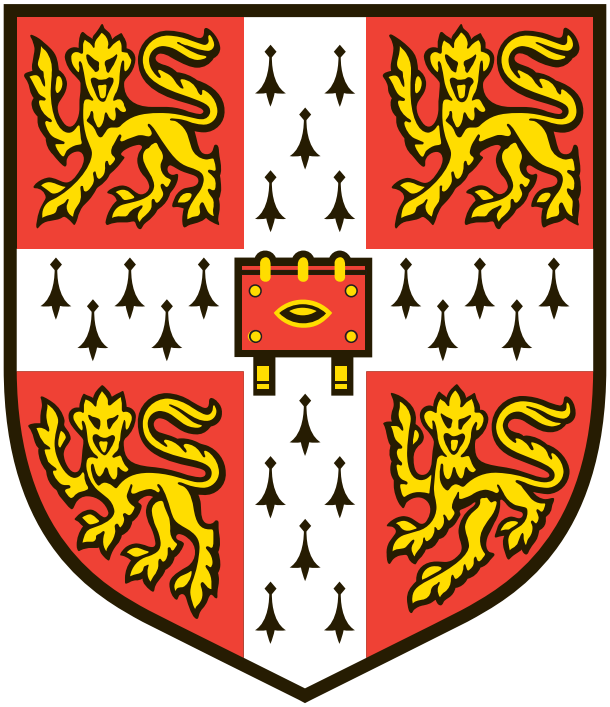
Alistair O'Brien

Typing OCaml in OCaml: a constraint-based approach

PhD Proposal



Queens' College

October 15, 2025

First year report submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy

Contents

1 Introduction

Type systems are widely valued by programmers for their ability to catch common programming errors at compile time, giving the assurance of *type safety*: "well-typed programs cannot go wrong" [20]. The recent trend in retrofitting type systems to dynamically typed languages (TypeScript for JavaScript, Hack for PHP, Typed Racket) reflects the growing demand for static guarantees in mainstream, industrial settings.

However, while static typing provides robustness, it often comes at the cost of verbosity. Programmers must explicitly annotate their code with types, which can obscure the program's structure and makes writing and maintaining code more tedious. Type inference algorithms alleviate this issue by inferring the type annotations, allowing programmers to benefit from static checks without incurring the annotation overhead.

Type inference is particularly powerful in functional programming languages, which emphasise abstraction, composition, and immutability. These features naturally lead to polymorphic and higher-order code, where explicit types would be prohibitively verbose.

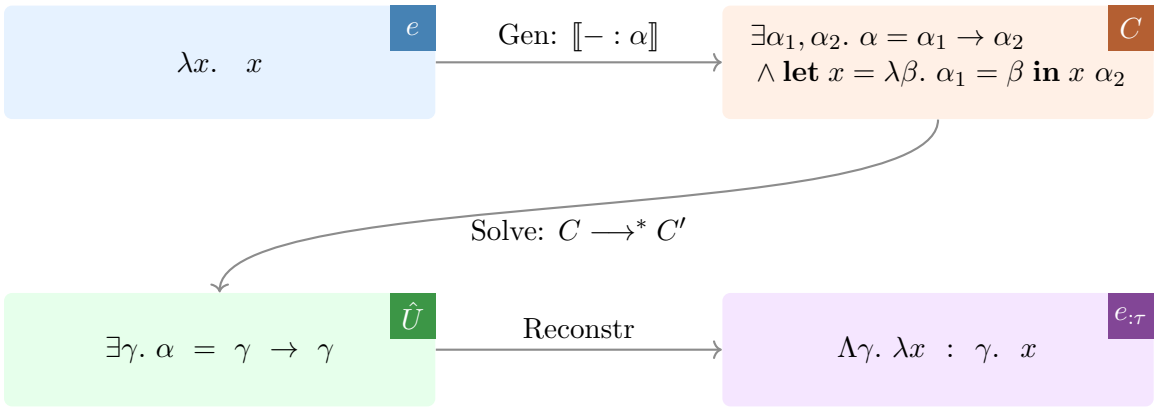
Although functional languages such as Haskell and OCaml remain relatively niche in terms of widespread industry use, many of their key ideas – higher-order functions, immutability, algebraic data types, and pattern matching – have found their way into mainstream languages such as Rust, Scala, Swift, and even TypeScript. As these features go mainstream, so does the need for powerful type inference to support them – making research in this area more relevant than ever!

HM type inference Many functional languages employ type inference algorithms derived from the Hindley-Milner type system [20, 13]. This system supports implicit *let-polymorphism* and is traditionally implemented by algorithm $\mathcal{W}$ [20] or its variants such as $\mathcal{J}$ [20]. These algorithms use unification and partial substitutions to resolve equalities between types during inference.

While elegant and effective in their original setting, they become significantly more complex when extended to support richer type system features. The added complexity often translates into in large, intricate, and difficult-to-maintain type checker implementations.

We propose an alternative approach: *constraint-based type inference*. Rather than interleaving inference with unification in the style of $\mathcal{W}$, we decouple the process into

distinct phases: constraint generation, constraint solving and type reconstruction. The idea, due to Pottier and Rémy [29], is deceptively straightforward: for some arbitrary term e, we generate a constraint C such that e is well-typed if and only if C is satisfiable. Once C has been solved, we construct an explicitly typed version of e, written $e_{:\tau}$ using C's solution.



This decomposition improves modularity with the separation of concerns, facilitates formal reasoning, and opens the door to improved optimisations.

OCaml OCaml (originally Objective Caml) is a general purpose, high-level programming language combining functional, object-oriented and imperative paradigms. Based of Milner's ML calculus [20], it has a sophisticated type system with one of the most complex and efficient type inference algorithms in existence.

The language can be seen as an amalgam of three largely orthogonal sublanguages:

Core ML An ML language with a wide range of features, including mutually recursive let-bindings, algebraic data types, patterns, mutable references, exceptions, type annotations and type abbreviations.

Modules A powerful module system is used to divide large OCaml programs into modules, each module consisting of a set of types and values. Functors allow parameterisation of modules, functioning as dependent functions from modules to modules. Functors are classified as either *generative* (generating abstract types) [37] or *applicative* (preserving and propagating abstract type equalities) [18]. Additionally, modules may also be treated as first-class citizens in the core language (so-called *first-class modules* [37]).

Objects An extension introduced by Rémy and Vouillon [35] that adds class-based objects using row polymorphism. Subsequently, extended by Garrigue [10] to accommodate polymorphic methods using semi-explicit first-class polymorphism. This extension also introduces explicit subtyping and variance annotations.

OCaml continues to evolve through additional extensions, including generalised algebraic data types (GADTs) [9], constructor and record label overloading, polymorphic variants

[6], labelled arguments [7], polymorphic parameters [51], and modal types [19].

OCaml's inference algorithm relies on a variant of algorithm $\mathcal{J}$, extended with Rémy's [33] efficient rank-based generalisation. Though performant, the implementation has grown complex and fragile due to: **(1)** the use of mutable state and imperative algorithms; **(2)** the tight coupling of inference, generalisation, and type reconstruction; **(3)** the difficulty of integrating newer language features into this framework.

These limitations make OCaml an ideal candidate for a constraint-based for type inference. Alleviating many of these engineering challenges.

Thesis aim Our central goal is to explore and validate constraint-based inference for a substantial fragment of OCaml, incorporating many of the language's most advanced and *fragile* features. Specifically, we aim to:

(1) Define a constraint language and solver suitable for: **(i)** GADTs, **(ii)** polymorphic variants, **(iii)** objects and classes, **(iv)** static overloading, **(v)** semi-explicit first-class polymorphism, **(vi)** polymorphic parameters.
(2) Develop the calculus and meta-theory of the above features, proving important properties such as soundness and completeness of inference and principal types.

This work aims to lay the foundation for the long-overdue rewrite of the type checker for OCaml and potentially influence the design of future type inference systems in other languages.

Organisation This report is structured as follows:

(1) Chapter 2 presents *omnidirectional type inference*, a novel technique for extensions of ML that often break principality such as static overloading and semi-explicit first-class polymorphism.
(2) Chapter 3 presents our initial work on a constraint-based approach for the type inference of GADTs, a notoriously difficult problem.
(3) Finally, Chapter 4 outlines the next stages of our project and presents a timeline for future work.

Remark 1 (The absence of a literature review)**.** *A staggering amount of work has been done on the type inference for advanced features of OCaml. For space reasons, we cannot offer a comprehensive review of the literature. Instead, we provided targeted reviews of relevant literature for the topics discussed in Chapters 2 and 3. In the present section, we offer a high-level overview of OCaml – not as a complete review, but to situate our work within the broader design landscape.*

2 Omnidirectional type inference

The HM type system has historically occupied sweet spot in the design space of strongly typed languages, owing to its *principal types* property: every well-typed term e has a most general type σ from which all other valid types for e are instances of σ. Yet this very property becomes an obstacle for many extensions. Some extensions, such as structural polymorphism, higher-kinded types, or type classes, preserve principality without issue. Others, including GADTs [47, 11], higher-rank polymorphism [10, 46, 40], or overloading of constructors and record fields[1], are more *fragile*: they demand explicit type annotations and force us to twist the ML specification with some algorithmic flavour to preserve *principality*.

We call such extensions *fragile* as they achieve completeness only with respect to a more or less ad-hoc specification that is a restriction of a more natural specification – designed especially to reject programs that would otherwise compromise the completeness of inference.

Existing approaches rely on a fixed inference order to propagate both inferred and annotated types as *known*, aiding disambiguation and allowing some annotations to be elided. In contrast, our *omnidirectional* approach lifts this ordering restriction, enabling more flexible and principal inference for these fragile extensions.

Bidirectional typechecking *Bidirectional typechecking* separates typing rules into two *modes*: checking ($\Gamma \vdash e \Leftarrow \tau$), which verifies a term e against a known type τ, and inference ($\Gamma \vdash e \Rightarrow \tau$), which infers the type of e.

Popularised by Pierce and Turner [26] it is widely used in languages featuring higher-rank polymorphism, dependent types, or subtyping. Bidirectional systems assign each syntactic construct a fixed mode – either checking or inference – dictating the flow of type information.

This fixed directionality, however, limits expressiveness. For instance, function applica-

[1]No citation is given since the current specification for constructor and record label overloading in OCaml is folklore.

tions admits two *mode-correct* rules:

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

In $(\lambda f.\ f\ ())\ \ (\lambda x.\ x : 1 \to 1)$, the argument's type dictates the function's type, a flow only permitted by Chk-App. Syn-App fails here, as the function cannot be typed independently of its argument.

Neither rule is optimal – each excels in contexts where the other fails – yet bidirectional typing forces us to irrevocably commit to one of the rules[2]. This fixed assignment is the core limitation of bidirectional typechecking. Principality may still hold, but only relative to a specification that made non-principal choices to begin with.

π-directional inference OCaml employs different mechanism centred on *polymorphism*, enforcing an ordering where let-bound terms are typed before their uses. We call this *π-directional* ("**pi**-directional") inference, reflecting that **p**olymorphism terms must be typed before their **i**nstances.

Originally developed for semi-explicit first-class polymorphism, it has also been used for the overloading of record fields, which we use for illustration here, as it is simpler[3]. The user may define nominal records with overlapping fields:

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

This can be thought as defining two abstract types `one, two` with overloaded access operators on `y`. Since both types are available in the current scope, the compiler must *statically* choose one of them. The question is which one? Consider:

```
let e0 r = r.x
let e1 r = r.y
let e2 = let r = { x = 1337; y = 42 } in r.x
let e3 = (fun r → r.y) { x = 1337; y = 42 }
```

To resolve the overloading, the type of the record should be *known* to be `one` or `two`. In e_0, this is clear: only `one` has the field `x`. In e_1, the type of `r` is unconstrained and thus ambiguous, so the expression fails to typecheck[4]. In both e_2 and e_3, there is a unique type for `r`. Yet, in OCaml, only the former is considered to be unambiguous.[5]

[2]Unless backtracking is added to the inference algorithm.

[3]We detail semi-explicit first-class polymorphism in Section 2.1.1.

[4]In fact, OCaml uses a default resolution strategy instead of failing when the type is ambiguous, which is to use that last matching definition in scope. Here, this will amount to choosing the type `two`. But we will ignore default resolution strategies for the moment.

[5]When running examples in OCaml, the `-principal` option should be used. By default OCaml does

This is because in the π-directional framework, all subexpressions in an application are considered as being inferred simultaneously *until* they are let-bound. More precisely, polymorphic types are considered to be *known* while monomorphic types are considered to be *unknown* (or rather not-yet-known).

Limitations of directionality Bidirectional typechecking is elegant and lightweight. It handles complex features well and supports the propagation of type information with minimal annotations. Its core weakness, however, lies in the need to fix an often arbitrary direction of type information – as in the application rule. Leading to a sound and complete algorithm with respect to a specification that made somewhat arbitrary choices to begin with.

π-directional inference appears better suited for ML, relying on polymorphism – the essence of ML. Yet it remains surprisingly weak: it does not even propagate user-written annotations from a function to its argument. For example, the following would be rejected as ambiguous under π-directional inference alone:

```
let e4 =
  let g : (one → int) → int = fun f → f { x = 0; y = 0 }
  in g (fun r → r.y)
```

To solve this, OCaml cheats and uses a weak form of bidirectional propagation, by passing the expected type of the argument so that `(fun r → r.y)` is typed with the expected type `(one → int)` and thus considered non-ambiguous. But this hybrid approach combines elements from both approaches without fully resolving the shortcomings of either.

Framework	$\mathtt{e}_0$	$\mathtt{e}_1$	$\mathtt{e}_2$	$\mathtt{e}_3$	$\mathtt{e}_4$	$\mathtt{e}_5$	$\mathtt{e}_6$	$\mathtt{e}_7$
Bidirectional	✓	✗	✓	✓	✓	✗	✗	✗
π-directional	✓	✗	✓	✗	✗	✗	✗	✗

Table 2.1: A table that summarises the typablity of each term under the approaches discussed so far. Some examples are introduced later *i.e.* $\mathtt{e}_5$ to $\mathtt{e}_7$.

Omnidirectional type inference The key idea of omnidirectional type inference is that *types can be inferred in any order*. Without polymorphism, this is straightforward: inference reduces to solving unification constraints, which are order-insensitive and easily deferred or rearranged.

