Alistair O'Brien

Typing OCaml in OCaml: a constraint-based approach

PhD Proposal

Queens' College

May 22, 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" [8]. 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 make 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 emphasize 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!

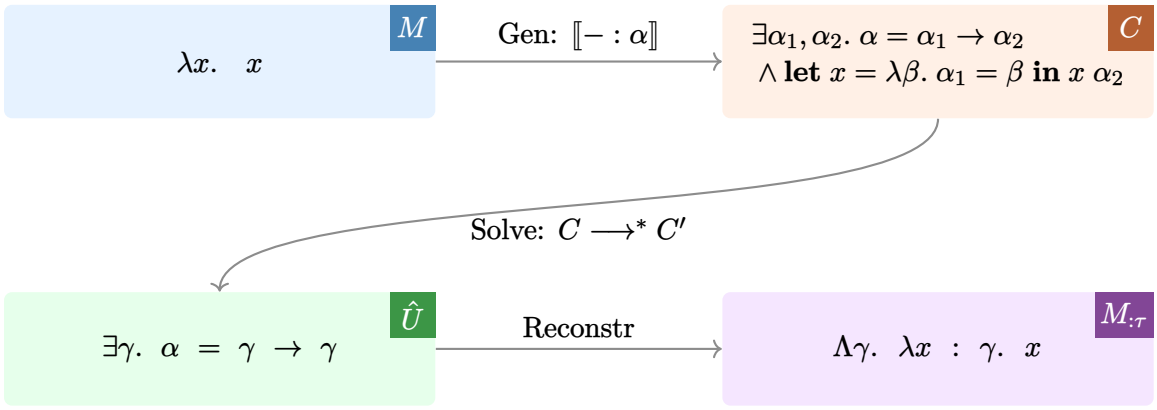
Many functional languages employ type inference algorithms derived from the Hindley-Milner type system [8, 4]. This system supports implicit *let-polymorphism* and is traditionally implemented by algorithm $\mathcal{W}$ [8] or its variants such as $\mathcal{J}$ [8]. These algorithms use unification and 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 [13], is deceptively straightforward: for some

arbitrary term M, we generate a constraint C such that M is well-typed if and only if C is satisfiable. Once C has been solved, we construct an explicitly typed version of M, written $M_{:\tau}$ using C's solution.



By framing type inference as a constraint-solving problem, our thesis aims to develop inference algorithms that are not only more flexible and performant, but also easier to reason about and maintain. We aim to verify our approach scales by developing and formalising a constraint language and solver capable of type checking a significant subset of the OCaml language, including many advanced features.

Organisation This report is structured as follows:

1. Chapter 2 introduces the necessary background on ML-style type inference, offers a broad overview of OCaml's features, and presents a formal basis for constraint-based type inference.

2. Chapters 3 and 4 report our initial progress on our project.

   Chapter 3 presents our initial work on *omnidirectional type inference*, a novel technique for type inference in the presence of features that rely on type-based disambiguation, such as overloading and first-class polymorphism.

   Chapter 4 presents a principal and easy constraint-based approach for the type inference of generalized algebraic data types, a notoriously difficult problem.

3. Finally, Chapter 5 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 3 and 4. Chapter 2 includes a high-level overview of OCaml – not as a complete review, but to situate our work within the broader design landscape.*

2 Background

In this chapter, we set the stage for the calculi and type systems presented in later chapters. Section 2.1 reviews the ML calculus and the Hindley-Milner [8, 4, 1] type system. Section 2.2 presents a constraint-based approach to type-inference for the ML calculus in the style of Pottier and Rémy. Here we pay particular attention as it serves as a basis for extensions in later chapters. We also outline the main features of OCAML's type system, with references to the relevant literature.

Notation We write $\overline{e}$ for a (possible empty) set of elements $\{e_1, ..., e_n\}$ and a (possibly empty) sequence $e_1, ..., e_n$. The interpretation of whether $\overline{e}$ is a set or a sequence is often implicit. We write $\overline{e_1} \# \overline{e_2}$ as a shorthand for when $\overline{e_1} \cap \overline{e_2} = \emptyset$. We write $\overline{e_1}, \overline{e_2}$ as the union or concatenation (depending on the interpretation) of $\overline{e_1}$ and $\overline{e_2}$. We often write e for the singleton set (or sequence).

2.1 The Hindley-Milner type system

The Hindley-Milner (HM) type system was independently developed by Hindley [8] and Milner [8]. Hindley's work arose in the context of formal logic, whereas Milner introduced similar ideas within the setting of the ML programming language, originally developed for the LCF theorem prover [3]. The standard presentation of the HM type system is attributed to Damas and Milner [1]. As a result, the system is also referred to as the Damas-Milner of Damas-Hindley-Milner type system.

Subsequent extensions to the original ML language culminated in the definition of Standard ML (SML) [9], which also marked the beginning of the broader ML family of languages. Among the most notable languages in this family are OCAML [7, 15] and F# [17]. Beyond the ML lineage, the HM type system also served as a foundation for type inference in several other programming languages, including Haskell and Swift.

Syntax The syntax of the ML calculus, presented in Figure 2.1, comprises of terms e formed from variables x, constants c, lambda abstractions $\lambda x.\ e$, applications $e_1\ e_2$, and polymorphic let-bindings $\textbf{let}\ x = e_1\ \textbf{in}\ e_2$. Notably, the calculus does not include any

explicit type annotations; as such, the system is formulated in *Curry-style*, wherein type information is inferred rather than explicitly provided.

Types are split into *monotypes* (often referred to as types) and *type schemes*. Types τ include type variables α and type constructor applications. We assume a fixed set of type constructors F, which at least includes functions types and the unit type; each constructor has a fixed arity, denoted $\mathsf{arity}(\mathsf{F})$. Type schemes σ extend monotypes by allowing the universal quantification of type variables, but only at the outermost level – a restriction known as *prenex polymorphism*. Standard notions such as the set of free type variables $\mathsf{fv}(-)$ and type substitutions $-[\alpha := \tau]$ are defined as usual.

$$\tau \;::=\; \alpha \mid \overline{\tau}\ \mathsf{F} \qquad \text{Types}$$

$$\mathsf{F} \;::=\; \mathbf{unit} \mid \cdot \to \cdot \mid \ldots \qquad \text{Type constructors}$$

$$\sigma \;::=\; \tau \mid \forall \alpha.\ \sigma \qquad \text{Type schemes}$$

$$e \;::=\; x \mid c \mid \lambda x.\ e \mid e_1\ e_2 \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \qquad \text{Terms}$$

$$\Gamma \;::=\; \cdot \mid \Gamma, x : \sigma \mid \Gamma, \alpha \qquad \text{Contexts}$$

$$\textsc{Ml-Var}\ \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \textsc{Ml-Const}\ \frac{c : \sigma \in \Delta}{\Gamma \vdash c : \sigma}$$

$$\textsc{Ml-Fun}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \qquad \Gamma \vdash \tau_1}{\Gamma \vdash \lambda x.\ e : \tau_1 \to \tau_2} \qquad \textsc{Ml-App}\ \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\textsc{Ml-Gen}\ \frac{\Gamma, \alpha \vdash e : \sigma \qquad \alpha \# \Gamma}{\Gamma \vdash e : \forall \alpha.\ \sigma} \qquad \textsc{Ml-Inst}\ \frac{\Gamma \vdash e : \forall \alpha.\ \sigma \qquad \Gamma \vdash \tau}{\Gamma \vdash e : \sigma[\alpha := \tau]}$$

$$\textsc{Ml-Let}\ \frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau}$$

Figure 2.1: Syntax and typing rules of the Ml calculus

Typing The typing judgments $\Gamma \vdash e : \sigma$ assigns a type σ to the term e under the context Γ. The contexts Γ serves two roles: it tracks the types of term variables and introduces fresh polymorphic type variables, representing the current scope. Contexts are ordered and duplicate entires are not permitted. We write Γ, Δ for the concatenation of two contexts

(assuming disjointness holds). The domain of a context Γ is denoted $\mathsf{dom}\,\Gamma$, consists of the term and type variables it binds. For convenience, we often write $\overline{e} \# \Gamma$ as a shorthand for $\overline{e} \# \mathsf{dom}\,\Gamma$.

Most presentations of ML often do not explicitly track the type variables in scope. In contrast, the presentation here uses the type context Γ to contain polymorphic type variables. This additional scoping information is leveraged by the *well-formedness* relation $\Gamma \vdash \tau$ for types, which ensures the correct scoping of type variables in types. The relation is defined in Figure 2.2.

$$\textsc{ML-Wf-TyVar}\ \frac{\alpha \in \Gamma}{\Gamma \vdash \alpha} \qquad \textsc{ML-Wf-TyCon}\ \frac{\forall i,\ \Gamma \vdash \tau_i \qquad \mathsf{arity}(\mathsf{F}) = |\overline{\tau}|}{\Gamma \vdash \overline{\tau}\ \mathsf{F}} \qquad \textsc{ML-Wf-TyScm}\ \frac{\Gamma, \alpha \vdash \sigma \qquad \alpha \# \Gamma}{\Gamma \vdash \forall \alpha.\ \sigma}$$

$$\textsc{ML-Ctx-Emp}\ \frac{}{\cdot\ \mathsf{ctx}} \qquad \textsc{ML-Ctx-Var}\ \frac{\Gamma\ \mathsf{ctx} \qquad \Gamma \vdash \sigma \qquad x \# \Gamma}{\Gamma, x : \sigma\ \mathsf{ctx}} \qquad \textsc{ML-Ctx-TyVar}\ \frac{\Gamma\ \mathsf{ctx} \qquad \alpha \# \Gamma}{\Gamma, \alpha\ \mathsf{ctx}}$$

Figure 2.2: Well-formedness of ML types, type schemes and contexts

We implicitly assume the well-formedness of the typing context Γ in the typing rules. ML-Var retrieves a the type scheme $x : \sigma$ from the context Γ. Similarly, constants are typed using an implicit typing context Δ from which we select their type schemes (ML-Const). Function types are introduced via lambda abstractions: in ML-Lam, the system guesses a well-formed type τ_1 for the type of x, typechecks the body e is under the extended context $\Gamma, x : \tau_1$ producing the return type τ_2, and assigns the abstraction the function type $\tau_1 \to \tau_2$. Conversely, function types are eliminated by applications; in ML-App, the type of the argument must match the function's parameter type τ_1 and application returns the type τ_2.

