* $x\ \#$ *from conjunct*. Apply S-Let-ConjRight.

Subcase C_1 *begins with* $\wedge$ *with* $\alpha, \bar{\alpha}\ \#$ *from conjunct* . Try apply S-Let-ConjLeft

Subcase C_2 *begins with* $\exists i^{x'}.\,, x \neq x'$. Apply S-Exists-Inst-Let

Subcase $\alpha' \in \bar{\alpha}$ *is determined by* C_1. Apply S-Exists-Lower

Subcase C_2 *is solved.*

Thus C_2 must be true (due to above cases).

Subsubcase C_1 *is solved*. Thus C_1 must be $\bar{\epsilon}$.

There are two cases:

- $\exists \alpha, \bar{\alpha}.\, \bar{\epsilon} \equiv \mathsf{true}$. Apply S-Let-Solve.
- $\exists \alpha, \bar{\alpha}.\, \bar{\epsilon} \not\equiv \mathsf{true}$. It must be the case there is some β that dominates a α' in $\alpha, \bar{\alpha}$ in $\bar{\epsilon}$. Hence $\exists \alpha, \bar{\alpha} \setminus \alpha'.\, \bar{\epsilon}$ determines α'. So we can apply S-Exists-Lower.

Subsubcase C_1 *is stuck.*

The constraint $\mathsf{let}\ x\ \alpha\ [\bar{\alpha}] = C_1\ \mathsf{in}\ C_2$ remains stuck, since no additional term variable bindings occur for the scope of C_1, ruling out the instantiation cases. Additionally, we cannot apply backpropagation since C_2 is true.

Subcase C_2 *is stuck.*

Subsubcase $\hat{\mathscr{C}}[x\ \tau]$. We have $x \# \mathsf{bv}(\hat{\mathscr{C}})$.

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 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 (S-BackProp) applies with an equation in $\hat{\mathscr{C}}$, or the entire constraint is stuck.
- $\alpha' \notin \alpha, \bar{\alpha}$. Apply S-Inst-Mono.

Subsubcase *For any* $\hat{\mathscr{C}}[\mathsf{match}\ \alpha'\ \mathsf{with}\ \bar{\chi}]$. We have $\hat{\mathscr{C}}[\alpha' \not\times]$.

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.

□

E.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 E.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:

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

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

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

We have:

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

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

Case

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

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

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

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 E.17 (TERMINATION). *The constraint solver terminates on all inputs.*

PROOF. The difficulty for termination comes from the "discharge" rules S-MATCH-TYPE, S-MATCH-VAR 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-partial instantiations $x\ \tau$ decreases strictly on S-LET-APPR when a partial 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 partial 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:

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

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

Case

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

We aim to show that the weight of the rewritten constraint $\exists \bar{\beta}'.\, \gamma = \varsigma\,\bar{\beta}' \wedge i^x[\bar{\beta} \rightsquigarrow \bar{\beta}']$ is strictly less than the original $i^x[\alpha' \rightsquigarrow \gamma]$.

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

To ensure a strict decrease, it suffices to show that sw $(\varsigma) > |\bar{\beta}'|$. Given that $|\bar{\beta}'| = |\varsigma|$, and by the definition of sw (ς), this inequality holds. Therefore, the weight strictly decreases under S-Inst-Copy.

Thus the constraint solver terminates. □

E.4 Correctness

Lemma E.18. *Given non-simple C constraint. If every match constraint $\mathscr{C}[\text{match } \tau \text{ with } \bar{\chi}] = C$ satisfies $\mathscr{C}[\tau \not\times]$, then C is unsatisfiable.*

Proof. By contradiction, inverting on the canonical derivation of C. □

Corollary E.19. *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 equivalence to C.*

Proof. We show each direction individually:

Case $\Longrightarrow$.

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

We have C is satisfiable. By Theorem E.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* $\hat{\mathscr{C}}[\text{match } \tau \text{ with } \bar{\chi}]$ and $\mathscr{C}[\tau \not\times]\varsigma$ for any shape ς.

If C is false, this contradicts our assumption that C is satisfiable. Similarly, by Lemma E.18, if C is $\hat{\mathscr{C}}[\text{match } \tau \text{ with } \bar{\chi}]$, then this also contradicts the satisfiability of C.

Subcase $C \longrightarrow C'$.