The challenge arises with ML-style let-polymorphism, where all known implementations infer the type of a let-binding before typechecking its uses

not follow its principal type system for efficiency reasons.

$$
\begin{array}{rcll}
\alpha, \beta, \gamma & & & \text{Type variables} \\
\tau & ::= & \alpha \mid 1 \mid \tau_1 \to \tau_2 \mid \bar{\tau}\ \mathsf{t} \mid \ldots & \text{Types} \\
C & ::= & \mathbf{true} \mid \mathbf{false} \mid C_1 \wedge C_2 \mid \exists\alpha.\ C \mid \forall\alpha.\ C \mid \tau_1 = \tau_2 & \text{Constraints} \\
& \mid & \mathbf{let}\ x = \lambda\alpha.\ C_1\ \mathbf{in}\ C_2 \mid x\ \tau & \\
& \mid & \mathbf{match}\ \tau\ \mathbf{with}\ \bar{\chi} & \\
\chi & ::= & \rho \to C & \text{Branches} \\
\rho & ::= & 1 \mid \alpha \to \beta \mid \_\ t \mid \ldots & \text{Patterns}
\end{array}
$$

Figure 2.1: Syntax of constraints.

Yet, it is not mandatory. Pottier and Rémy's constraint-based account of ML shows that constraints can, in principle, be solved in any order by duplicating them – albeit at a significant cost to efficiency. Still, it shows that ordering is a convenience, not a necessity.

To make omnidirectional inference practical for ML, we introduce two new technical devices: **(1)** *partial type schemes*, which can be instantiated before being fully solved, and consequently refined incrementally; **(2)** *suspended match constraints*, which delay solving ambiguous constraints until enough contextual information is available. This comes at a cost: everything is hard.

(1) Implementing partial type schemes (while preserving efficiency) is non-trivial.
(2) Giving an adequate semantics for suspended constraints is also tricky, as we must declaratively characterise the notion of *contextually known* type information.

But the payoff is substantial: the semantics of suspended constraints naturally yield user-facing typing rules for fragile ML extensions, and our implementation remains sound and complete with respect to them.

The rest of this chapter is organised as follows. Section 2.1 introduces suspended constraints and demonstrates their application to various ML extensions. Section 2.1.3 defines their formal semantics. Section 2.2 describes a small extension of ML, dubbed OML[6], incorporating these features, along with its typing rules and constraint generator. Finally, Section 2.3 describes our constraint solver.

2.1 Suspended constraints

We begin by introducing the syntax of our constraint language, given in Figure 2.1. We write $\mathcal{T}$ for the set of types and $\mathcal{T}^*$ for the subset of non-variable types. The constraint language contains tautologic constraints (**true**), unsatisfiable constraints (**false**), and logical 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.

[6]Short for OmniML or omnidirectional ML.

There are two constructs that deal with the introduction and elimination of constraint abstractions. As the name suggests, 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 $\textbf{let}\ x = \lambda\alpha.\, C_1\ \textbf{in}\ C_2$ 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 bound to x.

Finally, we introduce *suspended match constraints*. These constraints defer constraint solving until the *shape*, *e.g.* the top-level constructor[7], of a type becomes known. Unknown unification variables lack a defined shape, thus solving is deferred until unification propagates it. At that point, the shape is matched against the branches to select the next constraint to solve. Formally, a suspended constraint is written as $\textbf{match}\ \tau\ \textbf{with}\ \bar{\chi}$, where:

(1) τ is the matchee type. The constraint remains suspended until τ's shape is determined.

(2) $\bar{\chi}$ is a list of branches $\rho \to C$, where each shape pattern ρ binds subcomponents of the type's shape. For example, the pattern $\_\ t$ matches with any nominal type and binds its name to t. The constraint C is then solved in the extended context.

If the shape of τ is never determined, the constraint remains unsatisfiable. The remainder of this section illustrates the role of suspended constraints in supporting *fragile* language features. These include: **(1)** semi-explicit first-class polymorphism, **(2)** constructor and record label overloading for nominal algebraic datatypes.

We demonstrate how the typability of each of these features can be elaborated into constraints, formalised 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. A formal account of the semantics of suspended constraints and the declarative typing rules for these features is deferred to Section 2.1.3 and Section 2.2, respectively.

2.1.1 Semi-explicit first-class polymorphism

Semi-explicit first-class polymorphism [10] uses *annotated types* to track the origins of polymorphic types. The type constructor $[\sigma]^\varepsilon$ boxes a polymorphic type 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 may themselves be generalised, yielding type schemes such as $\forall\varepsilon.\, [\sigma]^\varepsilon$.

The introduction form for polytypes is a boxing operator $[e : \exists\bar{\alpha}.\, \sigma]$ with an explicit polytype annotation $\exists\bar{\alpha}.\, \sigma$ where $\bar{\alpha}$ are type variables that are free in σ. The resulting

[7] We intentionally leave the notion of shape underspecified here; it is formalised in Section 2.2.

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 may be used in place of the free variables $\bar{\alpha}$. The annotation variable ε can thus be generalised. 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 $\langle e \rangle$ the type τ that is an instance of σ. If, by contrast, e has the type $[\sigma]^{\varepsilon}$ for some non-generalisable annotation variable ε, then $\langle e \rangle$ is ill-typed. It is precisely the polymorphism of ε that ensures that the polytype is indeed known and not being inferred.

In particular $\lambda x.\, \langle x \rangle$ is not typable, since the λ-bound variable x is assigned a monotype. Consequently, the only admissible type for x is $x : [\sigma]^{\varepsilon}$ for some σ and ε. However, since ε is bound in the surrounding context at the point of typing $\langle x \rangle$, it cannot be generalised 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 $\lambda x.\, \mathbf{let}\ \ x = (x : [\sigma])\ \ \mathbf{in}\ \ \langle x \rangle$, is well-typed because the explicit annotation introduces a fresh variable annotation ε_1, which can then be generalised, yielding $\forall\varepsilon_1.\, [\sigma]^{\varepsilon_1}$.

The very purpose of annotation variables is to distinguish *known*, polymorphic polytypes from yet-unknown, monomorphic ones. However, they are somewhat difficult to understand and sensitive to the placement of type annotation, an artifact of the fixed directionality of generalisation 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 generalisation and annotation variables interact. In the first term, the annotation occurs in an unboxing operator introducing fresh annotation variables and may therefore be generalised 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 generalised. As unboxing requires a generalised polytype, the second term is ill-typed despite the annotation.

Suspended match constraints eliminate this sensitivity to directionality, by allowing any type information to be treated as known – rendering annotation variables unnecessary.

Typechecking $[e : \exists\bar{\beta}.\,\sigma]$ is, intuitively:

$$[\![[e : \exists\bar{\beta}.\,\forall\bar{\gamma}.\,\tau] : \alpha]\!] \triangleq \exists\bar{\beta}.\,(\forall\bar{\gamma}.\,[\![e : \tau]\!]) \wedge \alpha = [\forall\bar{\beta}.\,\tau]$$

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 behaviour in the following constraint:

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

Here, the match constraint (**match** β **with** ...) suspends the instantiation until β is resolved to be a known polytype. If it is already known, the constraint discharges immediately binding the scheme to s and behaves like a standard instantiation constraint $s \leq \alpha$.

By waiting for e's type to be *known*, we may ensure principal types without annotation variables.

2.1.2 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.

OCaml supports a limited yet useful form of static overloading for record labels and datatype constructors:

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

type m = L | M
type n = M | N
```

When encountering overloaded labels or constructors, OCaml resolves ambiguity using local type information. As explained in the introduction, this process is underpinned by π-directional inference, where nominal types carry annotation variables ε, written $\bar{\tau}\ \mathsf{t}^{\varepsilon}$[8], to track which types are *known*. This mechanism allows one to deduce known types from generalised type constructors $\forall\varepsilon.\,\bar{\tau}\ \mathsf{t}^{\varepsilon}$.

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

[8]The annotation variable ε attaches to the nominal type t not the children $\bar{\tau}$.

typing constraint:

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

This constraint defers resolution until more information is available: specifically, until the name of the record type for the e is known.

We assume a global typing environment Ω, which provides type scheme declarations for constructors and record fields. These are written in the form:

$$\mathsf{K} : \forall\bar{\alpha}.\ \tau \to \bar{\alpha}\ \mathsf{t} \in \Omega$$
$$\ell : \forall\bar{\alpha}.\ \tau \to \bar{\alpha}\ \mathsf{t} \in \Omega$$

Since constructors and labels may be overloaded across distinct type constructors, Ω may contain multiple such schemes for a given constructor or label. However, within a given datatype, we assume that each constructor or label is uniquely defined. Thus, we write $\Omega(\ell/\mathsf{t})$ or $\Omega(\mathsf{K}/\mathsf{t})$ to denote the unique type scheme associated with a specific label or constructor in the context of the type t.

Once the record type of β is known to be $\bar{\tau}\ \mathsf{t}$, the suspended constraint (**match** β **with** ...) triggers the matching clause, binding t to t and retrieves the appropriate type scheme for ℓ from Ω, scoped to t. It then instantiates this scheme and imposes the resulting constraints on the argument and return types of the label.

As seen in the examples above, this approach is faithful to OCaml's current behaviour[9], and in fact improves on it: certain expressions, *e.g.* $\mathtt{e}_3$ to $\mathtt{e}_7$, are well-typed under our account but rejected by OCaml's current type checker.