ML-Gen and ML-Inst correspond to implicit *generalization* and *instantiation* respectively. Generalization universally quantifies a type variable α, introducing it as a fresh polymorphic variable in the typing context. In ML-Inst, we specialize a type scheme σ by substituting the universally bound type variables for an arbitrary (guessed) type.

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

2.1.1 Type inference

The declarative type system for ML, shown in Figure 2.1, does not immediately yield a type inference algorithm. This is primarily due to two reasons: the system is not syntax-directed, and it relies on non-deterministic reasoning, such as the need to guess types in the ML-LAM rule.

Syntax-directed system A type system is said to be syntax-directed if each syntactic form corresponds to a unique typing rule. The original ML rules violate this property: generalization (ML-GEN) and instantiation (ML-INST) can be applied at arbitrary points, introducing ambiguity in derivations.

However, by observing that generalization is only necessary at let-bindings and that instantiation is required when variables or constants are looked up, we can refactor the system accordingly. Yielding a sound and complete syntax-directed presentation of the ML type system, where type schemes are instantiated eagerly at variables and constants, and generalization is performed only at let-bindings.

$$\textsc{ML-Var-Inst}\quad \frac{x : \forall \overline{\alpha}.\, \tau \in \Gamma \qquad \Gamma \vdash \overline{\tau'}}{\Gamma \vdash x : \tau[\overline{\alpha := \tau'}]} \qquad \textsc{ML-Const-Inst}\quad \frac{c : \forall \overline{\alpha}.\, \tau \in \Delta \qquad \Gamma \vdash \overline{\tau'}}{\Gamma \vdash c : \tau[\overline{\alpha := \tau'}]}$$

$$\textsc{ML-Let-Gen}\quad \frac{\Gamma, \overline{\alpha} \vdash e_1 : \tau_1 \qquad \overline{\alpha} \# \Gamma \qquad \Gamma, x : \forall \overline{\alpha}.\, \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau_2}$$

Type inference Type inference for ML is captured by algorithm $\mathcal{W}$ [8], which traverses the term e, introduces fresh type variables for unknown (guessed) types, and accumulates type constraints based on the syntax-directed rules above. These constraints are solved via *unification* [16] to produce a consistent type assignment. At let-bindings, the algorithm performs generalization by quantifying over type variables not free in the context, enabling polymorphism; instantiation then occurs at variable and constant lookup. Variants such as $\mathcal{J}$ [8] follow the same structure, framing inference as a interleaved combination of *constraint generation* and *solving*.

2.1.2 Principal type schemes

A defining feature of ML type inference is that it enjoys the existence of *principal types schemes*, that is to say every well-typed term has a 'most general' type scheme. To formalize this notion, we first introduce the instantiation relation among type schemes,

shown in Figure 2.3.

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \tau \leq \tau}\ \textsc{Ml-}\leq\textsc{-Refl} \qquad \frac{\Gamma \vdash \tau \qquad \Gamma \vdash \sigma_1[\alpha := \tau] \leq \sigma_2}{\Gamma \vdash \forall \alpha.\ \sigma_1 \leq \sigma_2}\ \textsc{Ml-}\leq\textsc{-ForallL}$$

$$\frac{\alpha \# \Gamma \qquad \Gamma, \alpha \vdash \sigma_1 \leq \sigma_2}{\Gamma \vdash \sigma_1 \vdash \forall \alpha.\ \sigma_2}\ \textsc{Ml-}\leq\textsc{-ForallR}$$

Figure 2.3: Instantiation relation for Ml type schemes

The judgement $\Gamma \vdash \sigma_1 \leq \sigma_2$ expresses that σ_2 is a instance of σ_1, indicating that σ_1 is *more general* than σ_2. That is, for any concrete instantiation of σ_2, we can find an instantiation of σ_1 that yields the same type.

Ml-$\leq$-Refl trivially asserts that a monotype is an instance of itself. In Ml-$\leq$-ForallL, a universally quantified type on the left may be instantiated by substituting α with any monotype to match the right hand side. Conversely, Ml-$\leq$-ForallR has a type scheme $\forall \alpha.\ \sigma$ on the right hand side. In this case, we introduce a fresh polymorphic type variable α into the context to test whether the σ_1 is an instance of σ_2 under any instantiation of α – this is essentially a form of skolemization

With the machinery in place, we can now formally state that Ml enjoys *principality*: every typable term admits a most general type scheme from which all other typings can be instantiated.

Theorem 1. *If $\Gamma \vdash e : \sigma$, then there exists σ' such that $\Gamma \vdash e : \sigma'$, and for all σ'' such that $\Gamma \vdash e : \sigma''$, we have $\Gamma \vdash \sigma' \leq \sigma''$.*

For example, the identity function $\lambda x.\ x$ has the principal type scheme $\forall \alpha.\ \alpha \to \alpha$, which is more general than any typings, such as $\mathbf{unit} \to \mathbf{unit}$, $(\mathbf{unit} \to \mathbf{unit}) \to (\mathbf{unit} \to \mathbf{unit})$, etc.

2.2 Constraint-based type inference

The Hm type system and algorithm $\mathcal{W}$ established practical foundation for the type inference of languages with parametric polymorphism. As interest grew in supporting features like extensible records [14, 20], overloading [19, 10] and subtyping [2], constraint-based approaches to inference begin to emerge in the early 1990s.

These early systems were often tightly coupled to specific language features. A shift towards generality came with frameworks like Jones' *qualified types* [6] and Odersky et

al.'s HM(X) [11], parameterizing inference over a constraint domain X. Their algorithm guarantees soundness and completeness assuming basic properties like decidability of constraint satisfiability.

This approach has seen real-world adoption, notably in the GHC Haskell compiler which leverages this approach to provide a multitude of configurable type system extensions.

Pottier and Rémy [13] later further refined this paradigm by fully decoupling constraint generation from solving. Rather than interleave constraint generation and solving, they translate the entire program into a single constraint whose satisfiability characterises typability. Pottier [12] later describes how type reconstruction may proceed from the solutions to this constraint, yielding a clean, modular inference architecture.

We now present their system for constraint-based type inference for the ML calculus, which forms the basis for all subsequent extensions in later chapters.

2.2.1 Constraint language

The interface between constraint generation and solving is the *constraint language*. Pottier and Rémy's [13] key insight for the reduction of type inference to constraint solving is the inclusion of *term variables* in their constraint language. It comprises of constraint forms that bind term variables and forms that use them.

Syntax The syntax is given in Figure 2.4. The constraint language contains tautologic constraints (**true**), unsatisfiable constraints (**false**) and logical conjunctions ($C_1 \wedge C_2$). $\tau_1 = \tau_2$ asserts that the types τ_1 and τ_2 are equal. The constraint form $\exists\alpha.\ C$ binds an existentially quantified type variable in C.

The remaining two constructs 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 application constraints $x\ \tau$ which applies the type τ is the abstraction bound to x.

The well-formedness of constraints is expressed by the judgement $\Xi \vdash C$. This is required for meta-theoretic purposes and is defined in the standard way: it ensures that all term and type variables occurring in C are either bound within C or appear in Ξ.

Semantics Constraints are a *formal logic*: a syntax with a semantic interpretation. We define the meaning of constraints via a satisfiability relation, written $\Phi \vdash C$, which states when an assignment Φ (sometimes referred to as a model) satisfies a constraint C.

$$
\begin{array}{rcll}
C & ::= & \mathbf{true} \mid \mathbf{false} \mid C_1 \wedge C_2 \mid \exists \alpha.\, C \mid \tau_1 = \tau_2 & \text{Constraints} \\
 & \mid & \mathbf{let}\ x = \Lambda\ \mathbf{in}\ C \mid x\ \tau & \\
\Lambda & ::= & \lambda \alpha.\, C & \text{Constraint abstractions} \\
\Phi & ::= & \cdot \mid \Phi, x : \sigma \mid \Phi, \alpha \mid \Phi, \alpha := \tau & \text{Assignments} \\
\Xi & ::= & \cdot \mid \Xi, x \mid \Xi, \alpha & \text{Constraint contexts}
\end{array}
$$

Figure 2.4: Syntax of constraints – a subset of first-order logic with first-order abstractions

Departing from Pottier and Rémy's presentation, we adopt an 'open' finite-tree model where types may contain variables, as opposed to the 'closed' Herbrand universe used in their work. This choice simplifies the proofs of metatheoretic properties in our later work, though we conjecture that the corresponding ground semantics is equivalent.