By Theorem E.13, we have $C \equiv C'$, thus C' is satisfiable. So by induction, we have $C' \longrightarrow^* \hat{C}$ and $\hat{C}$ is a solved form equivalent to C'. By transitivity of equivalence, we therefore have $\hat{C} \equiv C$, as required.

Case $\Longleftarrow$.

By induction on the rewriting $C \longrightarrow^* \hat{C}$.

Subcase

$$\frac{}{\hat{C} \longrightarrow^* \hat{C}}\ \textsc{Zero-Step}$$

We have $C = \hat{C}$ by inversion. All solved forms are satisfiable, thus C is satisfiable.

Subcase

$$\dfrac{C \longrightarrow C' \qquad C' \longrightarrow^{*} \hat{C}}{C \longrightarrow^{*} \hat{C}}\ \textsc{One-Step}$$

By induction, we have C' is satisfiable. By Theorem E.13, $C \equiv C'$, hence C is satisfiable.

□

F 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.

F.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 F.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 F.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. □

The simple syntax-directed system has an inversion lemma:

Lemma F.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 $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} \{\overline{l = e}\} : \tau$, then $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} l_i = e_i : \tau$ for $1 \leq i \leq n$ and $\bar{l}\ !\ \tau$.

(viii) If $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} l = e : \tau$, then $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} e : \tau'$ and $l : \tau' \to \tau$.

(ix) If $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} e.l : \tau$, then $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} e : \tau'$, $l : \tau' \to \tau$ and $l\ !\ \tau$.

(x) If $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} (e_1, \dots, e_n) : \tau$, then $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} e_i : \tau_i$ for all $1 \leq i \leq n$ and $\tau = \Pi_{i=1}^{n} \tau_i$.

(xi) If $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} e.j/n : \tau$, then $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} e : \Pi_{i=1}^{n} \tau_i$ and $\tau = \tau_j$, with $n \geq j$.

(xii) If $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} [e : \exists \bar{\alpha}.\, \forall \bar{\beta}.\, \tau'] : \tau$, then $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} e : \tau[\bar{\alpha} := \bar{\tau}]$, $\bar{\beta} \# \Gamma$ and $\tau = [\forall \bar{\beta}.\, \tau'][\bar{\alpha} := \bar{\tau}]$.

(xiii) 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 $\sigma \leq \tau$.

(xiv) If $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} \{e\} : \tau$, then $\Gamma \vdash^{\mathsf{sd}}_{\mathsf{simple}} e : \tau'$.

F.2 Canonicalization of typability

Our system satisfies a similar canonicalization theorem to constraint satisfiability.

Lemma F.4 (Composability of unicity).

(i) *If* $\mathscr{E}_1[e \triangleleft \varsigma]$, *then* $\mathscr{E}_2[\mathscr{E}_1][e \triangleleft \varsigma]$.
(ii) *If* $\mathscr{E}_1[e \triangleright \varsigma]$, *then* $\mathscr{E}_2[\mathscr{E}_1][e \triangleright \varsigma]$.
(iii) *If* $\mathscr{L}[\ell \,!\, \mathsf{t}]$, *then* $\mathscr{E}_2[\mathscr{L}][\ell \,!\, \mathsf{t}]$.

Proof. By induction on $\mathscr{E}_2$. □

Lemma F.5 (Decanonicalization). *If* $\Vdash e : \tau$, *then* $\emptyset \vdash e : \tau$.

Proof. By induction on the given derivation $\Vdash e : \tau$. □

Theorem F.6 (Canonicalization). *If* $\vdash e : \sigma$, *then* $\Vdash e : \tau$ *for any instance* τ *of* σ.

Proof. By induction on the following measure of e:

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

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

(1) $\#\mathsf{implicit}\ e$ is the number of implicit constructs in e *i.e.*, overloaded tuple projections $e.j$, field projections $e.\ell$, 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. □

F.3 Unifiers

A substitution ϑ is an idempotent function from type variables to types. The (finite) domain of ϑ is the set of type variables such that $\vartheta(\alpha) \neq \alpha$ for any $\alpha \in \mathsf{dom}\ \vartheta$, while the codomain consists of the free type variables of its range. We use the notation $[\bar{\alpha} := \bar{\tau}]$ for the substitution ϑ with domain $\bar{\alpha}$ and $\vartheta(\bar{\alpha}) = \bar{\tau}$.