```
let e5 r = (r.y, r.x)
let e6 = let gety r = r.y in gety { x = 1; y = 2 }
let e7 f =
  let g () = (f ()).y in ignore (f : unit → one); g ()
```

Suspended constraints offer a robust mechanism for dealing with extensions of ML that depend on a notion of *known* type information. What remains is to formalise this notion: that is, to give a precise semantics for when a type variable is considered *known*, and how suspended constraints are discharged accordingly.

2.1.3 Semantics

The semantics of constraints is given, as is standard, by a satisfiability judgement of the form $\phi \vdash C$: the constraint C is satisfied by the solution, or valuation, ϕ. The semantic environment ϕ maps type variables to *ground* types $\mathfrak{g} \in \mathcal{G}$, and term variables to sets of

[9]Modulo default disambiguation rules.

$$\textsc{Sat-True}\quad \phi \vdash \mathbf{true} \qquad \textsc{Sat-Conj}\quad \frac{\phi \vdash C_1 \qquad \phi \vdash C_2}{\phi \vdash C_1 \wedge C_2} \qquad \textsc{Sat-Exists}\quad \frac{\phi[\alpha := \mathfrak{g}] \vdash C}{\phi \vdash \exists \alpha.\ C} \qquad \textsc{Sat-Forall}\quad \frac{\forall \mathfrak{g}.\ \phi[\alpha := \mathfrak{g}] \vdash C}{\phi \vdash \forall \alpha.\ C}$$

$$\textsc{Sat-Unif}\quad \frac{\phi(\tau_1) = \phi(\tau_2)}{\phi \vdash \tau_1 = \tau_2} \qquad \textsc{Sat-Let}\quad \frac{\phi \vdash \exists \alpha.\ C_1 \qquad \phi[x := \phi(\lambda \alpha.\ C_1)] \vdash C_2}{\phi \vdash \mathbf{let}\ x = \lambda \alpha.\ C_1\ \mathbf{in}\ C_2} \qquad \textsc{Sat-App}\quad \frac{\phi(\tau) \in \phi(x)}{\phi \vdash x\ \tau}$$

Figure 2.2: Semantics of constraints (without suspended constraints)

ground types $\mathfrak{G} \subseteq \mathcal{G}$. We write $\phi[\alpha := \mathfrak{g}]$ for the environment agreeing with ϕ on all type variables except α, which is mapped to $\mathfrak{g}$. Similarly, $\phi[x := \mathfrak{G}]$ denotes an update for term variables. By homomorphism, ϕ maps types to ground types.

The judgement is defined, for all constraint-formers except suspended constraints, in Figure 2.2. **true** (Sat-True) is satisfied in any environment, and **false** in none. An environment ϕ satisfies $C_1 \wedge C_2$ (Sat-Conj) if it satisfies both C_1 and C_2. The existential rule $\phi \vdash \exists \alpha.\ C$ (Sat-Exists) requires that ϕ provides a "solution" for α: it maps it to a type τ that satisfies the expected constraints in C. The universal constraint $\forall \alpha.\ C$ (Sat-Forall) requires C to be satisfiable under any type mapped to α in ϕ. A unification constraint (Sat-Unif) holds if both sides are mapped to the exact same type under the environment.

The rule for let-bindings (Sat-Let) reflects that the constraint abstraction $\lambda \alpha.\ C_1$ must first be *satisfiable*, checking that C_1 holds under some instantiation of its bound variable; then the abstraction's interpretation to x in ϕ and subsequently checking the satisfiability of C_2. Formally, constraint abstractions $\lambda \alpha.\ C$ are interpreted semantically as a set of ground types $\mathfrak{g}$ that satisfy $\phi[\alpha := \mathfrak{g}] \vdash C$.

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

Application constraints $x\ \tau$ (Sat-App) are interpreted by checking that τ belongs to the set of types mapped to x by ϕ.

Closed constraints are either satisfiable in any semantic environment (*i.e.* they are tautologies) or unsatisfiable. For example, consider the constraint $\exists \alpha.\ \alpha = 1$, its satisfiability is established by the following derivation:

$$\dfrac{\dfrac{1 = 1}{\phi[\alpha := 1] \vdash \alpha = 1}\ \textsc{Sat-Unif}}{\phi \vdash \exists \alpha.\ \alpha = 1}\ \textsc{Sat-Exists}$$

We write $C_1 \vDash C_2$ to denote that C_1 entails C_2 *i.e.* every solution of C_1 also satisfies C_2, and $C_1 \equiv C_2$ when both constraints have the same solutions.

Shapes A shape is, informally, a type with holes – represented by type variables. Their grammar is:

$$\zeta \quad ::= \quad \bar{\gamma}.\,\tau \qquad \text{Shape}$$

where $\bar{\gamma}$ is a sequence of the free variables of τ. Hence, shapes are closed. They are considered equal modulo type equivalence and α-equivalence. We write $\bot$ for the trivial shape $\gamma.\,\gamma$. We write $\mathcal{S}$ the set of shapes and $\mathcal{S}^*$ for the subset of non-trivial shapes. Shapes are equipped with an ordering – that of the standard instantiation ordering, defined by:

$$\textsc{Shape-Inst}\quad \frac{\bar{\gamma}_2 \,\#\, \bar{\gamma}_1.\,\tau_1}{\bar{\gamma}_1.\,\tau_1 \leq \bar{\gamma}_2.\,\tau_1[\bar{\gamma}_1 := \bar{\tau}]}$$

When writing $\zeta \leq \zeta'$, we say ζ is more general than ζ'.

Given a shape $\bar{\gamma}.\,\tau$, we write $\bar{\tau}\ \langle\bar{\gamma}.\,\tau\rangle$ for applying the substitution $[\bar{\gamma} := \bar{\tau}]$ to τ, that is $\bar{\tau}\ \langle\bar{\gamma}.\,\tau\rangle \triangleq \tau[\bar{\gamma} := \bar{\tau}]$. A type τ has a decomposition $\bar{\tau}\ \langle\zeta\rangle$ iff $\tau = \bar{\tau}\ \langle\zeta\rangle$; we say that ζ is *a* shape of τ. A decomposition is trivial if the shape is $\bot$.

Definition 1. *A non-trivial shape ζ is the principal shape of the type τ if:*

(1) $\tau = \bar{\tau}\ \langle\zeta\rangle$,
(2) $\forall \zeta' \in \mathcal{S}^*.\, \tau = \bar{\tau}'\ \langle\zeta'\rangle \implies \zeta \leq \zeta'$.

Theorem 1 (Principality of shapes)**.** *Every non-variable type τ admits a principal shape.*

A principal shape $\bar{\gamma}.\,\tau$ is canonical if the sequence of its free variables $\bar{\gamma}$ appear in the order in which the variables occur in τ. We write ς for canonical principal shapes.

We write $\mathsf{shape}(\tau) = \varsigma$ for the canonical principal shape of τ (if defined). Canonical principal shapes can be seen as a generalisation of type constructors, capable of coping with polytypes. For example, the polytype $[\forall\alpha.\ (\alpha \to \alpha) \to \beta \to 1]$ has the shape $\gamma.\,[\forall\alpha.\ (\alpha \to \alpha) \to \gamma]$. Generally speaking, principal shapes are dictated by:

- If τ is not a polytype, *e.g.* $\bar{\tau}\ \mathsf{t}$ then the principal shape is $\bar{\gamma}.\,\bar{\gamma}\ \mathsf{t}$.
- If τ is a polytype $[\forall\bar{\beta}.\ \tau']$, then τ' is *skeletal*, that is all subterms that do not lead to a polymorphic variable in $\bar{\beta}$ must be a variable in $\bar{\gamma}$.

The canonical principal shape of a type governs its decomposition. We write $\mathsf{decomp}(\tau)$ for the decomposition of τ induced by its canonical principal shape, if it exists.

Suspended constraints To support suspended constraints, we first introduce a rule that applies when the shape of the matchee is statically determined from the constraint:

$$\textsc{Sat-Susp-Use}\quad \frac{\tau \in \mathcal{T}^* \qquad \rho_i \textbf{ matches } \mathsf{decomp}(\tau) = \theta \qquad \phi \circ \theta \vdash C_i}{\phi \vdash \textbf{match } \tau \textbf{ with } \bar{\rho} \to \bar{C}}$$

The rule states that a suspended constraint is satisfied by ϕ whenever τ is a non-variable type with a defined shape, and there is a *unique* pattern ρ_i among the branches that matches this shape, yielding a substitution θ, such that the corresponding constraint C_i is satisfied under the extended environment $\phi \circ \theta$.

We require that the match against the shape is unambiguous (*i.e.* there is a unique matching pattern) but not necessarily exhaustive[10]. The definition of the matching function $\rho \textbf{ matches } \bar{\tau}\ \langle \varsigma \rangle = \theta$ is given in Figure 2.3.

$$\textsc{Match-Unit}\quad 1 \textbf{ matches } \cdot\ \langle \cdot.\, 1 \rangle = \cdot \qquad \textsc{Match-Arr}\quad \alpha_1 \to \alpha_2 \textbf{ matches } (\tau_1, \tau_2)\ \langle \gamma_1, \gamma_2.\, \gamma_1 \to \gamma_2 \rangle = [\bar{\alpha}_i := \bar{\tau}_i]$$

$$\textsc{Match-Nom}\quad \_\, t \textbf{ matches } \bar{\tau}\ \langle \bar{\gamma}.\, \bar{\gamma}\ \mathsf{t} \rangle = [t := \mathsf{t}] \qquad \textsc{Match-Poly}\quad [s] \textbf{ matches } \bar{\tau}\ \langle \bar{\gamma}.\, [\sigma] \rangle = [s := \sigma[\bar{\gamma} := \bar{\tau}]]$$

Figure 2.3: The definition of the **matches** partial function, including polytype patterns.

The key innovation of our semantics lies in the following rule, which – unlike the others – is not syntax-directed:

$$\textsc{Sat-Susp-Ctx}\quad \frac{\mathcal{C}[\alpha\,!\,\bar{\gamma}.\, \tau'] \qquad \phi \vdash \mathcal{C}[\exists \bar{\gamma}.\, \alpha = \tau' \wedge \textbf{match } \tau' \textbf{ with } \bar{\chi}]}{\phi \vdash \mathcal{C}[\textbf{match } \alpha \textbf{ with } \bar{\chi}]}$$

This rule lets us substitute the matchee variable α by a more defined shape, provided the constraint context $\mathcal{C}$ determines α to have precisely this shape. The side-condition $\mathcal{C}[\alpha\,!\,\varsigma]$, which we refer to as the *unicity predicate*, ensures this precisely: it expresses that the context $\mathcal{C}$ constraints all solutions of α have the *same* principal shape ς. The formal definition, which is somewhat technical, is given below.

Definition 2. *A type variable α has a* uniquely known principal *shape within the context $\mathcal{C}[-]$, written $\mathcal{C}[\alpha\,!\,\varsigma]$, iff for all assignments ϕ and ground types $\mathfrak{g}$, then $\phi \vdash \mathcal{C}[\alpha = \mathfrak{g}]$ implies that $\mathsf{shape}(\mathfrak{g}) = \varsigma$. In other words:*

$$\mathcal{C}[\alpha\,!\,\varsigma] \quad \triangleq \quad \forall \phi, \mathfrak{g}.\ \phi \vdash \mathcal{C}[\alpha = \mathfrak{g}] \implies \mathsf{shape}(\mathfrak{g}) = \varsigma$$

[10] If no branch matches the shape, the constraint is unsatisfiable by default.

This condition ensures that we're not guessing *a shape arbitrarily; the shape must already be determined by the ambient context.*

The astute reader may have noticed a complication resulting from our definition of the unicity predicate: $\mathcal{C}[\alpha\,!\,\varsigma]$ contains a negative occurrence of the satisfiability judgement, in order to express that ς is unique among all solutions of the ambient context $\mathcal{C}$. This negative occurrence means that *a priori*, the rules from Figure 2.2 including Sat-Susp-Use, Sat-Susp-Ctx do not form an inductive definition in the usual sense. However, it can easily be resolved by observing that the rules, while not syntactically well-founded, can be defined in a decreasing way by noting that the conclusion of Sat-Susp-Ctx $\mathcal{C}[\mathbf{match}\ \alpha\ \mathbf{with}\ \bar{\chi}]$ is structurally larger than the negatively-occurring judgement $\mathcal{C}[\alpha = \tau]$.

Theorem 2. *The relation $\phi \vdash C$ is well-founded.*

2.2 The OML calculus

Syntax Figure 2.4 defines the grammar of OML. Terms include all of ML: variables x, the unit literal (), lambda-abstractions $\lambda x.\ e$, applications $e_1\ e_2$, annotations $(e : \exists\bar{\alpha}.\,\tau)$, and let-bindings $\mathbf{let}\ \ x = e_1\ \ \mathbf{in}\ \ e_2$. We extend this with: **(1)** tuples $(e_1, \ldots, e_n)$ with overloaded tuple projections $e.i$, modelling a simplification of record overloading; **(2)** semi-explicit first-class polymorphism, with boxing $[e : \exists\bar{\alpha}.\,\sigma]$ and unboxing $\langle e\rangle$ constructs.

Each construct that endangers principality (written e^i), specifically projections and unboxing, have an explicitly annotated counterpart (written e^x): $e.i/n$ and $\langle e : \exists\bar{\alpha}.\,\sigma\rangle$, respectively.

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

Figure 2.4: Syntax of types and terms of the OML calculus.

Types are split into monotypes (or just types) and type schemes. Types τ include variables α, the unit type 1, function types $\tau_1 \to \tau_2$, n-ary products $\Pi_{i=1}^{n}\bar{\tau_i}$, and polytypes $[\sigma]$. Schemes σ are universally quantified types of the form $\forall\bar{\alpha}.\,\tau$.

Typing The typing judgement $\Gamma \vdash e : \sigma$ assigns a type scheme σ to a term e under the context Γ. Contexts consist of term variable bindings. The rules are given in Figure 2.5. Rules Var through Let are standard ML typing rules.

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

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

$$\textsc{Oml-Let}\ \frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \textbf{let}\ \ x = e_1\ \ \textbf{in}\ \ e_2 : \tau} \qquad \textsc{Oml-Tuple}\ \frac{(\Gamma \vdash e_i : \tau_i)_{i=1}^n}{\Gamma \vdash (e_1, \ldots, e_n) : \Pi_{i=1}^n \bar{\tau}_i}$$

$$\textsc{Oml-Proj-X}\ \frac{\Gamma \vdash e : \Pi_{i=1}^n \bar{\tau}_i \qquad 1 \le j \le n}{\Gamma \vdash e.j/n : \tau_j} \qquad \textsc{Oml-Proj-I}\ \frac{\mathcal{E}[e\,!\,\bar{\gamma}.\, \Pi_{i=1}^n \bar{\gamma}_i] \qquad \Gamma \vdash \mathcal{E}[e.j/n] : \tau}{\Gamma \vdash \mathcal{E}[e.j] : \tau}$$

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

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

Figure 2.5: Typing rules for OML.

The typing rules for fully annotated terms (e^x) are straightforward (Proj-X and Use-X). However, typing rules for terms with omitted type annotations (e^i) are non-compositional: they depend on a surrounding one-hole context $\mathcal{E}$. Hence, they assert that the typability of the expression $\mathcal{E}[e^i]$ where e^i is an expression with an implicit type annotation. We first request a typing for the expression with an explicit annotation $\Gamma \vdash \mathcal{E}[e^x] : \tau$ where e^x is a fully annotated variant of e^i[11]. We then require that the (shape of the) annotation is fully determined from the context, written $\mathcal{E}[e\,!\,\varsigma]$[12].

To describe the unicity predicate $\mathcal{E}[e\,!\,\varsigma]$, we introduce an internal cast construct $\{e : \tau\}$ which allows a term e with type τ to be typed in any context. The typing rule is:

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

Casts are not allowed in source programs: they only exist to define unicity. The unicity

[11]This is analogous to requiring that the suspended match constraint is satisfiable with a more defined shape.

[12]This is analogous to the unicity predicate in constraint satisfiability.

predicate is defined as:

$$\mathcal{E}[e\,!\,\varsigma] \quad \triangleq \quad \forall \Gamma, \tau, \mathfrak{g}.\ \Gamma \vdash \mathcal{E}[\{e : \mathfrak{g}\}] : \tau \implies \mathsf{shape}(\mathfrak{g}) = \varsigma$$

The implicit rule Proj-I types the projection $e.i$ when $\mathcal{E}$ forces the shape of e to be a tuple with arity n. Similarly, Use-I permits instantiating a polytype in $\langle e \rangle$ if the context $\mathcal{E}$ determines that e must be a polytype with the skeleton $\bar{\gamma}.\,[\sigma]$.

2.2.1 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 e is well typed. The translation function $[\![e : \alpha]\!]$ takes a term e and an expected type α.

$$\begin{array}{lcl}
[\![x : \alpha]\!] & \triangleq & x\ \alpha \\
[\![() : \alpha]\!] & \triangleq & \alpha = 1 \\
[\![\lambda x.\ e : \alpha]\!] & \triangleq & \exists \beta, \gamma.\ \alpha = \beta \to \gamma \wedge \textbf{let}\ x = \lambda \beta'.\ \beta' = \beta\ \textbf{in}\ [\![e : \gamma]\!] \\
[\![e_1\ e_2 : \alpha]\!] & \triangleq & \exists \beta, \gamma.\ \beta = \gamma \to \alpha \wedge [\![e_1 : \beta]\!] \wedge [\![e_2 : \gamma]\!] \\
[\![\textbf{let}\ \ x = e_1\ \ \textbf{in}\ \ e_2 : \alpha]\!] & \triangleq & \textbf{let}\ x = \lambda \beta.\ [\![e_1 : \beta]\!]\ \textbf{in}\ [\![e_2 : \alpha]\!] \\
[\![(e : \exists \bar{\alpha}.\,\tau) : \alpha]\!] & \triangleq & \exists \bar{\alpha}.\ (\!|e : \tau|\!) \wedge \alpha = \tau \\
[\![(e_1, \ldots, e_n) : \alpha]\!] & \triangleq & \exists \bar{\alpha}.\ \alpha = \Pi_{i=1}^{n} \bar{\alpha} \wedge \bigwedge_{i=1}^{n} [\![e_i : \alpha_i]\!] \\
[\![e.j/n : \alpha]\!] & \triangleq & \exists \beta, \bar{\beta}.\ \beta = \Pi_{i=1}^{n} \bar{\beta} \wedge [\![e : \beta]\!] \wedge \alpha = \beta_j \\
[\![e.j : \alpha]\!] & \triangleq & \exists \beta.\ [\![e : \alpha]\!] \wedge \textbf{match}\ \beta\ \textbf{with}\ (\Pi\ \beta_j \to \alpha = \beta) \\
[\![[e : \exists \bar{\beta}.\,\sigma] : \alpha]\!] & \triangleq & \exists \bar{\beta}.\ (\!|e : \sigma|\!) \wedge \alpha = [\sigma] \\
[\![\langle e \rangle : \alpha]\!] & \triangleq & \exists \beta.\ [\![e : \beta]\!] \wedge \textbf{match}\ \beta\ \textbf{with}\ ([s] \to s \leq \alpha) \\
(\!|e : \tau|\!) & \triangleq & \exists \alpha.\ \alpha = \tau \wedge [\![e : \alpha]\!] \\
(\!|e : \forall \alpha.\ \sigma|\!) & \triangleq & \forall \alpha.\ (\!|e : \sigma|\!) \\
[\![\emptyset \vdash e : \sigma]\!] & \triangleq & (\!|e : \sigma|\!) \\
[\![x : \sigma, \Gamma \vdash e : \sigma']\!] & \triangleq & \textbf{let}\ x = \lambda \alpha.\ \sigma \leq \alpha\ \textbf{in}\ [\![\Gamma \vdash e : \sigma']\!]
\end{array}$$

Figure 2.6: The constraint generation translation for terms

Constraint generation is defined in Figure 2.6, assuming all variables introduced are fresh.

Variables generate an instantiation constraint. () requires the expected type α to be 1. Functions generate fresh variables for the argument and return types and use a monomorphic[13] **let** constraint to bind the parameter x. Application binds two fresh variables for the function and argument types and ensures α is the return type of the

[13]The **let** constraint is monomorphic since α_3 is fully constrained by type variables defined outside the abstraction's scope and therefore cannot be generalised.

function. Let-bindings generate a polymorphic **let** constraint; $\lambda\alpha.\ [\![e : \alpha]\!]$ is a principal constraint abstraction for e: its intended interpretation is the set of all types that e admits.

Annotations bind their flexible variables and assert that the annotated type τ' matches the expected type τ. Tuples introduce fresh variables for the components and unify the expected type with the product of the components. Explicit projections unify α with the appropriate tuple component. For implicit projections, we suspended until the shape of the tuple is known, selecting the j-th component for the expected type.

For polytype boxing, we ensure that e has the specified polymorphic type and enforce that the overall type is the polytype $[\forall\bar{\alpha}.\ \tau']$. When unboxing, a suspended constraint delays instantiation until the inferred type of e is resolved to a polytype $[s]$, at which point α must be an instance of s.

This translation is sound and complete with respect to the typing rules:

Theorem 3. *(Constraint generation is sound and complete)* $\Gamma \vdash e : \sigma$ *if and only if* $\vDash [\![\Gamma \vdash e : \sigma]\!]$.

2.3 Constraint Solving

We now present a solver for the constraint language defined in Section 2.1. The solver operates as a rewriting system on constraints. Once no further transitions are applicable, the constraint is either in solved form – from which we can read off a most general solution – or the solver becomes stuck, indicating unsatisfiability.

Unification Our constraints ultimately reduce to equations between types, which we solve using first-order unification. Following Pottier and Rémy, we replace binary equality constraints with *multi-equations* – equations involving an arbitrary number of types rather than binary pairs. Multi-equations are considered modulo permutation of their members.

$$\begin{array}{rcll} U & ::= & \mathbf{true} \mid \mathbf{false} \mid U_1 \wedge U_2 \mid \exists\alpha.\ U \mid \epsilon & \text{Unification problems} \\ \epsilon & ::= & \tau \mid \epsilon = \tau & \text{Multi-equations} \end{array}$$

Figure 2.7: Syntax of unification problems

Like the solver, the unification algorithm is specified as a non-deterministic transition relation between unification problems $U_1 \longrightarrow U_2$, given in Figure 2.8. Rewriting is performed modulo α-equivalence, commutativity of conjunction, and under an arbitrary unification problem context $\mathcal{U}$.

We refer the reader to [29] for a detailed account of the Pottier and Rémy's original unification rules. Our specification closely follows their presentation but differs in a crucial respect: we use principal shapes in place of type constructors. This enables a uniform

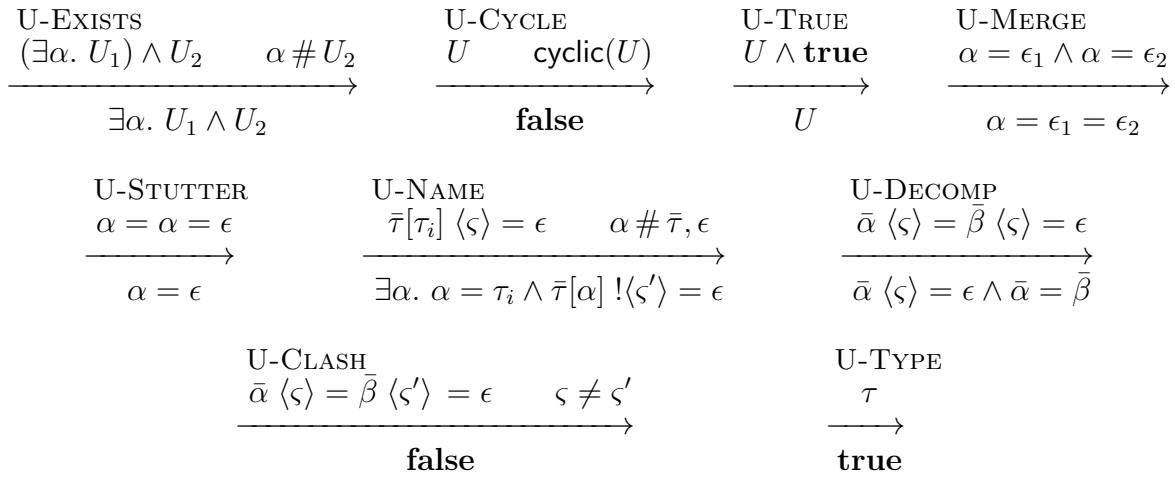


Figure 2.8: Unification algorithm as a series of rewriting rules $U_1 \longrightarrow U_2$. All shapes are canonical principal shapes.

treatment of monotypes and polytypes within unification, avoiding the need for specialised handling of polytypes, as found in Garrigue and Rémy's work [10].

Solved unification problems $\hat{U}$ take the form of a conjunction $\exists\bar{\alpha}.\ \bigwedge_{i=1}^{n} \epsilon_i$ of multi-equations, each with at most one non-variable type, no shared variables, and no cycles.

Solving rules We now introduce the rules of the constraint solver itself (Figure 2.9). These rules are non-deterministic. Rewriting is performed modulo α-equivalence, commutativity and associativity of conjunction and under an arbitrary *constraint context* $\mathcal{C}$.

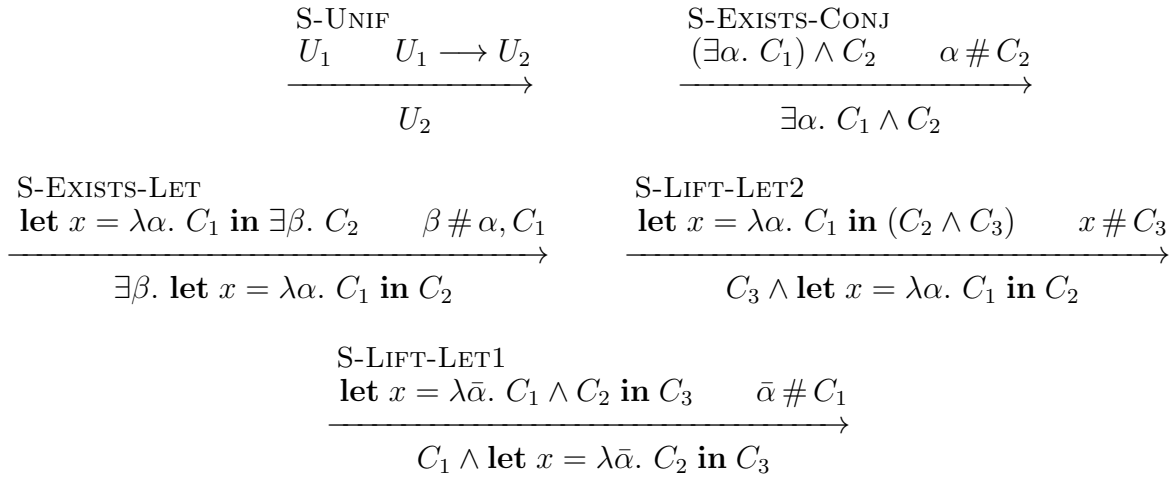


Figure 2.9: Select rewriting rules of the solver

Basic rules S-Unif invokes the unification algorithm to the current unification problem. The unification algorithm itself is treated as a block box by the solver, allowing the solver to be parameterised over the equational theory of types implemented by the unifier.

S-Exists-* lifts existential quantifiers to the nearest enclosing **let**, if any, or otherwise to the top of the constraint, which we refer to as a *region* within the constraint. Similarly, S-Lift-Let* hoists constraints out of let-binders when they do not depend on the locally bound variables. These lifting rules normalise the structure of each region into a block of existentially bound variables with a unification problem, together with unresolved constraints.

Suspended constraints S-Susp-Use solves suspended match constraints whose scrutinee has a locally known shape – either statically, or as a result of applying S-Susp-Ctx. The rule selects a unique matching branch based on the shape of the scrutinee and applies the generated substitution to the branch. If no branch matches, the rule is stuck, indicating that the constraint is unsatisfiable.

$$\text{S-Susp-Ctx}\quad \frac{\mathcal{C}[\mathbf{match}\ \alpha\ \mathbf{with}\ \bar{\chi}] \qquad \alpha = \tau = \epsilon \in \mathcal{C} \qquad \mathsf{shape}(\tau) = \bar{\gamma}.\,\tau'}{\mathcal{C}[\exists \bar{\gamma}.\ \alpha = \tau' \wedge \mathbf{match}\ \tau'\ \mathbf{with}\ \bar{\chi}]}$$

$$\text{S-Susp-Use}\quad \frac{\mathbf{match}\ \tau\ \mathbf{with}\ \bar{\rho} \to \bar{C} \qquad \tau \in \mathcal{T}^* \qquad \rho_i\ \mathbf{matches}\ \mathsf{decomp}(\tau) = \theta}{\theta(C_i)}$$

Figure 2.10: Solving rules for suspended match constraints.

When the solver encounters a suspended constraint with an unknown shape, it cannot proceed immediately. Instead, it must wait until it can construct a surrounding context $\mathcal{C}$ that includes a multi-equation unifying the scrutinee with a type whose shape is defined. This enables the suspended match to be rewritten, according to Sat-Susp-Ctx, thereby exposing the shape required to continue solving in S-Susp-Use.

Let constraints Application constraints can be solved by copying constraints (*i.e.* β-reduction)[14]:

$$\text{S-Let-App-Copy}\quad \frac{\mathbf{let}\ x = \lambda\alpha.\ C_1\ \mathbf{in}\ \mathcal{C}[x\ \tau] \qquad \alpha \# \tau}{\mathbf{let}\ x = \lambda\alpha.\ C_1\ \mathbf{in}\ \mathcal{C}[\exists\alpha.\ \alpha = \tau \wedge C_1]}$$

However, this duplicates constraint solving work. A more efficient strategy first solves the abstraction fully and then reuses it – solving constraint abstractions is equivalent to rank-based generalisation [29].

[14]Not entirely correct, as we will see with backpropagation.

But, the inclusion of suspended constraints complicates generalisation. To illustrate this, let us examine:

```
type three = { x : int; y : int } (* y is overloaded *)
...
let e9 r = let y = r.y in r.x + y
```

The generated constraint[15] is:

$$\exists\alpha.\ \textbf{let}\ y = \lambda\beta.\ \textbf{match}\ \alpha\ \textbf{with}\ (\_\ t \to \Omega(y/t) \leq \beta \to \alpha)\ \textbf{in}\ y\ \mathsf{int} \wedge \alpha = \mathsf{three}$$

Here, α is `r`'s type. The constraint remains suspended until `r.x` forces `r`'s type to be `three` When generalising `y`, the type β – captured by the match – has not yet been resolved. We must, however, allocate a scheme for `y`, resulting in $\forall\beta.\ \beta$ – which appears unsound since β later unifies with **int**.

This motivates *partial type schemes*: type schemes that delay commitment to certain quantifications (*e.g.* β). These *partially generalised* variables are generalised, but can still be refined in future as suspended constraints are discharged.

To encode partial schemes, we introduce *partial application constraints* $x[\rho \vdash \bar{\epsilon}]$, where ρ maps variables bound inside x's abstraction to fresh copies and $\bar{\epsilon}$ accumulates copied multi-equations. Copying proceeds by two rules:

(1) S-Papp-Exists allocates fresh existential variables at the call site that are mapped to uninstantiated variables within the abstraction.
(2) S-Papp-Unif copies multi-equations from the abstraction to the call site.

S-Papp-Solve removes partial applications once all necessary constraints have been copied – that is, when the copied equations entail the abstraction body. S-Let-Solve removes a **let** constraint when the bound term variable is unused and the abstraction is satisfiable.

S-Lower implements Rémy's efficient rank-based generalisation: variables $\bar{\beta}$ dominated by monomorphic variables are removed from the abstraction. Partial applications that copied $\bar{\beta}$ are notified, unifying their copies with the lowered $\bar{\beta}$.

Backpropagation Our semantics supports *backpropagation*, allowing type information from use sites to refine matched variables inside generalised abstractions. For example, in:

```
let e10 = let f r = r.y in f ({ x = 1; y = 1 })
```

[15] Simplified for readability.

S-Let-App

$$\frac{\textbf{let } x = \lambda\alpha.\ C_1 \textbf{ in } \mathcal{C}[x\ \tau] \qquad x \# \mathcal{C} \qquad \beta \# \tau, \alpha}{\textbf{let } x = \lambda\alpha.\ C_1 \textbf{ in } \mathcal{C}[\exists\beta.\ \beta = \tau \wedge x[[\alpha := \beta] \vdash \emptyset]]}$$

S-Papp-Exists

$$\frac{\textbf{let } x = \lambda\bar{\alpha}.\ \exists\beta.\ C_1 \textbf{ in } \mathcal{C}[x[\rho \vdash \bar{\epsilon}]] \qquad \beta \notin \mathsf{dom}\,\rho \qquad \beta \# \rho, \bar{\epsilon} \qquad x \# \mathcal{C}}{\textbf{let } x = \lambda\bar{\alpha}, \beta.\ C_1 \textbf{ in } \mathcal{C}[\exists\beta'.\ x[\rho[\beta := \beta'] \vdash \bar{\epsilon}]]}$$

S-Papp-Unif

$$\frac{\textbf{let } x = \lambda\bar{\alpha}.\ C_1 \wedge \epsilon \textbf{ in } \mathcal{C}[x[\rho \vdash \bar{\epsilon}]] \qquad \mathsf{dom}\,\rho = \bar{\alpha} \qquad \bar{\epsilon} \not\models \epsilon \qquad x \# \mathcal{C}}{\textbf{let } x = \lambda\bar{\alpha}.\ C_1 \wedge \epsilon \textbf{ in } \mathcal{C}[x[\rho \vdash \bar{\epsilon}, \epsilon] \wedge \rho(\epsilon)]}$$

S-Papp-Solve

$$\frac{\textbf{let } x = \lambda\bar{\alpha}.\ \bar{\epsilon}' \textbf{ in } \mathcal{C}[x[\rho \vdash \bar{\epsilon}]] \qquad \mathsf{dom}\,\rho = \bar{\alpha} \qquad \bar{\epsilon} \vDash \bar{\epsilon}' \qquad x \# \mathcal{C}}{\textbf{let } x = \lambda\bar{\alpha}.\ \bar{\epsilon}' \textbf{ in } \mathcal{C}[\textbf{true}]}$$

S-Let-Solve

$$\frac{\textbf{let } x = \lambda\bar{\alpha}.\ \bar{\epsilon} \textbf{ in } C \qquad x \# C \qquad \exists\bar{\alpha}.\ \bar{\epsilon} \equiv \textbf{true}}{C}$$

S-Lower

$$\frac{\textbf{let } x = \lambda\bar{\alpha}, \bar{\beta}.\ C \textbf{ in } \mathcal{C}[\overline{x[\rho \vdash \bar{\epsilon}]}] \qquad \exists\bar{\alpha}.\ C \text{ determines } \bar{\beta} \qquad \bar{\beta} \subseteq \mathsf{dom}\,\bar{\rho} \qquad x, \bar{\beta} \# \mathcal{C}}{\exists\bar{\beta}.\ \textbf{let } x = \lambda\bar{\alpha}.\ C_1 \textbf{ in } \mathcal{C}[\overline{x[\rho \setminus \bar{\beta} \vdash \bar{\epsilon}] \wedge \rho(\bar{\beta}) = \bar{\beta}}]}$$

Figure 2.11: Solving rules for let-bindings and applications.

OCaml rejects this, as π-directional inference cannot propagate the known type `three` back to `f`'s parameter. The (simplified) constraint generated when typing $\mathtt{e}_{10}$ is:

$$\begin{aligned}&\exists\alpha.\ \textbf{let } f = \lambda\delta.\ \exists\beta, \gamma.\ \delta = \beta \to \gamma \wedge \textbf{match } \beta \textbf{ with } (\_\ t \to \Omega(y/t) \leq \gamma \to \beta)\\ &\quad \textbf{in } f\ (\mathsf{three} \to \alpha)\end{aligned}$$

At the use site, the abstraction is applied with a type of known shape (`three`). S-BackProp propagates this shape back into the suspended match inside the abstraction by unifying it with the variable β. This is sound: the unicity predicate $\mathcal{C}[\alpha\,!\,\varsigma]$ ensures that β's shape is `three` provided the context $\mathcal{C}$ include the application constraint.

S-BackProp

$$\frac{\begin{array}{c}\mathcal{C}[\textbf{let } x = \lambda\bar{\alpha}.\ \mathcal{C}_1[\textbf{match } \alpha \textbf{ with } \bar{\chi}] \textbf{ in } \mathcal{C}_2[x[\rho \vdash \bar{\epsilon}]]]\\ \alpha \in \mathsf{dom}\,\rho \qquad \rho(\alpha) = \tau = \epsilon \in \mathcal{C}[\mathcal{C}_2] \qquad \mathsf{shape}(\tau) = \bar{\gamma}.\ \tau'\end{array}}{\mathcal{C}[\textbf{let } x = \lambda\bar{\alpha}.\ \mathcal{C}_1[\exists\bar{\gamma}.\ \alpha = \tau' \wedge \textbf{match } \tau' \textbf{ with } \bar{\chi}] \textbf{ in } \mathcal{C}_2[x[\rho \vdash \bar{\epsilon}]]]}$$

Theorem 4 (Progress). *If $\cdot \vdash C$ and C is not solved, then there exists a C' such that $C \longrightarrow C'$.*

Theorem 5 (Termination). *The constraint solver terminates on all inputs.*

Theorem 6 (Preservation). *If $C_1 \longrightarrow C_2$, then $C_1 \equiv C_2$.*

3 Generalised algebraic data types

3.1 Introduction

Generalised algebraic data types (GADTs), introduced by Xi [52], are a simple generalisation of algebraic data types that allow one to describe richer constraints between constructors and their types.

The canonical example use of GADTs is expressing a type-safe evaluator for a simply-typed embedded domain specific language (DSL). The data type `expr`, defined in Figure 3.1, for the abstract syntax trees is given a type parameter α that describes the type of the expression.