Key to our presentation of an open semantics is the notion of a *ordered assignment*; which defines the scope of existential variables and controls the free variables of their solution. Like typing contexts, assignments Φ are ordered and constraint declarations of polymorphic variables (α) and term variable bindings ($x : \sigma$). Unlike typing contexts, assignments also declarations of existential variables with solutions ($\alpha := \tau$).

$$
\textsc{Assn-Emp}\ \frac{}{\cdot\ \mathsf{assn}} \qquad
\textsc{Assn-Var}\ \frac{\Phi\ \mathsf{assn} \qquad x \# \Phi \qquad \Phi \vdash \sigma}{\Phi, x : \sigma\ \mathsf{assn}} \qquad
\textsc{Assn-TyVar}\ \frac{\Phi\ \mathsf{assn} \qquad \alpha \# \Phi}{\Phi, \alpha\ \mathsf{assn}}
$$

$$
\textsc{Assn-SolvedTyVar}\ \frac{\Phi\ \mathsf{assn} \qquad \alpha \# \Phi \qquad \Phi \vdash \tau}{\Phi, \alpha := \tau\ \mathsf{assn}}
$$

Figure 2.5: Well-formedness rules for ordered assignments

The well-formedness rules for assignments (Figure 2.5) not only prohibit duplicate declarations, but also enforce order: if $\Phi = \Phi_L, \alpha := \tau, \Phi_R$, the solution τ must be well-formed under Φ_L. Consequently, circularity is ruled out: $\alpha := \beta, \beta := \alpha$ is ill-formed.

Assignments may be viewed as parallel substitutions on types, substituting solved existential variables. We write $\{\Phi\}\tau$ for Φ applied as a substitution to the type τ; this operation is defined in Figure 2.6. We extend this substitution to type schemes in the typical capture avoiding way.

To relate assignments to typing contexts, we define an erasure operation $\lfloor - \rfloor$, which discards solutions while applying substitutions to the types in term bindings. This allows us to reuse relations defined earlier, such as the ML instantiation relation. It is also

$$
\begin{aligned}
\{\Phi\}\alpha &= \{\Phi\}\tau && \text{if } \alpha := \tau \in \Phi \\
\{\Phi\}\alpha &= \alpha && \text{otherwise} \\
\{\Phi\}\overline{\tau}\ \mathsf{F} &= \overline{\{\Phi\}\tau}\ \mathsf{F} \\
\{\Phi\}\forall\alpha.\ \sigma &= \forall\alpha.\ \{\Phi\}\sigma && \text{if } \alpha \# \Phi
\end{aligned}
$$

$$
\begin{aligned}
\lfloor\cdot\rfloor &= \cdot \\
\lfloor\Phi, \alpha\rfloor &= \lfloor\Phi\rfloor, \alpha \\
\lfloor\Phi, \alpha := \tau\rfloor &= \lfloor\Phi\rfloor \\
\lfloor\Phi, x : \sigma\rfloor &= \lfloor\Phi\rfloor, x : \{\Phi\}\sigma
\end{aligned}
$$

Figure 2.6: The definition of parallel substitution $\{-\}=$ and assignment erasure $\lfloor - \rfloor$ for constraint assignments

essential for stating the soundness and completeness results for constraint generation in Section 2.2.2.

We now turn to the definition of constraint satisfaction, given in Figure 2.7. The rules for truth (Sat-True) and conjunctions (Sat-Conj) are straightforward. To satisfy a constraint $\tau_1 = \tau_2$ (Sat-Equal), both τ_1 and τ_2 must be well-formed under Φ and application of Φ must make the two types equal. The Sat-Exists rule establishes the constraint is satisfiable if a well-formed type τ can be chosen such that C is satisfied by the extended assignment.

A let constraint (Sat-Let) binds the type variable α in C_1 and the term variable x in C_2. The constraint abstraction (Sat-Abs) is interpreted as a type scheme, produced by generalizing the solution of α. We note that this needn't be the most general type scheme. The application constraint $x\ \tau$ asserts that the type scheme associated with x may be instantiated to yield τ. The resulting rule Sat-App applies the assignment to the type scheme and τ.

2.2.2 Constraint generation

We now present a translation from Ml terms to constraints, such that the resulting constraint is satisfiable if and only if the term is well typed. The translation is defined as a function $[\![e : \tau]\!]$, where e is the term to be translated and τ is the expected type of e.

The expected type τ is permitted to contain type variables, which can be existentially bound in order to perform type inference. The models of constraint $[\![e : \tau]\!]$ interpret the free variables of τ such that τ becomes a valid type of e. For example, to infer the entire type of e we may pick a fresh type variable α for τ.

$$\textsc{Sat-True}\ \frac{}{\Phi \vdash \mathbf{true}} \qquad \textsc{Sat-Equal}\ \frac{\Phi \vdash \tau_1, \tau_2 \qquad \{\Phi\}\tau_1 = \{\Phi\}\tau_2}{\Phi \vdash \tau_1 = \tau_2}$$

$$\textsc{Sat-Conj}\ \frac{\Phi \vdash C_1 \qquad \Phi \vdash C_2}{\Phi \vdash C_1 \wedge C_2} \qquad \textsc{Sat-Exists}\ \frac{\Phi \vdash \tau \qquad \Phi, \alpha := \tau \vdash C \qquad \alpha \# \Phi}{\Phi \vdash \exists \alpha.\ C}$$

$$\textsc{Sat-Let}\ \frac{\Phi \vdash \Lambda \Rightarrow \sigma \qquad \Phi, x : \sigma \vdash C \qquad x \# \Phi}{\Phi \vdash \mathbf{let}\ x = \Lambda\ \mathbf{in}\ C} \qquad \textsc{Sat-App}\ \frac{\Phi \vdash \tau \qquad x : \sigma \in \Phi \qquad \lfloor \Phi \rfloor \vdash \{\Phi\}\sigma \leq \{\Phi\}\tau}{\Phi \vdash x\ \tau}$$

$$\textsc{Sat-Lam}\ \frac{\overline{\alpha}, \beta \# \Phi \qquad \Phi' \vdash \tau \qquad \Phi', \beta := \tau \vdash C}{\Phi \vdash \lambda \beta.\ C}\ (\textsc{Where}\ \Phi' := \Phi,\ \overline{\alpha})$$

Figure 2.7: The inductive rules for semantic interpretation of constraints.

$$\begin{aligned}
[\![x : \tau]\!] &= x\ \tau \\
[\![c : \tau]\!] &= \Delta(c)\ \tau \\
[\![\lambda x.\ e : \tau]\!] &= \exists \alpha_1 \alpha_2.\ \tau = \alpha_1 \to \alpha_2 \wedge \mathbf{let}\ x = \lambda \alpha_3.\ \alpha_3 = \alpha_1\ \mathbf{in}\ [\![e : \alpha_2]\!] && \text{fresh } \alpha_1, \alpha_2 \\
[\![e_1\ e_2 : \tau]\!] &= \exists \alpha_1 \alpha_2.\ \alpha_1 = \alpha_2 \to \tau \wedge [\![e_1 : \alpha_1]\!] \wedge [\![e_2 : \alpha_2]\!] && \text{fresh } \alpha_1, \alpha_2 \\
[\![\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau]\!] &= \mathbf{let}\ x = \lambda \alpha.\ [\![e_1 : \alpha]\!]\ \mathbf{in}\ [\![e_2 : \tau]\!] && \text{fresh } \alpha
\end{aligned}$$

Figure 2.8: The constraint generation translation for ML

The function $[\![- : =]\!]$ is defined in Figure 2.8. We assume that all variables α_i existentially bound by the translation are fresh. Unsurprisingly, variables generate an instantiation constraint. Primitive constructors such as () generate specialized instantiation constraints for their corresponding type schemes bound by some implicit context of let binders that $\Delta(c)$ maps to. Function generate a constraint that binds two fresh flexible type variables for the argument and return types. We use a **let** constraint to bind the argument in the constraint generated for the body of the function. The **let** constraint is monomorphic since α_3 is fully constrained by type variables defined outside the abstraction's scope and therefore cannot be generalized. Function application binds two fresh flexible type variables for the function and argument types and ensures α is the return type of the function. Let terms 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.

The translation is sound and complete with respect to ML's typing rules. That is to say, the term e is well-typed if and only if $[\![e : \alpha]\!]$ is satisfiable.

Theorem 2. *(Constraint generation is sound and complete with respect to ML) Given a well-formed assignment Φ. Then $\Phi \vdash [\![e : \alpha]\!]$ if and only if $\lfloor\Phi\rfloor \vdash e : \{\Phi\}\alpha$.*

2.2.3 Constraint solving

We now present a solver for the constraint language defined in Section 2.2.1, originally presented by Pottier and Rémy [13]. The solver is a stack machine defined in terms of states s and a transition relation between them. Once the solver cannot apply any more transitions, the state is in a solved form, indicating that the constraint under consideration is satisfiable or unsatisfiable.