The constraint induced by a substitution ϑ, written $\exists\vartheta$, is $\exists\bar{\beta}.\, \bar{\alpha} = \bar{\tau}$ where $\bar{\beta} = \mathsf{rng}\ \vartheta$, $\bar{\alpha} = \mathsf{dom}\ \vartheta$ and $\vartheta(\bar{\alpha}) = \bar{\tau}$.

Definition F.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 F.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 F.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 F.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 F.9 and simple inversion. $\square$

Lemma F.11. *For two substitutions ϑ, ϑ'. If $\exists\vartheta \vDash \exists\vartheta'$, there exists ϑ'' such that $\vartheta = \vartheta'' \circ \vartheta'$.*

Proof. Standard result, follows from definition of $\exists\vartheta$. $\square$

F.4 Soundness and completeness of constraint generation

Lemma F.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 $\mathscr{E}$. $\square$

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

Proof. By induction on e. $\square$

Lemma F.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. $\square$

Theorem F.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.

$$\begin{array}{lrcll}
 & & & e\ \text{simple} & \text{Premise} \\
 & \emptyset \vdash^{\mathsf{sd}}_{\mathsf{simple}} e : \vartheta(\alpha) & \iff & \vartheta\ \text{unifies}\ [\![e : \alpha]\!] & \text{Lemma F.14} \\
 & \emptyset \vdash^{\mathsf{sd}}_{\mathsf{simple}} e : \vartheta(\alpha) & \iff & \Vdash e : \vartheta(\alpha) & \text{When } e \text{ simple} \\
☞ & \Vdash e : \vartheta(\alpha) & \iff & \vartheta\ \text{unifies}\ [\![e : \alpha]\!] & \text{Above}
\end{array}$$

Case *n is* $k + 1$.

Subcase $\implies$.

Subsubcase

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

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

$$
\begin{array}{rll}
 & = \llbracket \{(e : \mathfrak{g})\} : \beta \rrbracket & '' \\
\lfloor \mathscr{C}[\alpha_1 = \mathfrak{g}] \rfloor & = \lfloor \text{let } \Gamma \text{ in } \llbracket \mathscr{E}[\square : \beta] : \alpha \rrbracket [\llbracket \{(e : \mathfrak{g})\} : \beta \rrbracket] \rfloor & '' \\
 & = \lfloor \text{let } \Gamma \text{ in } \llbracket \mathscr{E}[\{(e : \mathfrak{g})\}] : \alpha \rrbracket \rfloor & \text{Lemma F.12} \\
 & = \text{let } \Gamma \text{ in } \lfloor \llbracket \mathscr{E}[\{(e : \mathfrak{g})\}] : \alpha \rrbracket \rfloor & \text{By definition} \\
 & = \text{let } \Gamma \text{ in } \llbracket \lfloor \mathscr{E}[\{(e : \mathfrak{g})\}] \rfloor : \alpha \rrbracket & \text{Lemma F.13} \\
\phi & \text{ unifies let } \Gamma \text{ in } \llbracket \lfloor \mathscr{E}[\{(e : \mathfrak{g})\}] \rfloor : \alpha \rrbracket & \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 F.5} \\
\text{shape } (\mathfrak{g}) & = \nu\bar{\gamma}.\, \Pi_{i=1}^{n} \bar{\gamma} & \Longrightarrow \text{E} \\
 & \mathscr{C}[\alpha_1 \,!\, \nu\bar{\gamma}.\, \Pi_{i=1}^{n} \bar{\gamma}] & \text{Above} \\
\vartheta & \text{ unifies } \mathscr{C}[\text{match } \alpha_1 \text{ with } \Pi\, \gamma_j \to \beta = \gamma] & \text{By Susp-Ctx} \\
\llbracket e.j : \beta \rrbracket & = \exists \alpha_1.\, \llbracket e : \alpha_1 \rrbracket \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 } \llbracket \mathscr{E}[\square : \beta] : \alpha \rrbracket [\exists \alpha_1.\, \llbracket e : \alpha_1 \rrbracket \wedge \ldots] & '' \\
 & = \text{let } \Gamma \text{ in } \llbracket \mathscr{E}[\square : \beta] : \alpha \rrbracket [\llbracket e.j : \beta \rrbracket] & \text{Above} \\
 & = \text{let } \Gamma \text{ in } \llbracket \mathscr{E}[e.j] : \alpha \rrbracket & \text{Lemma F.12} \\
 & = \llbracket \mathscr{E}[e.j] : \alpha \rrbracket & \\
☞ \quad \vartheta & \text{ unifies } \llbracket \mathscr{E}[e.j] : \alpha \rrbracket &
\end{array}
$$

Subsubcase *Can-Poly-I*, *Can-Use-I*, *Can-Lab-I*.