```
type α expr =
  | Int : int → int expr
  | Add : (int → int → int) expr
  | App : (α → β) expr * α expr → β expr

let rec eval : type α. α expr → α =
  function
  | Int n → n
  | Add → fun x y → x + y
  | App (f, x) → eval f (eval x)
```

Figure 3.1: The abstract syntax tree and type safe evaluator for a simply-typed language in OCaml using GADTs.

An `Int n` has the type `int` in the DSL, thus its type is `int expr`. `Add` is the constructor for the addition operator, typed with `int → int → int`. `App` is a constructor for the application operator, which takes a function expression `(α → β) expr` and an argument expression `(α expr)`, returning a `β expr`.

The definition allows us to write a type-safe evaluator that does not perform any tagging or untagging of interpreter values. A key mechanism for this is the use of type-level equalities, introduced in `match` constructs. For instance, in the first case of `eval`, the type of `e` is `α expr` which is known to match `Int n` of type `int expr`. As a result, the equation α = `int` must hold within the branch. The added equation is later used to prove that the

result of the branch, that is, the integer `n`, has the type α, as required by `eval`'s signature.

3.1.1 Type inference

While GADTs are a powerful tool for type-safe programming, their inference poses challenges. In general, full type inference of GADTs is undecidable [38]. Practical systems regain decidability by relying on some annotations, yet these systems are often not principal. Sulzmann et al. [43] showed that programs with type-level equalities frequently have more than one principal type.

For example, in the `match` branch for `Int n`, `n` can be typed as `int` or α, without neither being more general than the other outside the branch. This breaks principality – key for efficient inference[1]– and raises the question: which type should we infer when there are multiple choices? In short, it's difficult.

3.1.2 Previous work

Type inference of GADTs is notoriously complex area and remains an active research area. For space reasons, we focus on works framing inference within the Hindley-Milner framework.

Simonet and Pottier [42] show that inference for HMG(X), an extension of HM(X) with GADTs, can be reduced to constraint satisfiability in a fragment of first-order logic involving implications.

$$C \quad ::= \quad \textbf{true} \mid \textbf{false} \mid \tau_1 = \tau_2 \mid \exists \alpha.\ C \mid \forall \alpha.\ C \mid C_1 \Rightarrow C_2 \qquad \text{Constraints}$$

Unfortunately, this fragment is intractable primarily due to the inclusion of universal quantification and implications. To recover tractability, Simonet [41] restricts implications to *rigid implications* of the form $\forall \bar{\alpha}.\ C_1 \Rightarrow C_2$ with $\mathsf{fv}(C_1) \subseteq \bar{\alpha}$. Still, their constraint generator demands programmer-supplied annotations on every GADT-matching function, specifying the scrutinee and return types. Furthermore, they sidestep principality issues by permitting implication constraints in type schemes, which further complicates things for the programmer.

Peyton Jones et al. [16] present a tractable inference algorithm for GADTs that also relies on programmer-supplied annotations. They introduced the wobbly types to distinguish annotated types (rigid) from inferred types (wobbly). Crucially, only matching on rigid types introduces type equalities. While effective, wobbly types proved unpredictable in behaviour and unwieldy in implementation, motivating cleaner alternatives: stratified type inference and OutsideIn.

[1]Since principality permits us to make locally optimal decisions without backtracking.

Stratified inference [28] improves on wobbly types by cleanly separating annotation propagation from inference in two phases. The first generates *shapes* – partial type annotations for match scrutinees and return types; while the second performs type inference. Though the overall system is incomplete due to the heuristic nature of the first phase, the second phase is sound, complete, and enjoys principal types. Nonetheless, introducing concepts like shapes increases cognitive load when programming, and full propagation of inferred types remains elusive.

OutsideIn [38] also separates propagation and inference, via a constraint-based approach using *delayed implications* $[\bar{\alpha}](\forall \bar{\beta}.\ C_1 \Rightarrow C_2)$. Constraint solving proceeds in two steps: first, solving simple constraints[2], propagating both annotated and inferred types; then solving the delayed implications. This deferral ensures that inference for GADT match branches occurs when more is known about the scrutinee and the expected return type.

Principality is maintained by restricting implication constraints from unifying any global unification variables in the closure $[\bar{\alpha}]$ unless already resolved. Thus, information only flows from the *outside* into the implication's conclusion. While the inference algorithm produces principal types, the underlying declarative type system does not.

Curiously, OutsideIn's delayed implications echo our own idea of suspended constraints (Chapter 2). Yet Vytiniotis et al. [47] argue – rather controversially – that delayed implication constraints make local let-generalisation all but unmanageable, both in theory and implementation. Their proposed fix is to abandon local let-generalisation altogether. We beg to differ, having already cleared that particular thicket in Section 2.3.

Garrigue and Rémy [11] introduce *ambivalent types* to reconcile principality with GADTs. An ambivalent type ζ is a set of types that are equal under the local constraints but not globally. Ambivalence arises precisely when the term does not have a principal type. An ambivalent type is said to have 'escaped' its scope if the set of types are no longer equal under the current context, thereby breaking principality. For instance:

```
type (_, _) eq = Refl : (α, α) eq

let g (type α) (w : (α, int) eq) (y : α) =
  match w with Refl → if y > 0 then y else 0
```

```
Error: This expression has type int
       but an expression was expected of type α
       This instance of α is ambiguous:
       it would escape the scope of its equation
```

Here, the `then` branch returns `y`, with type α, and the `else` branch returns an `int`. The resulting ambivalent type is $\alpha \approx$ `int`, which represents a type that is either α or `int`. When exiting the scope of the `match` branch, the ambivalent type escapes its scope as the

[2]Everything but non-trivial delayed implications.

equality no-longer holds. Annotations are used to fix this, eliminating ambivalence.

Ambivalent types depend on *sharing* to guarantee principal types. When instantiating a type scheme $\forall\alpha.\ \zeta$ *without* sharing, we lose the information that *all* copies of α must be synchronised. Garrigue and Rémy address this by using *annotation variables* (Section 2.1.1), written ψ^α, where all occurences of α share the structure ψ. The catch? Sharing must happen everywhere, as any type can be ambivalent. This complicates the system significantly, demanding a severe departure from the conventional ML presentation and making the formalisation notably harder to grasp.

3.2 The AML calculus

Ambivalent types remain the most expressive solution to date. Stratified type inference and wobbly types fall short by disallowing the propagation of inferred types altogether. OutsideIn does better but halts at GADT case branches. Ambivalent types go further: they allow information to flow even from within GADT branches, so long as doing so doesn't leak an ambivalent type.

Crucially, ambivalent types also come with a declarative type system that enjoys principal types. Yet their formalisation is remarkably complex – the authors themselves later discovered subtle flaws in their specification [30].

We introduce AML, a new calculus for typing GADT that builds on ambivalent types while drawing inspiration from contextual modal type theory to simplify the specification. Contextual modal type theory (CMTT) derives from intuitionistic modal logic S4 via the Curry-Howard correspondence. The contextual variant grades the necessity modality $\Box A$ with a context of propositions Ψ. The proposition $[\Psi]A$ holds in the current world if A can be proved in every accessible future world using *only* the propositions from Ψ as hypotheses. Computationally, $[\Psi]A$ describes programs of type A with free variables drawn solely from Ψ.

CMTT, extended with a first-class notion of *names*, offers a principled framework for tracking dependencies and detecting scope escapes – situations where a term refers to variables that are no longer bound, as in staged metaprogramming [22]. Our key insight is that the mechanism used in ambivalent types to prevent ill-scoped types can be elegantly recast using this approach: names track the scope of equalities, while contexts track their uses.

AML adopts this perspective by interpreting ambivalent types via the contextual box modality. A type $[\Psi]\tau$ denotes the equivalence class of τ under a set of *named* equalities Ψ. In the original ambivalent system, the problem of scoping arises, when a type, such as $\tau_1 \approx \tau_2$, becomes visible outside the branch defining the equality that entails $\tau_1 = \tau_2$.

In the example below, the `match` branch, introduces a fresh name ϕ for the equality

$\alpha = \mathbf{int}$. The result type of the branch is $[\phi]\alpha$ (or equivalently $[\phi]\mathbf{int}$), as the body relies on the assumption $\alpha = \mathbf{int}$. Since the type escapes the branch (as there is no annotation), ϕ escapes its scope – yielding an ill-typed program, as expected.