Machine states The states s of the solver are of the form (S, U, C), consisting of a stack S, a unification problem U, and a in-progress constraint C. The stack S represents the context in which $U \wedge C$ appears. It contains the binding of existential type variables and term variables that may appear in $U \wedge C$. The stack not only contains bindings, but also indicates how to proceed once the current constraint has been solved, akin to a continuation. Stacks are defined as a possibly empty list of stack frames F, which directly correspond to those constraint forms with at least one sub-constraint, and are separated by the cons operator $(S :: F)$.

$$F \quad ::= \quad \square \wedge C \mid \exists\alpha.\ \square \mid \textbf{let}\ x = \lambda\alpha.\ \square\ \textbf{in}\ C \mid \textbf{let}\ x = \hat{\Lambda}\ \textbf{in}\ \square \qquad \text{Frames}$$

$$S \quad ::= \quad \cdot \mid S :: F \qquad \text{Stacks}$$

Figure 2.9: Syntax of stack frames and stacks

The overall stack can be seen as a constraint with a hole in which $U \wedge C$ is placed. This gives rise to the $S[-]$ operator for plugging a constraint into the hole. It is defined as follows.

$$\begin{aligned}
\cdot[C] &= C \\
(S :: \exists\alpha.\ \Box)[C] &= S[\exists\alpha.\ C] \\
(S :: \mathbf{let}\ x = \lambda\alpha.\ \Box\ \mathbf{in}\ C_1)[C_2] &= S[\mathbf{let}\ x = \lambda\alpha.\ C_1\ \mathbf{in}\ C_2] \\
(S :: \mathbf{let}\ x = \hat{\Lambda}\ \mathbf{in}\ \Box)[C] &= S[\mathbf{let}\ x = \hat{\Lambda}\ \mathbf{in}\ C]
\end{aligned}$$

$$\textsc{Stk-Emp}\ \frac{}{\cdot\ \mathsf{stk}} \qquad \textsc{Stk-Conj}\ \frac{S\ \mathsf{stk} \qquad \mathsf{sctx}(S) \vdash C}{S :: \Box \wedge C\ \mathsf{stk}} \qquad \textsc{Stk-Exists}\ \frac{S\ \mathsf{stk} \qquad \alpha \# S}{S :: \exists\alpha.\ \Box\ \mathsf{stk}}$$

$$\textsc{Stk-Let-Abs}\ \frac{S\ \mathsf{stk} \qquad x, \alpha \# S \qquad \mathsf{sctx}(S), x \vdash C}{S :: \mathbf{let}\ x = \lambda\alpha.\ \Box\ \mathbf{in}\ C\ \mathsf{stk}} \qquad \textsc{Stk-Let}\ \frac{S\ \mathsf{stk} \qquad x \# S \qquad \mathsf{sctx}(S) \vdash \hat{\Lambda}}{S :: \mathbf{let}\ x = \hat{}\mathbf{in}\ \Lambda\Box\ \mathsf{stk}}$$

$$\mathsf{tyv}(S) = \begin{cases} \emptyset & \text{if } S = \cdot \\ \mathsf{tyv}(S'), \alpha & \text{if } S = S' :: \exists\alpha.\ \Box \\ & \text{or } S = S' :: \mathbf{let}\ x = \lambda\alpha.\ \Box\ \mathbf{in}\ C \\ \mathsf{tyv}(S') & \text{otherwise } S = S' :: \_ \end{cases}$$

$$\mathsf{tmv}(S) = \begin{cases} \emptyset & \text{if } S = \cdot \\ \mathsf{tmv}(S'), x \mapsto \hat{\Lambda} & \text{if } S = S' :: \mathbf{let}\ x = \hat{\Lambda}\ \mathbf{in}\ \Box \\ \mathsf{tmv}(S') & \text{otherwise } S = S' :: \_ \end{cases}$$

Figure 2.10: Stack well-formedness rules and definition of term and type variable contexts for stacks

The states are closed, meaning that S must bind all free variables in $U \wedge C$. To state this precisely, we rely on the well-formedness judgements shown in Figure **??**. We write $\mathsf{tmv}(S)$ and $\mathsf{tyv}(S)$ for the sets of term and type variables in the stack S, respectively. The constraint context synthesized from S, denoted $\mathsf{sctx}(S)$, consists of these variables and is defined as $\mathsf{dom}\ \mathsf{tmv}(S), \mathsf{tyv}(S)$. For a state (S, U, C) to be well-formed ($(S, U, C)\ \mathsf{st}$), we require that S is well-formed ($S\ \mathsf{stk}$), U and C are well-formed with respect to $\mathsf{sctx}(S)$.

Unification Unification is the process of solving equations between terms. It was first introduced by Robinson [16] in the context of first-order logic. Later work by Huet [5]

improved the efficiency of unification to quasi-linear time, relying on Tarjan's union-find data structure [18].

Our constraints are eventually reduced to equations between types. We express these reduced constraints as a unification problem U, which form a subset of constraints. Following Pottier and Rémy, we replace binary equality constraints with *multi-equations* – equations involving an arbitrary number of types, as opposed to exactly two. We identify multi-equations up to permutations of their members.

$$U \quad ::= \quad \mathbf{true} \mid \mathbf{false} \mid U_1 \wedge U_2 \mid \exists \alpha.\, U \mid \epsilon \qquad \text{Unification problems}$$

$$\epsilon \quad ::= \quad \tau \mid \epsilon = \tau \qquad \text{Multi-equations}$$

Figure 2.11: Syntax of unification problems

The specification of the unification algorithm as a non-deterministic transition relation between unification problems $U_1 \longrightarrow U_2$ is given in Figure 2.12. Rewriting is performed modulo α-equivalence, commutativity of conjunction and under an arbitrary unification problem context $\mathcal{U}$.

U-Exists rule lifts all existential quantifiers to the top of the unification problem, allowing all multi-equations to become part of a single conjunction. This enables applications of the U-Merge and U-Cycle rules. U-Merge identifies two multi-equations that share a common variable and merges them into a single multi-equation. U-Stutter eliminates repeated variables in a multi-equation. U-Decomp decomposes an equation between two types whose type constructors match, relying on injectivity of the type constructors. U-Clash complements the U-Decomp rule by handling the case where two types have differing constructors, in Ml, such an equation is unsatisfiable. U-Name introduces a new existential variable for a type variable for an argument of a type constructor, ensuring unification operates over shallow types, making sharing of type variables explicit and avoids the need for copying types in rules such as U-Decomp.

U-Cycle performs the *occurs check* to ensure that a type variable does not occur in the type it is being unified with. This is a necessary condition for unification, as it would otherwise lead to infinite types. We say that a type variable α occurs in β with respect to a unification problem U, denoted $\alpha \prec_U \beta$, if U contains a multi-equation of the form $\tau = \beta = \ldots$ where τ is a non-variable type that contains α. A unification problem is *cyclic*, written $\mathsf{cyclic}(U)$, if $\alpha \prec_U^* \alpha$ for some α.

U-False propagates unsatisfiability. U-True removes trivial constraints from the unification problem.

The solved forms (i.e normal forms) of the unification problems is a problem $\hat{U}$ which is either **false** or a conjunction of multi-equations $\exists \overline{\alpha}.\ \bigwedge_{i=1}^{n} \epsilon_i$ where every multi-equation contains at most one non-variable type, no two multi-equations share a variable, and $\hat{U}$ is

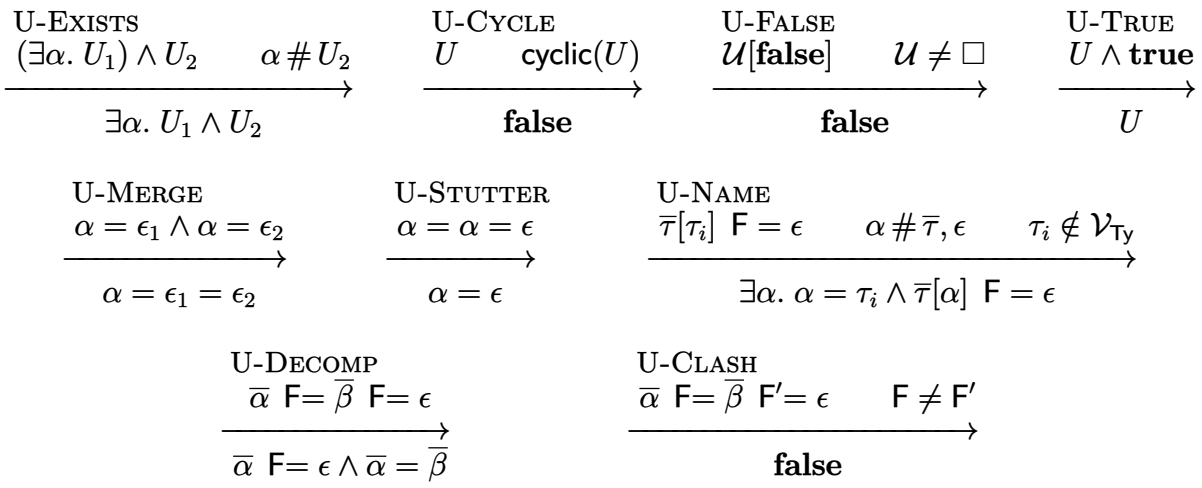


Figure 2.12: Unification algorithm as a series of rewrite rules $U_1 \longrightarrow U_2$

acyclic.

The rewriting system enjoys the following properties. First, unification is strongly normalizing, meaning that every unification problem has a unique solved form. Secondly, unification preserves equivalence, meaning that if $U_1 \longrightarrow U_2$ then U_1 and U_2 are semantically equivalent.

Lemma 3. *The rewriting relation $U_1 \longrightarrow U_2$ is strongly normalizing.*

Lemma 4. *If $U_1 \longrightarrow U_2$, then $U_1 \equiv U_2$.*

Solving rules As mentioned, the solver is defined as a transition system over states $s_1 \longrightarrow s_2$. The transition relation is defined in Figure 2.13.

The initial state of the solver is of the form $(\cdot, \mathbf{true}, C)$, where C is the constraint to be solved. The solver proceeds by applying the transition rules to the current state until it reaches a solved form, either $(S, \mathbf{false}, C)$ or $(\mathcal{X}, \hat{U}, \mathbf{true})$ where $\mathcal{X}$ is a stack of existential variables and $\hat{U}$ is a solved form of the unification problem.

Rewriting rules are performed modulo α-equivalence. S-Unify applies the unification algorithm to the current unification problem. The unification algorithm itself is treated as a block box by the solver. This permits on to parametrize the solver over the equational theory of types implemented by the unification algorithm.