Similar arguments.

Subcase $\Longleftarrow$.

Subsubcase

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

$$
\begin{array}{rll}
\llbracket e : \tau \rrbracket & = \text{let } \Gamma \text{ in } \llbracket \mathscr{E}[e.j] : \alpha \rrbracket & \text{Premise} \\
\mathscr{C} & = \text{let } \Gamma \text{ in } \llbracket \mathscr{E}[\square : \beta] : \alpha \rrbracket [\exists \alpha.\, \llbracket e : \alpha \rrbracket \wedge \square] & \text{Premise} \\
\vartheta & \text{ unifies } \mathscr{C}[\text{match } \alpha_1 := \nu\bar{\gamma}.\, \Pi_{i=1}^{n} \bar{\gamma} \text{ with } \ldots] & \text{Premise} \\
\vartheta & \text{ unifies } \llbracket \mathscr{E}[e.j/n] : \alpha \rrbracket & \text{Above (See } \Longrightarrow \text{ direction)} \\
 & \Vdash \mathscr{E}[e.j/n] : \vartheta(\alpha) & \text{By } \textit{i.h.} \\
\Gamma' & \vdash \mathscr{E}[\{(e : \mathfrak{g})\}] : \tau' & \text{Premise} \\
\Gamma' & = \emptyset & \mathscr{E}[\{(e : \mathfrak{g})\}] \text{ is closed} \\
 & \Vdash \mathscr{E}[\{(e : \mathfrak{g})\}] : \tau' & \text{Lemma F.5} \\
[\alpha := \tau'] & \text{ unifies } \llbracket \mathscr{E}[\{(e : \mathfrak{g})\}] : \alpha \rrbracket & \text{By } \textit{i.h.} \\
\phi[\alpha := \phi(\tau')] & \vdash \llbracket \mathscr{E}[\{(e : \mathfrak{g})\}] : \alpha \rrbracket & \text{By definition} \\
 & \mathscr{C}[\alpha_1 \,!\, \nu\bar{\gamma}.\, \Pi_{i=1}^{n} \bar{\gamma}] & \text{Premise} \\
\text{shape } (\mathfrak{g}) & = \nu\bar{\gamma}.\, \Pi_{i=1}^{n} \bar{\gamma} & \Longrightarrow \text{E} \\
 & \mathscr{E}[e \triangleright \nu\bar{\gamma}.\, \Pi_{i=1}^{n} \bar{\gamma}] & \text{Above} \\
 & \Vdash \mathscr{E}[e.j] : \vartheta(\alpha) & \text{By Can-Proj-I}
\end{array}
$$

Subsubcase $[e]$, $\langle e \rangle$, ℓ.

Similar arguments.

$\square$

F.5 Principal types

Theorem F.16 (Principal types). *For any well-typed closed term e, there exists a type τ, which we call principal, 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 F.15, the constraint $[\![e : \alpha]\!]$ is satisfiable (specifically under the unifier $\alpha = \tau$). By Corollary E.19, 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 F.15 that $\vdash e : \vartheta(\alpha)$, proving (i). For (ii), suppose $\vdash e : \tau'$ for some τ'. Then by Theorem F.15 again, there exists a unifier ϑ' of $[\![e : \alpha]\!]$ such that $\vartheta'(\alpha) = \tau'$. Since ϑ is most general, we have $\exists\vartheta' \models \exists\vartheta$, and by Lemma F.11, this implies the existence of a substitution ϑ'' such that $\vartheta' = \vartheta'' \circ \vartheta$. Hence, $\tau' = \vartheta'(\alpha) = \vartheta''(\vartheta(\alpha))$, witnessing that τ' is an instance of $\vartheta(\alpha)$, as required (ii). □

Contents