```
let g (type α) (w : (α, int) eq) (y : α) =
  match eq with Refl →
    (* fresh name ϕ for α = int *)
    (if y > 0 then y else 0
     (* has type [ϕ]α due to use of ϕ *))
```

Syntax Given in Figure 3.2, AML extends ML terms with explicit universal quantifiers $\Lambda\alpha.\ e$. The literal constructor **Refl** has type $\tau_1 = \tau_2$, introducing type-level equalities. The **match** $e_1 : \tau_1 = \tau_2$ **with** **Refl** $\to e_2$ eliminates the proof of the equality $\tau_1 = \tau_2$, introducing it as a *local constraint* in e_2 using the proof e_1.

$$
\begin{array}{rcll}
e & ::= & x \mid () \mid \lambda x.\ e \mid e_1\ e_2 \mid \mathbf{let}\ \ x = e_1\ \ \mathbf{in}\ \ e_2 & \text{Terms} \\
 & \mid & \Lambda\alpha.\ e \mid (e : \tau) & \\
 & \mid & \mathbf{Refl} \mid \mathbf{match}\ \ e_1 : \tau_1 = \tau_2\ \ \mathbf{with}\ \ \mathbf{Refl} \to e_2 & \\
\tau^\kappa & ::= & \alpha \mid 1 \mid \tau_1^\kappa \to \tau_2^\kappa \mid \tau_1^\kappa = \tau_2^\kappa \mid \emptyset \mid \tau^\kappa, \phi \mid [\tau_1^\kappa]\tau_2^\kappa & \text{Types} \\
\mathsf{F} & ::= & 1 \mid \cdot \to \cdot \mid \cdot = \cdot & \text{Type constructors} \\
\kappa & ::= & \mathsf{ty} \mid \mathsf{sc} & \text{Kinds} \\
 & & & \\
\Gamma & ::= & \cdot \mid \Gamma, x : \sigma \mid \Gamma, \alpha ::^\varphi \kappa \mid \Gamma, \phi : \tau_1 = \tau_2 & \text{Contexts} \\
\sigma & ::= & \tau^\kappa \mid \forall\alpha :: \kappa.\ \sigma & \text{Type schemes} \\
\varphi & ::= & \mathrm{f} \mid \mathrm{r} & \text{Flexibilities}
\end{array}
$$

Figure 3.2: The syntax of terms and types in AML.

Types τ^κ are *kinded*, where κ is the kind. We have the usual kind for types (ty), but also a kind for *scopes* (sc). Often we elide the kind, writing τ for types and Ψ for scopes. For clarity, we use α, β, γ for type variables, ϕ for equation names, and ς, ϱ for scope variables.

Types in AML extend standard ML types with equality types ($\tau_1 = \tau_2$) and *scoped ambivalent types*, written $[\Psi]\tau$. Informally, $[\Psi]\tau$ denotes the equivalence class of τ under the equations named in Ψ. Under the equivalence relation defined later, $[\Psi]\tau$ is equivalent to $[\Psi]\tau'$ whenever τ and τ' are provably equal under Ψ. Such a type is well-formed in a context Γ only if all names in Ψ are bound in Γ; otherwise, the equation is said to have escaped its scope.

A scope Ψ is a row of named equations: either empty $\emptyset$, a scope variable ς, or an extension Ψ, ϕ. Polymorphic type schemes generalise not only over type variables but scope variables, enabling *scope polymorphism*.

Contexts are extended with polymorphic scope variables and named equations. Each type variable is annotated with a *flexibility* φ which can either be rigid (r) if introduced

by an explicit Λ-quantifier, or flexible (f), if introduced by implicitly by generalisation. Only rigid variables may appear in type annotations.

Well-formedness Well-formedness is extended to ensure the coherence of ambivalent types, thus preventing scope escapes. The judgement $\Gamma \vdash \tau^{\kappa} :: \kappa\ \varphi$ tracks both kinds and flexibility. For convenience, we write $\Gamma \vdash \tau^{\kappa} :: \kappa$ when τ^{κ} is flexible.

Following Garrigue and Rémy, we impose two key restrictions on types:

(1) Only equations between *rigid* types can be introduced to the context.
(2) In $[\Psi]\tau$, the type τ must be rigid – since only rigid types may occur in equalities. Thus, a flexible type cannot be ambivalent.

$$\textsc{Aml-F}{\leq}\textsc{-Refl}\quad \varphi \leq \varphi \qquad \textsc{Aml-F}{\leq}\textsc{-Wk}\quad \mathrm{r} \leq \mathrm{f} \qquad \textsc{Aml-Ty-Var}\quad \frac{\alpha ::^{\varphi_1} \kappa \in \Gamma \qquad \varphi_1 \leq \varphi_2}{\Gamma \vdash \alpha :: \kappa\ \varphi_2}$$

$$\textsc{Aml-Ty-Con}\quad \frac{\forall i,\ \Gamma \vdash \tau_i :: \mathsf{ty}\ \varphi \qquad \mathsf{arity}(\mathsf{F}) = |\bar{\tau}|}{\Gamma \vdash \bar{\tau}\ \mathsf{F} :: \mathsf{ty}\ \varphi} \qquad \textsc{Aml-Ty-Box}\quad \frac{\Gamma \vdash \Psi :: \mathsf{sc} \qquad \Gamma \vdash \tau :: \mathsf{ty}\ \mathrm{r}}{\Gamma \vdash [\Psi]\tau :: \mathsf{ty}} \qquad \textsc{Aml-Ty-EmpSc}\quad \Gamma \vdash \emptyset :: \mathsf{sc}$$

$$\textsc{Aml-Ty-SnocSc}\quad \frac{\Gamma \vdash \Psi :: \mathsf{sc} \qquad \phi \in \mathsf{dom}(\Gamma)}{\Gamma \vdash \Psi, \phi :: \mathsf{sc}} \qquad \textsc{Aml-Scm-Ty}\quad \frac{\Gamma \vdash \tau :: \mathsf{ty}}{\Gamma \vdash \tau\ \mathsf{scm}} \qquad \textsc{Aml-Scm-}\forall\quad \frac{\Gamma, \alpha ::^{\mathrm{f}} \kappa \vdash \sigma\ \mathsf{scm} \qquad \alpha \# \Gamma}{\Gamma \vdash \forall \alpha :: \kappa.\ \sigma\ \mathsf{scm}}$$

$$\textsc{Aml-Ctx-Emp}\quad \cdot\ \mathsf{ctx} \qquad \textsc{Aml-Ctx-Var}\quad \frac{\Gamma\ \mathsf{ctx} \qquad \Gamma \vdash \sigma\ \mathsf{scm} \qquad x \# \Gamma}{\Gamma, x : \sigma\ \mathsf{ctx}} \qquad \textsc{Aml-Ctx-TyVar}\quad \frac{\Gamma\ \mathsf{ctx} \qquad \alpha \# \Gamma}{\Gamma, \alpha ::^{\varphi} \kappa\ \mathsf{ctx}}$$

$$\textsc{Aml-Ctx-Eq}\quad \frac{\Gamma\ \mathsf{ctx} \qquad \Gamma \vdash \tau_1, \tau_2 :: \mathsf{ty}\ \mathrm{r} \qquad \phi \# \Gamma}{\Gamma, \phi : \tau_1 = \tau_2\ \mathsf{ctx}}$$

Figure 3.3: Well-formedness rules for AML.

Type equivalence We define an equivalence relation on types, written $\Gamma \vdash \tau_1 \equiv \tau_2\ [\Psi]$, meaning that τ_1 and τ_2 are equivalent in the context Γ and under the scope Ψ. We elide the scope and write $\Gamma \vdash \tau_1 \equiv \tau_2$ when Ψ is empty.

Equivalence relies on the standard rules: reflexivity, symmetry, transitivity, congruence, injectivity, and the use of equalities from the context Γ provided they're referenced in Ψ.

The final three rules are specific to scoped ambivalent types. **(1)** Aml-≡Box allows scoped types to be compared under the same scope. **(2)** Aml-≡Dist distributes the scopes within type constructors. This rule is crucial for ensuring eliminators can be applied

$$\textsc{Aml-}\equiv\textsc{Refl}\ \frac{\Gamma \vdash \tau :: \mathsf{ty}}{\Gamma \vdash \tau \equiv \tau\ [\Psi]} \qquad \textsc{Aml-}\equiv\textsc{Sym}\ \frac{\Gamma \vdash \tau_1 \equiv \tau_2\ [\Psi]}{\Gamma \vdash \tau_2 \equiv \tau_1\ [\Psi]} \qquad \textsc{Aml-}\equiv\textsc{Trans}\ \frac{\Gamma \vdash \tau_1 \equiv \tau_2\ [\Psi] \qquad \Gamma \vdash \tau_2 \equiv \tau_3\ [\Psi]}{\Gamma \vdash \tau_1 \equiv \tau_3\ [\Psi]}$$

$$\textsc{Aml-}\equiv\textsc{Use}\ \frac{\phi : \tau_1 = \tau_2 \in \Gamma \qquad \phi \in \Psi}{\Gamma \vdash \tau_1 \equiv \tau_2\ [\Psi]} \qquad \textsc{Aml-}\equiv\textsc{Conj}\ \frac{\forall i,\ \Gamma \vdash \tau_i \equiv \tau_i'\ [\Psi] \qquad \mathsf{arity}(\mathsf{F}) = |\bar{\tau}| = |\bar{\tau}'|}{\Gamma \vdash \bar{\tau}\ \mathsf{F} \equiv \bar{\tau}'\ \mathsf{F}\ [\Psi]}$$

$$\textsc{Aml-}\equiv\textsc{Inj}\ \frac{\Gamma \vdash \bar{\tau}\ \mathsf{F} \equiv \bar{\tau}'\ \mathsf{F}\ [\Psi]}{\Gamma \vdash \tau_i \equiv \tau_i'\ [\Psi]} \qquad \textsc{Aml-}\equiv\textsc{Box}\ \frac{\Gamma \vdash \tau_1 \equiv \tau_2\ [\Psi_1]}{\Gamma \vdash [\Psi_1]\tau_1 \equiv [\Psi_1]\tau_2\ [\Psi_2]} \qquad \textsc{Aml-}\equiv\textsc{Dist}\ \frac{\Gamma \vdash [\Psi_1](\bar{\tau}\ \mathsf{F}) :: \mathsf{ty} \qquad \mathsf{arity}(\mathsf{F}) > 0}{\Gamma \vdash [\Psi_1](\bar{\tau}\ \mathsf{F}) \equiv \overline{[\Psi_1]\tau}\ \mathsf{F}\ [\Psi_2]}$$

$$\textsc{Aml-}\equiv\textsc{M}\ \frac{\Gamma \vdash \tau :: \mathsf{ty\ r}}{\Gamma \vdash [\emptyset]\tau \equiv \tau\ [\Psi]}$$

Figure 3.4: Type equivalence rules for AML.

correctly. The arity side-condition ensures that the scope cannot be forgotten, as in the case of 1 (which has no subterms). **(3)** Aml-≡M eliminates trivial scopes.

These rules are inspired by corresponding modal logic axioms and preserve equivalence in the presence of local equalities.

Typing The typing rules are given in Figure 3.5. The ML rules, Aml-Var through to Aml-Let are standard.

The instantiation relation is extended to support type equivalence and scope polymorphism. Interestingly, a consequence of our instantiation relation is that the following rule is derivable in AML:

$$\textsc{Aml-Equiv}\ \frac{\Gamma \vdash e : \tau_1 \qquad \Gamma \vdash \tau_1 \equiv \tau_2}{\Gamma \vdash e : \tau_2}$$

This rule plays a crucial role in manipulating scopes in our typing judgements with the distributive rule.

Type equalities Reflexivity introduces a type equalitiy $\tau = \tau$ via **Refl**. Matching on a witness e_1 with **match** $e_1 : \tau_1 = \tau_2$ **with** **Refl** $\rightarrow e_2$ adds the equality $\tau_1 = \tau_2$ implicitly to the context Γ while checking e_2. The witness must match $\tau_1 = \tau_2$ but may be ambivalent – eliminated using annotations (discussed below). Since the equality is local to e_2, it must not appear in the return type τ. We ensure this by allocating a fresh name for the equality $\phi \# \Gamma$ and ensure ϕ does not occur in τ using well-formedness under Γ. If violated, the equality escapes its scope.

$$\textsc{Aml-Var}\ \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \textsc{Aml-Unit}\ \Gamma \vdash () : \forall \varsigma :: \mathsf{sc}.\ [\varsigma]1 \qquad \textsc{Aml-Fun}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \qquad \Gamma \vdash \tau_1 :: \mathsf{ty}}{\Gamma \vdash \lambda x.\ e : \tau_1 \to \tau_2}$$

$$\textsc{Aml-App}\ \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \qquad \textsc{Aml-Gen}\ \frac{\Gamma, \alpha ::^{\mathrm{f}} \kappa \vdash e : \sigma \qquad \alpha \# \Gamma}{\Gamma \vdash e : \forall \alpha :: \kappa.\ \sigma}$$

$$\textsc{Aml-Inst}\ \frac{\Gamma \vdash e : \sigma_1 \qquad \Gamma \vdash \sigma_1 \leq \sigma_2}{\Gamma \vdash e : \sigma_2} \qquad \textsc{Aml-Let}\ \frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \mathbf{let}\ \ x = e_1\ \ \mathbf{in}\ \ e_2 : \tau}$$

$$\textsc{Aml-Forall}\ \frac{\Gamma, \alpha ::^{\mathrm{r}} \mathsf{ty} \vdash e : \sigma \qquad \alpha \# \Gamma \qquad \alpha \notin \mathsf{dangerous}(\sigma)}{\Gamma \vdash \Lambda\alpha.\ e : \forall \alpha :: \mathsf{ty}.\ \sigma}$$

$$\textsc{Aml-Annot}\ \frac{\Gamma \vdash e : \tau_1 \qquad \Gamma \vdash \tau :: \mathsf{ty\ r} \qquad \Gamma \vdash \lceil \tau \rceil \leq \tau_1, \tau_2}{\Gamma \vdash (e : \tau) : \tau_2} \qquad \textsc{Aml-Refl}\ \Gamma \vdash \mathbf{Refl} : \forall \alpha :: \mathsf{ty}.\ \alpha = \alpha$$

$$\textsc{Aml-Match}\ \frac{\Gamma \vdash (e_1 : \tau_1 = \tau_2)_{\_} \qquad \phi \# \Gamma \qquad \Gamma, \phi : \tau_1 = \tau_2 \vdash e_2 : \tau \qquad \Gamma \vdash \tau :: \mathsf{ty}}{\Gamma \vdash \mathbf{match}\ \ e_1 : \tau_1 = \tau_2\ \ \mathbf{with}\ \ \mathbf{Refl} \to e_2 : \tau}$$

$$\textsc{Aml-}\leq\textsc{-Equiv}\ \frac{\Gamma \vdash \tau_1 \equiv \tau_2}{\Gamma \vdash \tau_1 \leq \tau_2} \qquad \textsc{Aml-}\leq\textsc{-ForallL}\ \frac{\Gamma \vdash \tau :: \kappa \qquad \Gamma \vdash \sigma_1[\alpha := \tau] \leq \sigma_2}{\Gamma \vdash \forall \alpha :: \kappa.\ \sigma_1 \leq \sigma_2} \qquad \textsc{Aml-}\leq\textsc{-ForallR}\ \frac{\alpha \# \Gamma \qquad \Gamma, \alpha ::^{\mathrm{f}} \kappa \vdash \sigma_1 \leq \sigma_2}{\Gamma \vdash \sigma_1 \leq \forall \alpha :: \kappa.\ \sigma_2}$$

Figure 3.5: The typing rules for AML.

Annotations Annotations eliminate scopes and ambivalence, thereby preventing scope escapes. A (rigid) type τ admits a family of *consistent instances* by inserting fresh scopes. Annotations therefore simply check if the term can be typed with two consistent instances of the annotated type.

$$\begin{array}{rcll} \lceil \alpha \rceil & \triangleq & \varsigma \rhd [\varsigma]\alpha & \text{fresh } \varsigma \\ \lceil \mathsf{F} \rceil & \triangleq & \varsigma \rhd [\varsigma]\mathsf{F} & \text{fresh } \varsigma \\ \lceil \bar{\tau}\ \mathsf{F} \rceil & \triangleq & \mathcal{S}_1, \ldots, \mathcal{S}_n \rhd \bar{\tau}'\ \mathsf{F} & \text{where } \lceil \tau_i \rceil = \mathcal{S}_i \rhd \tau_i \end{array}$$

Here $\mathcal{S}$ is a set of scope variables. We write $\forall \lceil \tau \rceil$ for $\forall \mathcal{S}.\ \tau'$ where $\lceil \tau \rceil = \mathcal{S} \rhd \tau'$. This scheme describes a set of instances that are equivalent to τ modulo scopes. This trick is also re-applied (and inlined) for constant rules with no subterms, such as Aml-Unit

Rigid variables Generalisation over rigid variables arises only in Aml-Forall, where the explicit quantifier $\Lambda\alpha.\ e$ requires that the rigid variable α is not in a *dangerous* position.

A dangerous position is one that is under a (non-trivial) scope. Formally:

$$\begin{aligned}
\mathsf{dangerous}(\alpha) &= \emptyset \\
\mathsf{dangerous}(\bar{\tau}\ \mathsf{F}) &= \bigcup_{i=1}^{n} \mathsf{dangerous}(\tau_i) \\
\mathsf{dangerous}([\Psi]\tau) &= \begin{cases} \mathsf{fv}(\tau) & \text{if } \Psi \neq \emptyset \\ \mathsf{dangerous}\tau & \text{otherwise} \end{cases} \\
\mathsf{dangerous}(\forall \alpha :: \kappa.\ \sigma) &= \mathsf{dangerous}(\sigma) \setminus \{\alpha\}
\end{aligned}$$

The restriction is necessary because AML allows flexible instantiations of polymorphic variables. If a generalisable rigid variable were allowed in a scoped position, instantiation could yield ill-formed types: scoped ambivalent types $[\Psi]\tau$ require τ to be rigid.

3.2.1 Constraint generation

Our constraint language extends that of Section 2.2.1 to support scoped ambivalent types, reinterpreting equality constraints as equivalences in the new type algebra.

$$C \quad ::= \quad \ldots \mid \mathbf{let}\ x = \forall \bar{\alpha}.\lambda\beta.\ C_1\ \mathbf{in}\ C_2 \mid \phi : \tau_1 = \tau_2 \Rightarrow C \qquad \text{Constraints}$$

$$\begin{array}{lcl}
[\![() : \alpha]\!] & \triangleq & \exists \varsigma.\ \alpha = [\varsigma]1 \\
[\![(e : \tau) : \alpha]\!] & \triangleq & (\!|e : \tau|\!) \wedge \lceil \tau \rceil \leq \alpha \\
[\![\mathbf{Refl} : \alpha]\!] & \triangleq & \exists \beta.\ \alpha = (\beta = \beta) \\
[\![\mathbf{match}\ \ e_1 : \tau_1 = \tau_2\ \ \mathbf{with}\ \ \mathbf{Refl} \to e_2 : \alpha]\!] & \triangleq & (\!|e_1 : \tau_1 = \tau_2|\!) \wedge \phi : \tau_1 = \tau_2 \Rightarrow [\![e_2 : \alpha]\!] \\
[\![\Lambda\beta.\ e : \alpha]\!] & \triangleq & \mathbf{let}\ x = \forall \beta.\lambda\gamma.\ [\![e : \gamma]\!]\ \mathbf{in}\ x\ \alpha \\
(\!|e : \tau|\!) & \triangleq & \exists \alpha.\ \lceil \tau \rceil \leq \alpha \wedge [\![e : \alpha]\!]
\end{array}$$

Figure 3.6: Syntax and constraint generation function for AML.

Quantifiers and annotations Although Λ-quantifiers in terms $\Lambda\alpha.\ e$ could be typed using constraint-level universal quantification $\forall\alpha.\ C$, we instead introduce *rigid constraint abstractions* $\forall\bar{\alpha}.\lambda\beta.\ C$, to ensure linear-time constraint generation. Like ordinary constraint abstractions, they are applied via $x\ \tau$, but are interpreted as $\exists\bar{\alpha}.\ C[\beta := \tau]$. Unlike standard abstractions, their satisfability (*e.g.*, in Sat-Let) requires checking the constraint $\forall\bar{\alpha}.\ \exists\beta.\ C$ under the current environment.

We illustrate their necessity by noting that an explicitly quantified term $\Lambda\alpha.\ e$ may not necessarily be polymorphic, it may implicitly be instantiated to a monotype. Consider:

```
let id_int (n : int) = (fun (type α) (x : α) : α → x) n
```

Here, the polymorphic function is instantiated to `int` $\to$ `int`. To encode this without rigid constraint abstractions, we would duplicate constraints:

$$[\![\Lambda\beta.\, e : \alpha]\!] \triangleq (\forall\beta.\, \exists\gamma.\, [\![e : \gamma]\!]) \wedge \exists\beta.\, [\![e : \alpha]\!]$$

Type equalities and implications To represent type equalities introduced by pattern matching, we add implication constraints $\phi : \tau_1 = \tau_2 \Rightarrow C$ binding the equality $\tau_1 = \tau_2$ to ϕ. Existential quantifiers $\exists\alpha.\, C$ must ensure proper scoping of equation names when guessing solutions in Sat-Exists. Match expressions generate these implications by extending the scope with $\tau_1 = \tau_2$ under a fresh name ϕ when typing the body e_2.

4 Thesis proposal

Thus far, we have introduced *omnidirectional type inference*, a principal constraint-based inference technique that supports fragile ML extensions. We also outlined how this framework extends to GADTs using a new presentation of ambivalent types.

Our thesis aims to demonstrate that constraint-based inference can scale to key OCaml features: GADTs, polymorphic variants, objects and classes, static overloading, polytypes, and polymorphic parameters.

This final section outlines the remaining work and open problems, each marked with a MoSCoW[1] priority to indicate its importance to the overall thesis. For each feature, we aim to establish the meta-theoretic properties:

(1) Soundness and completeness of constraint generation: the generated constraint reflects e's typability – satisfied if and only if e is well-typed.

(2) Correctness and termination of solving: satisfiable constraints yield solved forms; unsatisfiable ones lead to failure.

(3) Principality: every well-typed term admits a most general type, which is the result of inference.

Most of these results will likely be established by induction, as in the ML setting. We do not plan to mechanise our results – due to time constraints and the lack of existing formalisations[2]. Our focus in on the design and implementation of the constraint-based framework itself.

This work is part of a broader research programme in collaboration with Gabriel Scherer, Didier Rémy, and Samuel Vivien at INRIA.

4.1 Structural polymorphism

Structural polymorphism refers to a form of parametric polymorphism where some types have known structure. For example, the function `print_name` (Figure 4.0) operates on *any record* with a `name` field.

[1] Must have, Should have, Could have, Won't have.

[2] For instance, the principality of ambivalent types is yet to be formalised, despite the efforts on COCTI [30].

OCaml's polymorphic variants and objects both rely on structural polymorphism. Inference for these features builds on Garrigue's framework [8], though various other presentations exist [34, 49, 32, 25, 6].

Polymorphic variants are similar to traditional variant types but need no explicit type declaration. They are useful for extensible errors and tracking structural properties at the type-level. They are written with a backtick:

```
let nan = `Not_a_number
```

This expression has type `[> `Not_a_number]`, meaning any variant type that *includes at least* the constructor `` `Not_a_number ``. Variant types can also have upper bounds:

```
let is_a_number t =
  match t with
  | `Not_a_number → false
  | `Int _ → true
```

The inferred type is `[< `Int of int | `Not_a_number ] → bool`, the variant type is interpreted as a variant containing constructors `` `Int `` or `` `Not_a_number ``, or some subset. These upper and lower bounds form a *subtyping* relation on polymorphic variants. A fully determined variant, such as `number`, fixes both bounds to be equal:

```
type number = [ `Int of int | `Float of float | `Not_a_number ]
```

Objects in OCaml are a structurally typed collection of methods and values, resembling records. For example:

```
let alistair = object
  method name = "Alistair O'Brien"
  method mobile = "07976-700055"
end

let print_name p = Printf.printf "Name: s" p#name
```

Figure 4.1: Example use of objects and field projection.

Unlike Java or C++, OCaml's object types are structural as opposed to nominal. Object types consist of a list of *methods* (or fields) between two chevrons `< .. >`, coming in two forms:

Open The function `print_name` has type `< name: string; .. > → unit`, requiring a `name` field of type `string`, but allowing additional fields.

Closed The object `alistair` has the inferred type `< name: string; mobile: string >`, meaning exactly those two fields must be present, with no others.

Objects support depth and width subtyping via an explicit coercion operator `:>`. Classes act as object templates, supporting inheritance and overriding.

Research direction (Must) Garrigue's framework utilises kinds to encode structural polymorphism. A kind $\kappa = (L, U, T)$ is a triple that specifies which tags may or may not be present (*presence information*) and associates argument types with each tag, where:

- L is the set of required tags $\{tag_1, \ldots, tag_n\}$;
- U is the set of allowed tags, either a finite set or $\top$ (all tags);
- T is a finite mapping (row) from tags to types $tag_1 :: \tau_1 :: \ldots :: tag_m : \tau_m$;

where $L \subseteq U$, $L \subseteq \mathsf{dom}\, T$, and if $U \neq \top$, then $U \subseteq \mathsf{dom}\, T$. Additionally, each tag in L has a unique type in T.

We reinterpret kinds as constraints and aim to design a constraint language capable of typing both polymorphic variants and objects, The language includes atomic subtyping constraints $L \leq \tau$ and $\tau \leq U$ for rows and $\tau \leq \tau'$ for explicit coercions. This offers a more principled view, and aligns more closely with OCaml's current behaviour and integrates naturally with existing constraint-based frameworks.

Prior constraint-based work for polymorphic variants exists: Castagna et al. [1] formalised polymorphic variants using set-theoretic types (*i.e.* negation, union and intersection types) with semantic subtyping. While more expressive than Garrigue's framework, their approach has exponential complexity $2^{\mathcal{O}(n)}$ due to the use of semantic subtyping, making it impractical when combined with let-generalisation.

4.2 Omndirectional type inference

4.2.1 Default rules

Omnidirectional type inference enables order-independent constraint solving for ML guaranteeing principal types for fragile extensions. However, its commitment to principality and its avoidance of 'guessing' introduces limitations – particularly in settings where unique shapes cannot be inferred from the context.

A concrete example of this limitation arises in OCaml's treatment of polymorphic object methods. Consider the class:

```
class [α] list_collection l = object
  val contents = l
  method mem x = List.mem x contents
  method fold : (β → α → β) → β → β
    = fun f x → List.fold_left f x contents
end
```

Internally, OCaml represents such methods using polytypes. For instance, the polymorphic method `fold` is actually:

```
method fold = [fun f x →
    List.fold_left f x contents : ∀β. (β → α → β) → β → β ]
```

For uniformity, even monomorphic methods (like `mem`) are encoded as:

```
method mem = [fun x → List.mem x contents : ∃α. α]
```

This uniform treatment relieves users from marking methods as polymorphic when calling them. But, it shifts the challenge to the type system: inferring a polytype shape without adequate contextual information is more likely. For instance:

```
let send_mem l x = l#mem x
```

fails to typecheck under principled inference, since the polytype shape of `l` cannot be inferred from the context.

Research direction (Must) To support such cases, we propose a *default rule*: when propagation fails to determine a polytype's shape, we assume its monomorphic. For instance, the constraint:

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

uses the **else** clause as a fallback when the shape of β cannot be determined from the context. Default rules are also able to capture OCaml's default disambiguation of constructors and record labels.

While they appear to be a neat fix, default rules introduce subtle semantic complexities. For example:

$$\begin{aligned}&\textbf{match}\ \alpha\ \textbf{with}\ (\textbf{int} \to \beta = \textbf{int})\ \textbf{else bool}\\ \wedge\ &\textbf{match}\ \beta\ \textbf{with}\ (\textbf{int} \to \alpha = \textbf{int} \mid \textbf{bool} \to \textbf{true})\ \textbf{else int}\end{aligned}$$

The satisfiability of this constraint depends on which default clause fires first – yet the solver cannot determine this order *a priori*.

Our prototype explores a semantics where mutually dependent default branches fire in lockstep. This avoids unsoundness but requires computing the strongly connected components of the match dependency graph during solving – an operation that is non-trivial.

Despite their utility in modeling OCaml's behaviour, default rules pose challenges for both performance and formal semantics.

4.2.2 Polymorphic parameters

Polymorphic parameters, proposed by White [51], extends ML with functions of the form $\lambda x : \sigma.\ e$ where σ is a (typically rank-1) polymorphic type scheme. Inferring the rank-2 type $\sigma \to \tau$ for the function, which can be applied to arguments that are at least a polymorphic as σ. This provides succinct higher-rank idioms without full-blown System F. For example:

```
let apply_twice = fun (f : ∀α. α → α) → (f 42, f "1337")
```

Here, `f` is explicitly annotated as the polymorphic identity function, enabling its application at multiple instantiations.

Research direction (Must) We can encode polymorphic parameters using our existing polytype machinery, using the translation function $\mathcal{T}[\![-]\!]$. Specifically, we reinterpret the domain of function types as polytypes and translate applications to pass an *implicit polytype*.

$$\begin{array}{rcl@{\qquad}rcl} \mathcal{T}[\![\sigma \to \tau]\!] & \triangleq & [\mathcal{T}[\![\sigma]\!]] \to \mathcal{T}[\![\tau]\!] & \mathcal{T}[\![\lambda x : \sigma.\ e]\!] & \triangleq & \lambda x.\ \textbf{let}\ \ x = \langle (x : [\mathcal{T}[\![\sigma]\!]]) \rangle\ \ \textbf{in}\ \ \mathcal{T}[\![e]\!] \\ \mathcal{T}[\![1]\!] & \triangleq & 1 & \mathcal{T}[\![e_1\ e_2]\!] & \triangleq & \mathcal{T}[\![e_1]\!]\ [\mathcal{T}[\![e_2]\!]] \\ \mathcal{T}[\![\alpha]\!] & \triangleq & \alpha & \mathcal{T}[\![\lambda x.\ e]\!] & \triangleq & \lambda x.\ \textbf{let}\ \ x = \langle x \rangle\ \ \textbf{in}\ \ \mathcal{T}[\![e]\!] \end{array}$$

This encoding relies on an implicit polytype boxing operator $[e]$, which introduces a polytype without requiring an annotation, provided it can uniquely be inferred from the context. Its generated typing constraint is shown in Figure 4.2.

$$[\![[e] : \alpha]\!] \triangleq \textbf{let}\ x = \lambda\beta.\ [\![e : \beta]\!]\ \textbf{in match}\ \alpha\ \textbf{with}\ ([s] \to x\ s)$$

Figure 4.2: Constraint generation for implicit polytypes. The constraint $x\ s$ means that constraint abstraction bound to x can be instantiated to the type scheme s.

A consequence of the translation is that the implicit boxing and unboxing may limit the propagation of types. For example, seemingly simple programs are no-longer well-typed:

```
let apply f x = f x
```

We have two potential solutions in mind: **(1)** Introduce a default monomorphisation rules to polymorphic parameters. In the example, we could infer the type $\forall\alpha, \beta.\ (\alpha \to \beta) \to \alpha \to \beta$ for `apply`. But since every application and abstraction introduces a suspended constraint, the solver may have to aggressively apply the default rule to make progress, which in our current prototype could impact performance of inference. **(2)** Alternatively, we could permit quantification over polytypes themselves, *i.e.*, the type of `apply` could be $\forall\xi :: \mathsf{poly}, \beta.\ (\xi \to \beta) \to \xi \to \beta$ where ξ is a polytype *variable*.

4.2.3 Modular implicits

Modular implicits is a proposed extension to OCaml's module system, designed to support *ad-hoc polymorphism* through type-directed implicit parameters. Inspired by Scala's implicits and Harper et al's [3] modular type classes, they allow functions to receive first-class modules implicitly, resolved by the type information at the call site.

Figure 4.3 shows a simple pretty-printing library using implicits. The function `pp` takes an implicit module argument of type `Pretty_printer` and applies its `pp` method to format and print the given value `x`. The module `Pp_int` supplies an implicit instance for `int`.