S-Exists-U, S-Exists-Conj and S-Exists-Let lift existential quantifiers to the nearest enclosing **let** frame, if any, or to the top of the stack. The side conditions prevent the capture of type variables in the unification problem U.

S-Solve-False propagates the unsatisfiability of the constraint to be solved to the unification problem, producing a normal form. S-Solve-Equal discovers an equation be-

$$\textsc{S-Unify}\ \frac{S, U_1, C \qquad U_1 \longrightarrow U_2}{S, U_2, C} \qquad \textsc{S-Exists-U}\ \frac{S, \exists \overline{\alpha}.\ U, C \qquad \overline{\alpha} \# C}{S :: \exists \overline{\alpha}.\ \square, U, C}$$

$$\textsc{S-Exists-Conj}\ \frac{S_1 :: \square \wedge C_1 :: \exists \overline{\alpha}.\ \square :: S_2, U, C_2 \qquad \overline{\alpha} \# C_1}{S_1 :: \exists \overline{\alpha}.\ \square :: \square \wedge C_1 :: S_2, U, C_2}$$

$$\textsc{S-Exists-Let}\ \frac{S_1 :: \mathbf{let}\ x = \hat{\Lambda}\ \mathbf{in}\ \square :: \exists \overline{\alpha}.\ \square :: S_2, U, C_2 \qquad \overline{\alpha} \# \hat{\Lambda}}{S_1 :: \exists \overline{\alpha}.\ \square :: \mathbf{let}\ x = \hat{\Lambda}\ \mathbf{in}\ \square :: S_2, U, C_2} \qquad \textsc{S-Solve-False}\ \frac{S, U, \mathbf{false}}{S, \mathbf{false}, \mathbf{true}}$$

$$\textsc{S-Solve-Equal}\ \frac{S, U, \tau_1 = \tau_2}{S, U \wedge \tau_1 = \tau_2, \mathbf{true}} \qquad \textsc{S-Solve-Exists}\ \frac{S, U, \exists \alpha.\ C \qquad \alpha \# U}{S :: \exists \alpha.\ \square, U, C} \qquad \textsc{S-Solve-Conj}\ \frac{S, U, C_1 \wedge C_2}{S :: \square \wedge C_2, U, C_1}$$

$$\textsc{S-Solve-Let}\ \frac{S, U, \mathbf{let}\ x = \lambda \alpha.\ C_1\ \mathbf{in}\ C_2 \qquad \alpha \# U}{S :: \mathbf{let}\ x = \lambda \alpha.\ \square\ \mathbf{in}\ C_2, U, C_1} \qquad \textsc{S-Solve-App}\ \frac{S, U, x\ \tau \qquad \mathsf{tmv}(S)(x) = \lambda \alpha.\ U}{S, U, \exists \alpha.\ \alpha = \tau \wedge U}$$

$$\textsc{S-Pop-Conj}\ \frac{S :: \square \wedge C_2, U, \mathbf{true}}{S, U, C_2} \qquad \textsc{S-Pop-Let}\ \frac{S :: \mathbf{let}\ x = \hat{\Lambda}\ \mathbf{in}\ \square, U, \mathbf{true}}{S, U, \mathbf{true}}$$

$$\textsc{S-Lower}\ \frac{S :: \mathbf{let}\ x = \lambda \alpha.\ \square\ \mathbf{in}\ C_2 :: \exists \overline{\beta} \overline{\gamma}.\ \square, U, \mathbf{true} \qquad \overline{\gamma} \# C_2 \qquad \exists \alpha \overline{\beta}.\ U\ \mathsf{determines}\ \overline{\gamma}}{S :: \exists \overline{\gamma}.\ \square :: \mathbf{let}\ x = \lambda \alpha.\ \square\ \mathbf{in}\ C_2 :: \exists \overline{\beta}.\ \square, U, \mathbf{true}}$$

$$\textsc{S-Gen}\ \frac{S :: \mathbf{let}\ x = \lambda \alpha.\ \square\ \mathbf{in}\ C_2 :: \exists \overline{\beta}.\ \square, U_1 \wedge U_2, \mathbf{true} \qquad \alpha, \overline{\beta} \# U_1 \qquad \exists \alpha \overline{\beta}.\ U_2 \equiv \mathbf{true}}{S :: \mathbf{let}\ x = \lambda \alpha.\ \exists \overline{\beta}.\ U_2\ \mathbf{in}\ \square, U_1, C_2}$$

Figure 2.13: The transition system for the solver

tween two types in the constraint and adds it to the unification problem. S-SOLVE-EXISTS produces a new existential frame. S-SOLVE-CONJ solves a conjunction by arbitrarily choosing one of the conjuncts to solve first. Similarly, S-SOLVE-LET solves the constraint abstraction of the **let** constraint by producing a new **let** frame. The ensures that the constraint abstraction is solved before solving the body of the **let** constraint. This solved form is leveraged by the S-SOLVE-APP rule, which solves the application constraint by applying the constraint abstraction bound to x in S. Using solved constraint abstractions avoids duplication of effort in solving the same constraint multiple times.

S-POP-CONJ and S-POP-LET shrink the stack by popping the top frame. The former pops a conjunction frame once the left conjunct has been solved, continuing to solve the right conjunct. The latter pops a **let** frame once the body of the **let** constraint has been solved, and U cannot contain term variables. So the frame is no longer needed.

The remaining rules deal with **let** frames. S-GEN is meant to be applied once the constraint abstraction of the **let** frame has been normalized. It splits the current unification problem into two parts U_1 and U_2, where U_2 constrains the polymorphic variables in the constraint abstraction, expressed by $\exists\alpha\overline{\beta}.\ U_2 \equiv \mathbf{true}$, and U_1 is entirely made up of constraints between variables bound in S, expressed by the side condition $\alpha, \overline{\beta} \# U_1$. Note that U_2 may contain existential variables bound in S. The condition that $\exists\alpha\overline{\beta}.\ U_2 \equiv \mathbf{true}$ ensures that the constraint abstraction is satisfiable.

S-LOWER lowers the existential quantifier in the **let** frame to the outer scope, turning polymorphic variables into monomorphic ones. The side condition informally states that $\overline{\gamma}$ are dominated by monomorphic variables in the unification problem, preventing them from being generalized. We refer the reader to [**? ?**] for how to efficiently determine this.

Let us now finally state the properties of the constraint solver. Firstly, the solver terminates.

Theorem 5. *The rewriting system $s_1 \longrightarrow s_2$ is strongly normalizing.*

Secondly, the rewriting process preserves semantic equivalence of the states. We interpret the state S, U, C as the constraint $S[U \wedge C]$.

Theorem 6. *If $S_1, U_1, C_1 \longrightarrow S_2, U_2, C_2$, then $S_1[U_1 \wedge C_1] \equiv S_2[U_2 \wedge U_2]$*

2.3 OCAML

This section offers a concise overview of OCAML and points to relevant references for it's type system. For a more thorough introduction to the language itself, we direct the reader to [??].

OCAML (originally Objective Caml) is a general purpose, high-level programming language combining functional, object-oriented and imperative paradigms. As a member

of the ML family, it features a sophisticated type system with ML-style polymorphism with numerous extensions. 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 parameterization of modules, functioning as dependent functions from modules to modules. Functors are classified as either *generative* (generating abstract types) [??] or *applicative* (preserving and propagating abstract type equalities) [??]. Additionally, modules may also be treated as first-class citizens in the core language (so-called *first-class modules*).

Objects An extension introduced by Rémy and ?? [??] that adds class-based objects using row polymorphism [??]. Subsequently, extended by Garrigue [??] 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 generalized algebraic data types [??], constructor/record label overloading [??], polymorphic variants [??], labelled arguments [??], polymorphic parameters [??] and modal types [??].

3 Omnidirectional type inference

3.1 Motivation

We begin by first introducing *bidirectional typechecking* [??], an alternative approach to Hm for type inference. We discuss the limitations of bidirectional typechecking, providing us with the motivation for *omnidirectional type inference.*

3.1.1 Bidirectional typechecking

Bidirectional typechecking separates typing rules into two *modes*: checking ($\Gamma \vdash e \Leftarrow \tau$) and synthesizing ($\Gamma \vdash e \Rightarrow \tau$). Checking takes Γ, e, τ as inputs and checks if the term e can be typed with τ. Whereas, synthesizing takes Γ, e as inputs and outputs the inferred type of τ if e is well-typed.

Bidirectional typing was popularized by Pierce and Turner's work on local type inference, which was introduced as an alternative to Hm type systems which could easily deal with polymorphic languages with subtyping. Various other authors have proved the effectiveness of bidirectional typechecking in several other settings, including systems with higher-rank polymorphism [??], generalized algebraic data types [??], and dependent types [??].

Most bidirectional typechecking rules follow the *Pfenning recipe* [??]. This recipe gives a guideline on how to bidirectionalze a type assignment system $\Gamma \vdash e : \tau$. The recipe distinguishes introduction from elimination rules. Constructs which correspond to introduction forms are *checked* against a given type, while constructs corresponding to elimination forms *infer* their types. For illustration, Figure ?? gives the bidirectional typing rules for the simply typed lambda calculus, based on this recipe.

The Chk-Unit rule checks that () has the type **unit**. Syn-Var infers the type of x by fetching it from the context. Syn-Annot and Chk-Eq switch the modes of inferences. The Chk-Lam rule checks the term $\lambda x.\ e$ against $\tau_1 \to \tau_2$, the return type τ_2 is used to check e in the extended context.