```
module type Pretty_printer = sig
  type t
  val pp : Format.formatter → t → unit
end

implicit module Pp_int = struct
  type t = int
  let pp = Int.pp
end

let pp { Pp : Pretty_printer } ppf t = Pp.pp ppf t

let () = pp 1337
```

Figure 4.3: Example usage of modular implicits for a pretty printer library that prints 1337.

Research direction (Should) We believe omnidirectional type inference could serve as a constraint-based approach to implement principal implicit parameter resolution in the presence of let-generalisation.

Specifically, we aim to extend our constraint language to typecheck a calculus for untyped implicits, inspired by COCHIS [39], with implicit abstraction $\lambda_?\tau.\ e$ and resolution $?\tau$. While performing resolution, suspended constraints could be used to wait until the shape of the implicit type τ is known, which in turn would permit us to prune the search space – eventually resulting in a unique solution or an ambiguity error.

For instance, the pretty-printing example is encoded as:

$$\begin{aligned} \textbf{implicit}\ e_1 : \tau\ \ \textbf{with}\ \ e_2 &\triangleq (\lambda_?\tau.\ e_2)\ \ \textbf{with}\ \ e_1 \\ \alpha\ \texttt{pretty\_printer} &\triangleq \{\texttt{pp} : \texttt{Format.formatter} \to \alpha \to \textbf{unit}\} \end{aligned}$$

$$\begin{aligned} &\textbf{implicit}\ \{\texttt{pp} = \texttt{Int.pp}\} : \textbf{int}\ \texttt{pretty\_printer}\ \ \textbf{with} \\ &\textbf{let}\ \ \texttt{pp} = \lambda_?\alpha\ \texttt{pretty\_printer}.\ \lambda\texttt{ppf}, x : \alpha.\ ?(\alpha\ \texttt{pretty\_printer}).\texttt{pp}\ \texttt{ppf}\ x\ \ \textbf{in} \\ &\texttt{pp}\ ?\_\ 1337 \end{aligned}$$

4.3 Generalised algebraic data types

AML refines Garrigue and Rémy's earlier work but, like theirs, lacks formal treatment of type propagation for GADT pattern matching. Currently, AML requires an explicit type annotation on the scrutinee, a significant limitation compared to others systems such as OutsideIn.

OCaml uses π-directional inference for this: the type of the scrutinee must be known before any type equalities introduced by pattern matching[3]. This mechanism lacks formalisation.

Research direction (Must) We propose using *deep* suspended match constraints to enforce that the matchee's type is fully known before typing the branch. This would be written as:

$$\begin{aligned}[\![\textbf{match } e_1 \textbf{ with Refl} \to e_2 : \alpha]\!] \triangleq \\ &\exists \alpha.\ [\![e_1 : \alpha]\!] \\ &\wedge \textbf{dmatch } \alpha \textbf{ with } (\tau_1 = \tau_2 \to \phi : \tau_1 = \tau_2 \Rightarrow [\![e_2 : \alpha]\!])\end{aligned}$$

However, this inherently delays the typing of the match branches until the scrutinee's type is fully resolved, thereby rejecting:

```
let e2 (type α β) x = match x with Refl → (x : (α, β) eq)
```

Research direction (Could) AML inherits the rigid variable restrictions and unsharing of scopes from Garrigue and Rémy's work. However, these tricks, while standard in π-directional inference, complicates AML's specification, which ultimately results in the dangerous check in Aml-Forall. Identifying a cleaner formulation without such tricks would be preferable.

Research direction (Should) There appears to be an interesting connection between omnidirectional inference and OutsideIn, which relies on delayed implications which mirror (in some respect) our suspended constraints. A promising direction would be to reframe AML using the techniques we've developed for OML.

Research direction (Could) If AML proves both expressive and practical, a compelling avenue for future work is to relate it more explicitly to CMTT. This would involve elaborating well-typed AML terms into a fully explicit CMTT-style calculus with operational

[3]If unknown, the OCaml applies a default rule that introduces the equalities as unification constraints.

semantics, thereby enabling us to establish formal properties such as type safety and to provide a runtime interpretation of ambivalence.

4.4 Timeline

Table 4.1 shows our timeline for the next two years.

Time	Scheduled work
Jun - Sep 2025	Complete work on suspended constraints for static overloading and polytypes. Begin working on submission to POPL 2026 on omnidirectional type inference and suspended constraints. Summer break.
Milestone	*Successfully submitted paper to POPL 2026.*
Oct - Dec 2025	Investigate using suspended constraints for implicit parameter resolution. Continue work on omnidirectional type inference, in particular, default rules.
Jan - Mar 2026	Design and implement constraint-based inference for polymorphic parameters.
Apr - Jun 2026	Begin working on structural polymorphism. Implement a prototype. Start working on submission to POPL 2027 for polymorphic parameters.
Milestone	*Meta-theory for polymorphic parameters established.*
Jul - Sep 2026	Continue working on structural polymorphism, establish main meta-theoretic results. Summer break.
Milestone	*Successfully submitted work on polymorphic paramters to POPL 2027.*
Oct - Dec 2026	Investigate approaches for GADTs with type propagation and surface-level pattern matching constructs.
Jan - Mar 2027	Write up work on structural polymorphism. Select an approach for GADTs, begin working on meta-theory and prototype.
Milestone	*Our work on structural polymorphism is submitted successfully to ICFP 2027.*
Apr - Jun 2027	Continue work on GADTs.
Milestone	*Prototype for GADTs complete.*
Jun - Sep 2027	Complete work on GADTs. Start writing submission for POPL 2028. Summer break.
Milestone	*Meta-theory results for GADTs proven.*

Table 4.1: A timeline for the next two years.

Bibliography

[1] Giuseppe Castagna, Tommaso Petrucciani, and Kim Nguyen. Set-theoretic types for polymorphic variants. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 378–391. ACM, 2016. doi: 10.1145/2951913.2951928. URL `https://doi.org/10.1145/2951913.2951928`.

[2] Luís Damas and Robin Milner. Principal type-schemes for functional programs. In Richard A. DeMillo, editor, *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982*, pages 207–212. ACM Press, 1982. doi: 10.1145/582153.582176. URL `https://doi.org/10.1145/582153.582176`.

[3] Derek Dreyer, Robert Harper, Manuel M. T. Chakravarty, and Gabriele Keller. Modular type classes. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 63–70. ACM, 2007. doi: 10.1145/1190216.1190229. URL `https://doi.org/10.1145/1190216.1190229`.

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

[5] You-Chin Fuh and Prateek Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In Josep Díaz and Fernando Orejas, editors, *TAPSOFT'89: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Barcelona, Spain, March 13-17, 1989, Volume 2: Advanced Seminar on Foundations of Innovative Software Development II and Colloquium on Current Issues in Programming Languages (CCIPL)*, volume 352 of *Lecture Notes in Computer*

Science, pages 167–183. Springer, 1989. doi: 10.1007/3-540-50940-2_35. URL https://doi.org/10.1007/3-540-50940-2_35.

[6] Jacques Garrigue. Programming with polymorphic variants. In *ML workshop*, volume 13. Baltimore, 1998.

[7] Jacques Garrigue. Labeled and optional arguments for objective caml. 03 2001.

[8] Jacques Garrigue. Simple type inference for structural polymorphism. In *The Second Asian Workshop on Programming Languages and Systems, APLAS'01, Korea Advanced Institute of Science and Technology, Daejeon, Korea, December 17-18, 2001, Proceedings*, pages 329–343, 2001.

[9] Jacques Garrigue and JL Normand. Adding gadts to ocaml: the direct approach. In *Workshop on ML*, page 27, 2011.

[10] Jacques Garrigue and Didier Rémy. Semi-explicit first-class polymorphism for ML. *Inf. Comput.*, 155(1-2):134–169, 1999. doi: 10.1006/INCO.1999.2830. URL https://doi.org/10.1006/inco.1999.2830.

[11] Jacques Garrigue and Didier Rémy. Ambivalent types for principal type inference with gadts. In Chung-chieh Shan, editor, *Programming Languages and Systems - 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. Proceedings*, volume 8301 of *Lecture Notes in Computer Science*, pages 257–272. Springer, 2013. doi: 10.1007/978-3-319-03542-0_19. URL https://doi.org/10.1007/978-3-319-03542-0_19.

[12] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979. ISBN 3-540-09724-4. doi: 10.1007/3-540-09724-4. URL https://doi.org/10.1007/3-540-09724-4.

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

[14] Gérard P. Huet. A unification algorithm for typed lambda-calculus. *Theor. Comput. Sci.*, 1(1):27–57, 1975. doi: 10.1016/0304-3975(75)90011-0. URL https://doi.org/10.1016/0304-3975(75)90011-0.

[15] Mark P. Jones. A theory of qualified types. In Bernd Krieg-Brückner, editor, *ESOP '92, 4th European Symposium on Programming, Rennes, France, February 26-28, 1992, Proceedings*, volume 582 of *Lecture Notes in Computer Science*, pages 287–306. Springer, 1992. doi: 10.1007/3-540-55253-7_17. URL https://doi.org/10.1007/3-540-55253-7_17.

[16] Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly types: type inference for generalised algebraic data types. Technical report, Technical Report MS-CIS-05-26, Univ. of Pennsylvania, 2004.

[17] Xavier Leroy. *The ZINC experiment: an economical implementation of the ML language*. PhD thesis, INRIA, 1990.

[18] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In Ron K. Cytron and Peter Lee, editors, *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 142–153. ACM Press, 1995. doi: 10.1145/199448.199476. URL `https://doi.org/10.1145/199448.199476`.

[19] Anton Lorenzen, Leo White, Stephen Dolan, Richard A Eisenberg, and Sam Lindley. Oxidizing ocaml with modal memory management. *Proceedings of the ACM on Programming Languages*, 8(ICFP):485–514, 2024.

[20] Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978. doi: 10.1016/0022-0000(78)90014-4. URL `https://doi.org/10.1016/0022-0000(78)90014-4`.

[21] Robin Milner, Mads Tofte, and Robert Harper. *Definition of standard ML*. MIT Press, 1990. ISBN 978-0-262-63132-7.

[22] Aleksandar Nanevski. Meta-programming with names and necessity. In Mitchell Wand and Simon L. Peyton Jones, editors, *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*, pages 206–217. ACM, 2002. doi: 10.1145/581478.581498. URL `https://doi.org/10.1145/581478.581498`.

[23] Martin Odersky, Philip Wadler, and Martin Wehr. A second look at overloading. In John Williams, editor, *Proceedings of the seventh international conference on Functional programming languages and computer architecture, FPCA 1995, La Jolla, California, USA, June 25-28, 1995*, pages 135–146. ACM, 1995. doi: 10.1145/224164.224195. URL `https://doi.org/10.1145/224164.224195`.

[24] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory Pract. Object Syst.*, 5(1):35–55, 1999.

[25] Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Trans. Program. Lang. Syst.*, 17(6):844–895, 1995. doi: 10.1145/218570.218572. URL `https://doi.org/10.1145/218570.218572`.

[26] Benjamin C. Pierce and David N. Turner. Local type inference. In David B. MacQueen and Luca Cardelli, editors, *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, pages 252–265. ACM, 1998. doi: 10.1145/268946.268967. URL https://doi.org/10.1145/268946.268967.

[27] François Pottier. Hindley-milner elaboration in applicative style: functional pearl. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 203–212. ACM, 2014. doi: 10.1145/2628136.2628145. URL https://doi.org/10.1145/2628136.2628145.

[28] François Pottier and Yann Régis-Gianas. Stratified type inference for generalized algebraic data types. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 232–244. ACM, 2006. doi: 10.1145/1111037.1111058. URL https://doi.org/10.1145/1111037.1111058.

[29] François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced topics in types and programming languages*, pages 389–490. MIT Press, 2004.

[30] Jacques Garrigue Xuanrui Qi. Formalizing ocaml gadt typing in coq. 2021.

[31] Didier Rémy. Typechecking records and variants in a natural extension of ML. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 77–88. ACM Press, 1989. doi: 10.1145/75277.75284. URL https://doi.org/10.1145/75277.75284.

[32] Didier Rémy. Typing record concatenation for free. In Ravi Sethi, editor, *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*, pages 166–176. ACM Press, 1992. doi: 10.1145/143165.143202. URL https://doi.org/10.1145/143165.143202.

[33] Didier Rémy. 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, 1992. URL http://gallium.inria.fr/~remy/ftp/eq-theory-on-types.pdf.

[34] Didier Rémy and Jerome Vouillon. Objective ML: A simple object-oriented extension of ML. In Peter Lee, Fritz Henglein, and Neil D. Jones, editors, *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, pages 40–53. ACM Press, 1997. doi: 10.1145/263699.263707. URL https://doi.org/10.1145/263699.263707.

[35] Didier Rémy and Jerome Vouillon. Objective ML: an effective object-oriented extension to ML. *Theory Pract. Object Syst.*, 4(1):27–50, 1998.

[36] John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965. doi: 10.1145/321250.321253. URL https://doi.org/10.1145/321250.321253.

[37] Claudio V. Russo. *Types for Modules*, volume 60 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2004. doi: 10.1016/S1571-0661(05)82620-9. URL https://www.sciencedirect.com/journal/electronic-notes-in-theoretical-computer-science/vol/60/suppl/C.

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

[39] Tom Schrijvers, Bruno C. d. S. Oliveira, Philip Wadler, and Koar Marntirosian. COCHIS: stable and coherent implicits. *J. Funct. Program.*, 29:e3, 2019. doi: 10.1017/S0956796818000242. URL https://doi.org/10.1017/S0956796818000242.

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

[41] Vincent Simonet and François Pottier. Constraint-Based Type Inference for Guarded Algebraic Data Types. Research Report RR-5462, INRIA, 2005. URL https://inria.hal.science/inria-00070544.

[42] Vincent Simonet and François Pottier. A constraint-based approach to guarded algebraic data types. *ACM Trans. Program. Lang. Syst.*, 29(1):1, 2007. doi: 10.1145/1180475.1180476. URL https://doi.org/10.1145/1180475.1180476.

[43] Martin Sulzmann, Tom Schrijvers, Peter Stuckey, and Katholieke Leuven. Type inference for gadts via herbrand constraint abduction. Technical report, 01 2008. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.142.4392.

[44] Don Syme. The early history of F#. *Proceedings of the ACM on Programming Languages*, 4(HOPL):1–58, 2020.

[45] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975. doi: 10.1145/321879.321884. URL https://doi.org/10.1145/321879.321884.

[46] Dimitrios Vytiniotis, Stephanie Weirich, and Simon L. Peyton Jones. Boxy types: inference for higher-rank types and impredicativity. In John H. Reppy and Julia Lawall, editors, *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006*, pages 251–262. ACM, 2006. doi: 10.1145/1159803.1159838. URL https://doi.org/10.1145/1159803.1159838.

[47] Dimitrios Vytiniotis, Simon L. Peyton Jones, Tom Schrijvers, and Martin Sulzmann. Outsidein(x) modular type inference with local assumptions. *J. Funct. Program.*, 21 (4-5):333–412, 2011. doi: 10.1017/S0956796811000098. URL https://doi.org/10.1017/S0956796811000098.

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

[49] Mitchell Wand. Type inference for record concatenation and multiple inheritance. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*, pages 92–97. IEEE Computer Society, 1989. doi: 10.1109/LICS.1989.39162. URL https://doi.org/10.1109/LICS.1989.39162.

[50] Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Inf. Comput.*, 93(1):1–15, 1991. doi: 10.1016/0890-5401(91)90050-C. URL https://doi.org/10.1016/0890-5401(91)90050-C.

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

[52] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In Alex Aiken and Greg Morrisett, editors, *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*, pages 224–235. ACM, 2003. doi: 10.1145/604131.604150. URL `https://doi.org/10.1145/604131.604150`.