The Syn-App rule is particularly interesting, since the rule permits types to propagate across adjacent nodes within the syntax tree. For example, in the expression $(\lambda f : \mathbf{unit} \to \mathbf{unit}.\ f\ ())\ (\lambda x.\ x)$, the type of x can be inferred from the type annotation on f *without*

$$\textsc{Chk-Unit}\ \frac{}{\Gamma \vdash () \Leftarrow \mathbf{unit}} \qquad \textsc{Syn-Var}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x \Rightarrow \tau} \qquad \textsc{Chk-Lam}\ \frac{\Gamma, x : \tau_1 \vdash e \Leftarrow \tau_2}{\Gamma \vdash \lambda x.\ e \Leftarrow \tau_1 \to \tau_2}$$

$$\textsc{Syn-App}\ \frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 \Leftarrow \tau_2}{\Gamma \vdash e_1\ e_2 \Rightarrow \tau_2} \qquad \textsc{Syn-Annot}\ \frac{\Gamma \vdash e \Leftarrow \tau}{\Gamma \vdash (e : \tau) \Rightarrow \tau} \qquad \textsc{Chk-Eq}\ \frac{\Gamma \vdash e \Rightarrow \tau_1 \qquad \tau_1 = \tau_2}{\Gamma \vdash e \Leftarrow \tau_2}$$

the use of unification. Specifically if we know that the type of e_1 is a function from $\tau_1 \to \tau_2$, we can check that e_2 has type τ_1. However, this flow of typing information is one-way: in $(\lambda f.\ f\ ())\ (\lambda x.\ x : \mathbf{unit} \to \mathbf{unit})$ the type of the function cannot be inferred from the argument. This fixed directionality is a central limitation of bidirectional typing.

Another issues arises from strict adherence to the Pfenning recipe. The introduction rules for unit literals is supposed to be in checking mode (rule Chk-Unit), not inferred. As a result, even trivial program like () are ill-typed unless annotated $((() : \mathbf{unit})$. In this particular case, bidirectional typechecking goes against its original intention of removing burden from programming, since this annotation is unnecessary. Practical systems often sidestep this by duplicating rules:

$$\textsc{Syn-Unit}\ \frac{}{\Gamma \vdash () \Rightarrow \mathbf{unit}}$$

Now we can type check (), but we pay the price with the duplication of typing rules. Worse still, the same criticism applies to other constructs, for example, pairs. This shows an additional drawback of bidirectional typechecking: to minimize annotations, many rules are duplicated for having both the inference and checking mode.

Bidirectionality also struggles with global inference. Consider the following term in Dunfield and Krishnaswami's system (Dk) with higher-rank polymorphism (and pairs):

$$\lambda f.\ (f\ (), (f : \forall \alpha.\ \alpha \to \alpha))$$

In the type assignment system, this term has a clear principal type. Yet in the Dk system, it fails to typecheck due to directionality: we must statically choose a direction for each subterm, even if there is no 'optimal' direction, thus some programs inevitably fall through the cracks.

Binding constructs are another pain point. Since they are niether introduction nor elimination forms, the Pfenning recipe doesn't offer a clear treatment. A common workaround

is to provide both checking and synthesis variants (duplicating rules yet again):

$$\textsc{Syn-Let}\quad \frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 \Rightarrow \tau_2}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \Rightarrow \tau_2} \qquad \textsc{Chk-Let}\quad \frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 \Leftarrow \tau_2}{\Gamma \vdash \textbf{let}\ x = \tau_1\ \textbf{in}\ e_2 \Leftarrow \tau_2}$$

This unfortunately only makes matters worse. Dunfield and Krishnaswami note that the annotability criterion for requiring annotations at redexes is no-longer true in the presence of **let** binders. For instance, consider the expression:

$$(\textbf{let}\ x = ()\ \textbf{in}\ \lambda f.\ f\ x)\ (\lambda x.\ x)$$

In this case, we either must annotate the body of the let expression and use Syn-Let or annotate the entire let expression and use Chk-Let. Thus, for some features, it may not be easy to give programmers a clear guideline for where to place annotations.

3.1.2 Our approach

Bidirectional typechecking is elegant and lightweight. It works well in the presence of many complex features and supports the propagation og type information with minimal annotations. But it suffers from:

1. **Unclear annotability**: especially in the presence of constructs like **let**-bindings.
2. **Rule duplication**: required to make the system practical for type inference.
3. **Fixed directionality**: makes the system brittle to program refactoring and limits inference for otherwise well-typed programs.

We'd like to retain the advantages of bidirectional typing, namely the use of propagated type information and removal of redundant annotations without the above issues.

Our key insight: *types can be inferred in any order.* Traditional algorithms enforce an ordering on the resolution of typing constraints – e.g. inferring the type of let-bound expressions before its uses. But from a constraint-based perspective, this isn't necessary. A sufficiently powerful constraint solver can resolve typing constraints in any order, provided it preserves semantic equivalence of constraints.

We call this approach *omnidirectional type inference.* It requires no division of rules into checking and inference modes, eliminating rule duplication. It also enables the inference of programs that bidirectional systems reject and reduces the need for manual annotations.

The remainder of this chapter formalises omnidirectional type inference and discusses its application to Ml type inference with features like static overloading and first-class polymorphism while trivially retaining properties like *principality.*

3.2 Suspended constraints

The core primitive for omnidirectional type inference is the *suspended match constraint* (often referred to simply as a *suspended constraint*). These are a new form of constraint that allows the type checker to delay the resolution of typing constraints until it has sufficient type information to do so.

A suspended constraint has the the form **match** α **with** f (defined in Figure ??) where:

1. The matchee α is a type variable. The constraint is suspended until α is unified (resolved) to a non-variable type.

2. The handler $f : \mathsf{TyShallow} \to C$ is a function from shallow types to constraints. It encodes the branches of the match constraint. Here, $\mathsf{TyShallow}$ denotes the set of shallow types formed by the grammar $\psi ::= \overline{\alpha}\ \mathsf{F}$.

Informally, the semantics of **match** α **with** f are as follows: the constraint remains suspended until the type variable α is unified to concrete non-variable type $\overline{\beta}\ \mathsf{F}$, at which point the handler is applied to this type $f(\overline{\beta}\ \mathsf{F})$ and the generated constraint is scheduled to be solved. If α is never resolved, the constraint remains unsatisfiable.

$$C \quad ::= \quad \ldots \mid \textbf{match}\ \alpha\ \textbf{with}\ f \qquad \text{Constraints}$$

Figure 3.1: The syntax of suspended match constraints

Suspended match constraints perform only shallow matching on the type of α. However, just as in functional programming languages for algebraic data types, shallow matches are expressive enough to encode deep pattern matches. For instance, the match constraint that waits until α is fully realised (i.e. contains no unification variables) can be encoded as:

$$\begin{aligned}\textbf{dmatch}\ \alpha\ \textbf{with}\ f = \textbf{match}\ \alpha\ \textbf{with}\ \lambda\overline{\beta}\ \mathsf{F}\ .\ \textbf{if}\ |\overline{\beta}| = 0\ \textbf{then}\ f(\mathsf{F}) \\ \textbf{else dmatch}\ \beta_1\ \textbf{with}\ (\lambda\tau_1. \\ \quad \ldots \\ \quad \textbf{dmatch}\ \beta_n\ \textbf{with}\ (\lambda\tau_n.\ f(\overline{\tau}\ \mathsf{F}))\ )\end{aligned}$$

3.2.1 A naïve semantics

The semantics of suspended constraints are best understood from the solver's perspective. We begin by providing an informal description of how our solver processes suspended

match constraints (which we later refine in Section ??), from this we show that a naïve semantics for suspended constraints would be incomplete.

Solver When solving **match** α **with** f, the solver proceeds as follows:

1. If α is already unified to a non-variable type, apply the handler f immediately to the resolved type.
2. If α is still unresolved. enqueue the suspended constraint in a scheduler of pending constraints.
3. During unification, whenever a type variable with a pending suspended constraint becomes resolved, the solver retrieves all constraints waiting on that variable, applies their handlers to the resolved types, and schedules the resulting constraints.

Once all active constraints have been processed, any suspended constraints that remain unresolved indicate failure.

Semantics A naive denotational semantics for suspended constraints would be to simply apply the assignment to α to f. More formally,

$$\frac{\Phi, \overline{\beta} := \mathsf{match}(\Phi, \alpha, \mathsf{F}) \vdash f(\overline{\beta}\ \mathsf{F})}{\Phi \vdash \textbf{match}\ \alpha\ \textbf{with}\ f}\ \textsc{Sat-Match-Naïve}$$

$$\mathsf{match}(\Phi, \alpha, \mathsf{F}) = \overline{\tau} \qquad \text{where } \{\Phi\}\alpha = \overline{\tau}\ \mathsf{F}$$

However, our informal solver would be incomplete with respect to this semantics. We let us consider the constraint $\exists\alpha.$ **match** α **with** $\lambda\_.\ \alpha =$ **unit**. This constraint does not unify α outside the suspended constraint, but once it is resumed, the variable must be unified with **unit**. So our informal semantics suggests that this constraint is unsatisfiable since α is never unified outside the match constraint. Yet our semantics satisfies this constraint with the assignment $\alpha :=$ **unit**: DERIVATION TREE

Moreover, suspended constraints can form complex dependency graphs with cycles. For example, our semantics needs to reject constraints like $\exists\alpha, \beta.$ **match** α **with** $\lambda\_.\ \beta =$ **unit** $\wedge$ **match** β **with** $\lambda\_.\ \alpha =$ **unit**.

Worse still, is that the binding location of α may not even contain the sufficient information to prove that α is realised. For example,

$$\exists\beta.\ \beta = \textbf{unit} \wedge \exists\alpha.\ \alpha = \beta \wedge \textbf{match}\ \alpha\ \textbf{with}\ f$$

While it is not standard to motivate the constraint's semantics from the solver's perspective, we treat the solver as a form of operational semantics, for which we aim to

establish a corresponding denotational semantics. Our view is that these two semantics should coincide. In practice, suspended match constraints are more straightforward to formalise operaitonally, and this operational view guides our design of the operational semantics.

Our conclusion is that this naïve semantics should be rejected in favour for the semantics of our informal solver. Our intent with the suspended constraint is that the blocking variable should not be *guessed* out of thin air (ex ...) but *deduced* from the surrounding context without knowledge of the constraint generated by the handler.

3.2.2 Contextual semantics

The idea, inspired by our specification of OML, is to contextually ensure that the only assignment that can satisfy the entire constraint requires that α is assigned to some type with a unique type constructor.

We begin by motivation the property of principal realisation by some context $\mathcal{C}[-]$. A type variable α is principally realised with the constructor F by the context $\mathcal{C}$ if for all assigments Φ, there exists an assignment Φ' for the hole such that $\Phi \vdash \mathcal{C}[\Phi' \vdash \mathbf{true}]$ implies $\{\Phi'\}\alpha = \overline{\tau}\ \mathsf{F}$.

For example, the above unsatisfiable constraint would have the context $\mathcal{C}[-] = [-]$. This context cannot realise α, thus the constraint is unsatisfiable.

THe semantics for a suspended constraint is to require that some surrounding context princiaplly realises the matchee. We formally write this as

$$\frac{\Phi \vdash \mathcal{C}[\Phi', \overline{\beta} := \mathsf{match}(\Phi', \alpha, \mathsf{F}) \vdash f(\overline{\beta}\ \mathsf{F})] \qquad \mathcal{C}[\alpha\ !] \text{ with constructor } \mathsf{F}}{\Phi \vdash \mathcal{C}[\Phi' \vdash \mathbf{match}\ \alpha\ \mathbf{with}\ f]}\ \textsc{Sat-Match}$$

3.3 Static overloading

This section discusses the application of omnidirectional type inference to *static overloading*. By static overloading, we mean a form of overloading where the overloading must be resolved statically. That is, type inference must statically disambiguate all occurances of overloading.

OCaml features a specialisation of static overloading, called *constructor and record label overloading*. This allows programmers to define data types with overloaded constructor or field names; to resolve the ambiguity, OCaml's type checker relies on type-based disambiguation:

```
type t = A;;
type u = A;;

let x = (A : t) ;;
let y = (A : u) ;;
```

This is predictable and efficient, but still relies on arbitrary directional choices. Making inference brittle to program refactoring and requires the programmer to experiment where they should place a type annotation as there is no clear specification for disambiguation. As shown below:

```
type t = A ;;
type u = A ;;

let x = (A, A : u * t);;
(* The following is ill-typed due to directionality, even though it
  should be a valid transformation *)
let y =
  let z = A in
  (z, A : t * u)
;;
```

This approach suffers from the annotability issue discussed in Section ??.

3.3.1 Kinded(1) types

3.4 First-class polymorphism

3.4.1 Semi-explicit first-class polymorphism

3.4.2 Polymorphic parameters

3.5 Solving suspended constraints

Solver challenges When solving a suspended constraint **match** α **with** f, we wait until α is unified with some type constructor. We could achieve this by simply yielding within the solver, making some progress elsewhere and revisiting the constraint. However, the interaction between let generalisation and suspended constraints makes this harder.

3.5.1 In the absence of let-generalisation

3.5.2 Partial generalisation and instantiation

3.6 Related work

Review Contextual typing

Review constraint handler rules

3.7 Future work

4 Generalized algebraic data types

4.1 Introduction

Generalized algebraic data types (GADTs), introduced by Xi [??], are a simple generalization of algebraic data types that allow one to describe richer constraints between constructors and their types. They are reminiscent of inductive types from the calculus of inductive constructions [??]. In functional programming languages, they have been put to a variety of uses, by many authors, among them [??], [??], [??], and [??].

The canonical example use of generalized algebraic data types is in writing a type-safe evaluator for a simply-typed embedded domain specific language. The algebraic data type `expr`, defined in Figure ??, 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 4.1: The abstract syntax tree and type safe evaluator for a simply-typed language in OCaml using generalized algebraic data types.

An `Int n` has the type `int` in the language, thus its type is `int expr`. `Add` is the constructor for the addition operator, having the type `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)`, and returns the type of the result `β expr`.

The definition allows us to write a type-safe evaluator that does not perform any tagging or untagging of values, also defined in Figure ??. A key mechanism for this is the

use of type-level equalities, introduced in `match` constructs, into the typing context. For instance, in the first case of `eval`, the variable `e`, which is of the type `'a expr`, is known to match the pattern `Int n`, which has the type `int expr`. As a result, the equation `'a = int` must hold within the branch. The added equation is later used to prove that the result of the branch, that is, the integer variable `n`, has the type `'a`, as required by `eval`'s signature.

4.1.1 Type inference

While GADTs are a powerful tool for writing type-safe programs, their inference is not without its challenges. In general, full type inference of GADTs is undecidable [??]. In practice, we can often restrict the type system to a decidable subset by relying on type annotations. However, the type system may still not be principal. Sulzmann et al. [**?**] demonstrated that programs with type-level equalities frequently have more than one principal type.

To illustrate this, we consider the example above. In the `match` branch for `Int n`, the type of `n` is `int`, yet it could be considered to have the type `'a`, where `'a`. Neither type is more general than the other *outside* the branch. This poses various problems. To begin, principality is a central property for efficient type inference since it allows us to make *locally optimal* decisions. Second, should a program have more than one principal type, which should we infer?

In short, "it's difficult".

4.1.2 Previous work

Type inference fo GADTs is a notoriously complex area and remains an active topic of research. Even when focusing solely on this topic, it is not feasible to provide a comprehensive overview of the extensive literature. Instead, we concentrate on works that situate type inference within the HM framework.

Simonet and Pottier show that type inference for HMG(X), an extension of HM(X) with GADTs, can be reduced to satisfiability of constraints in a subset of first-order logic, consisting of equations, conjunctions, existential and universal quantifiers, and implications.

$$C ::= \mathbf{true} \mid \mathbf{false} \mid \tau_1 = \tau_2 \mid C_1 \wedge C_2 \mid \exists \alpha.\, C \mid \forall \alpha.\, C \mid C_1 \Rightarrow C_2$$

Unfortunately, constraint satisfaction for their language is intractable AJOB: Is it undecidable or intractable? Simonet says intractable but their language permits conjunction of implications which is equivalent to simulatenous rigid E-unification problem which is

undecidable. The primary source of complexity lies in the use of universal quantification and implication in the constraint languages. If allowed, If allowed, they can encode negation, since $\neg C \equiv C \Rightarrow \textbf{false}$, significantly complicating solving.

To mitigate this, Simonet and Pottier propose leveraging programmer-supplied type annotations. This enables them to impose a syntactic restriction on implications, termed *rigid implications* $\forall \overline{\alpha}.\ C \Rightarrow D$ where $\mathsf{fv}(C) \subseteq \overline{\alpha}$. Despite this restriction, their constraint generation algorithm still requires programmers to annotate all functions that match on GADTs, specifying both the type of the scrutinee and return type of the match expression.

In addition to this, Simonet and Pottier avoid addressing the problem of principality by allowing implication constraints to appear in type schemes – which further complicates things for our poor programmer.

In a similar vein, Payton Jones at el [??] present a tractable type inference algorithm for GADTs that also relies programmer-supplied annotations. They introduced the notions of *rigid types* and wobbly types to denote type information that is available from type annotations and information which is inferred, respectively. Crucially, only matches on rigid types may introduce type equalities. While effective, wobbly types proved unpredictable in behavior and unwiedly in implementation. This ultimately motivated two cleaner alternatives: stratified type inference and OutsideIn.

Stratified type inference refines the wobbly types approach by cleanly separating the propagation of annotations from inference. The first phase transforms the program into an intermediate language and generates annotations (called *shapes*) for match scrutinees and local assumptions; the second phase performs type inference for the intermediate language. While the overall system remains incomplete due to the heuristic nature of the first phase, the second phase is sound, complete and enjoys principal types.

Both stratified type inference and wobbly types required the programmer to understand entirely new concepts (shapes and rigidity respectively). Yet both fall short of full propagation of type information, leaving the humble programmer with an uneasy question: when and where must I annotate?

OutsideIn also separates propagation and inference, but does so using a constraint-based approach, using *implication constraints* $[\overline{\alpha}](\forall \overline{\beta}.\ C \Rightarrow D)$. Constraint solving proceeds into two phases: the first phase resolves "simple" constraints – everything but non-trivial implications – propagating both annotated and inferred type information; the second tackles the deferred implication constraints. This staging ensures that type inference for match branches involving GADTs is postponed until the second phase, when more is known about the scrutinee and the expected return type.

Their trick for tackling principality lies in the treatment of unification variables: implication constrains must refrain from unifying any global unification variables in the 'closure' $[\overline{\alpha}]$ unless those variables have already been resolved elsewhere. Information

flows only from the outside into the implications conclusion, not vice-versa. Though the inference algorithm produces principal types, the declarative type system it serves does not.

Curiously, the idea of delaying inference echos our own idea of *omnidirectional type inference* (Section ??). Yet Vytiniotis et al. [??] 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 ??.

Garrigue and Rémy introduce *ambivalent types* as a way to reconcile a declarative type system for GADTs with principality. Informally, an *ambivalent type* ζ is a *set of types* that are *equal under the local constraints*. 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. To illustrate this, we consider:

```
type (_, _) eq = Refl : ('a, 'a) eq

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

(*Error message here *)
```

The `then` branch returns `y`, with type `a`, whereas the `else` branch returns a value of type `int`. The resultant type is the ambivalent type `a` $\approx$ `int`, which represents a type that is either `a` or `int`. When existing the scope of the `match` branch, the ambivalent type escapes it scope, since the local equality `a` $=$ `int` is no-longer present in the context!

Ambiguities are eliminated using annotations; for an expression $(e : \tau)$, the expressions e and $(e : \tau)$ may have differing ambivalent types ζ_1, ζ_2, but τ must be included in both – to ensure soundness.

Ambivalent types rely on *sharing* to guarantee the inference of principal types. When instantiating a type scheme $\forall\alpha.\ \zeta$ *without* sharing, we lose the information that *all* copies of α must be synchronized – i.e. ... . Garrigue and Rémy address this by using *labelled types* (Section ??), written ψ^α, where α is used to explicitly share the structure ψ. The catch? Sharing must happen everywhere, as any type may be ambivalent. As a result, every type in the system must carry labels. This requirement complicates matters considerably, demanding a severe departure from the conventional ML presentation and making the formalisation notably harder to grasp.

4.1.3 The road to AML

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 – it permits propagation, but halts at GADT case branches. Ambivalent types go further still: 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 [??].

In developing our approach, we sought to retain the expressivity of ambivalent types while prioritising simplicity: in terms of its formal specification, implementation, and the rules needed to be understood by end users (e.g. where to place annotations). To guide our own design, we identify several – admittedly subjective – criteria our solution ought to satisfy:

1. **Predictable behaviour**: The flow of type information should be intuitive to the everyday programmer.

2. **ML types**: Our ideal solution would be to use the type language of ML, already familiar to programmers. The types of HMG(X) and ambivalent types are potentially confusing for programmers. Wobbly types and stratified inference require an understanding of wobblyness or shapes, concepts alien to most programmmers.

3. **Close to ML type inference**: It should be possible to implement the solution as a simple extension of HM type inference, allowing it to be integrated into existing type inference systems and retaining the spirit of the traditional ML presentation.

In this chapter, we present AML, a new calculus of the inference of GADTs that builds on the core ideas of ambivalent types while drawing inspiration from contextual modal type theory.

Our design is guided by the criteria outlined above. As we will show, AML meets these goals. It uses conventional type notation and introduces only modest, well-motivated extensions to the familiar machinery of HM, making it both approachable and practical for implementation.

4.2 The AML calculus

Contextual modal type theory (CMTT) derives from a contextual variant of intuitionistic modal logic S4 through the annotation of the inference rules with proof terms and a computational interpretation in the style of the Curry-Howard correspondence. Modal

logics in general reason about truth in a universe of possible worlds. In the specific case of S4, the key feature is the propositional constructor $\Box$ (called "necessity" or "box" for short). The proposition $\Box\varphi$ is considered proved in the current world, if we can produce a proof of φ in every possible feature worlds. The derived computational interpretation resulted in type systems for staged metaprogramming.

The contextual variants further grades the necessity constructor with a context of propositions Ψ. The proposition $[\Psi]\varphi$ is considered proved in the current world if we can produce a proof of A in every possible future world using *only* the propositions from Ψ as hypotheses. The computational interpretation considers the type $[\Psi]\tau$ as programs of type τ that admit free variables from the context Ψ, and no others. The typing discipline has obtained calucli for meta-programming with 'open code' [??]. ...

The central judgement of CMTT takes the form $\Gamma; \Delta \vdash e : \tau$, where Γ represents the usual typing context, while Δ represents a modal context containing hypotheses $u :: \tau[\Psi]$. Modal variables u stand for open terms with free variables declared in the context Ψ. The introduction and elimination rules for the contextual modality are given in Figure ??.

$$\textsc{Cmtt-}\Box\textsc{I}\quad \frac{\Psi; \Delta \vdash e : \tau}{\Gamma; \Delta \vdash \mathbf{box}\ \Psi.\ e : [\Psi]\tau} \qquad \textsc{Cmtt-}\Box\textsc{E}\quad \frac{\Gamma; \Delta \vdash e_1 : [\Psi]\tau_1 \qquad \Gamma; \Delta, u :: \tau_1[\Psi] \vdash e_2 : \tau_2}{\Gamma; \Delta \vdash \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2 : \tau_2}$$

$$\textsc{Cmtt-Mvar}\quad \frac{u :: \tau[\Psi] \in \Delta \qquad \Gamma; \Delta \vdash \sigma : \Psi}{\Gamma; \Delta \vdash u[\sigma] : \tau}$$

CMTT provides a natural framework for detecting scope escapes – situations where a term attempts to use variables outside their valid scope. For instances, in staged metaprogramming, a piece of generated code should not reference variables that will not be available when that code is eventually ran. The key insight for our work is that the mechanism used in ambivalent types to prevent ill-typed programs can be elegantly recast using concepts from CMTT.

Aml applies the contextual discipline by taking the context Ψ to be a set of equations – producing an equivalence class of programs ... .

In the ambivalent types system, type equations arising from GADT pattern matching are local to specific branches. The challenge occurs when an ambivalent type "escapes" its context – that is, when a type $\tau_1 \approx \tau_2$ is no-longer considered consistent under the current context because it becomes visible outside the scope where those equations are valid. This is precisely analogous to a scope escape in CMTT. By reformulating ambivalent types in terms of contextual modality, we can express $\tau_1 \approx \tau_2$ as $[\Psi]\tau$ where Ψ contains the local type equalities necessary for $\tau_1 \approx \tau_2$ and τ is some member from their equivalence class.

In AML, we refer to Ψ as a *scope*.

Syntax The syntax of terms is given in Figure ??. AML extends ML terms with an explicit universal quantifier $\Lambda\alpha.\ e$. The literal constructor **Refl** has the type $\tau = \tau$, used to introduce 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.
TODO: Colour the ML constructs in gray

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

The syntax of types is also given in Figure ??. In AML types τ^κ are *kinded*, where κ is the kind of a type τ, denoted by the superscript. We have the usual kind for types (ty), but also a kind for *scopes* (sc). Often the kind of a type is apparent or not relevant and most of the time we will elide the kind to reduce clutter, writing τ for types and Ψ for scopes. For clarity, we use α, β, γ for type variables and ς, ϱ for scope variables.

Types consist of normal ML types, including type variables and type constructors. Type constructors are extended to include the equality witness type ($=$). AML types als include *scoped ambivalent types*, which informally denotes a set of provably equivalent types. Under our equivalence relation, introduced later, $[\Psi]\tau$ is equivalent to a set of types $[\Psi]\tau'$ where τ and τ' are provably equivalent using the equation s in Ψ. An ambivalent type is only coherent (in the context Γ) if the equations in Ψ are in Γ. Otherwise, the scope is said to have escaped the context.

We define a scope as a row of equation names. A scope is either empty $\emptyset$, a polymorphic scope variable ς, or an extension of a scope Ψ with an equation ϕ, written as Ψ, ϕ.

Polymorphic type schemes are defined in a mostly typical ML fashion, generalizing over zero or more variables. However, we extend the notion of polymorphism to also quantify over scope variables as well, introducing a form of scope polymorphism.

Contexts are an extension of ML contexts with polymorphic scope variables and named

type equations. Each type variable bound in the context is associated with a *flexibility* marker φ which can either be rigid (r) or flexible (f). A variable is marked rigid if introduced by an explicit quantifier and flexible if introduced implicitly by generalization. The distiction serves two purposes. First, it identifies explicitly introduced type variables, which we leverage to ensure type annotations only make reference to such variables. The second purpose we explain below in our definition of well-formedness.

Well-formedness Well-formedness for types serves a cruical job in AML. Namely, it is responsible for ensuring that ambivalent types are coherent, thus preventing scope escapes. The definition of well-formedness for types, type schemes and contexts is mostly standard: ensuring types are well-kinded, prevent duplicate bindings in contexts, and enforce correct scoping of type variables.

However, our well-formedness judgements for types, written $\Gamma \vdash \tau^{\kappa} :: \kappa\ \varphi$, additionally tracks the flexibility φ of the type. Tracking flexibility allows us to introduce two restrictions inspired by Garrigue and Rémy's work on ambivalent types:

1. Only equations between *rigid* types can be introduced to the context.

2. In a scoped ambivalent type $[\Psi]\tau$, the type τ under the scope Ψ must be rigid.

These restrictions also simplify our presentation while remaining sufficient to encode OCaml's behaviour. We discuss these restrictions more in Section ??. For convenience, we write $\Gamma \vdash \tau^{\kappa} :: \kappa$ when the type is flexible. The $\varphi_1 \leq \varphi_2$ relation is used to weaken the flexibility of type variables, permitting the following rule to be derivable:

Type equivalence Types have an associated equivalence relation, written $\Gamma \vdash \tau_1 \equiv \tau_2\ [\Psi]$, read: under the context Γ, τ_1 and τ_2 are equivalent in the scope Ψ. We refer to Ψ as the support in the equivalence relation. We often write $\Gamma \vdash \tau_1 \equiv \tau_2$ if the support is empty.

To prove equivalence, we can rely on equalities referenced in Ψ as well as standard rules such as reflexivity, symmetry, transitivity, congruence, and decomposition. Type constructors are injective. Additionally, we introduce three rules associated with scoped ambivalent types.

The first rule permits us to prove equalities between scoped types using the equations in their scopes. The latter two rules are axioms: distributivity and M. Distributivity allows the movement of scopes with a type, which is cruicial for ensuring that eliminators can be applied correctly. Axiom M permits us to remove trivial scopes. They are inspired by the corresponding axioms in modal logic.

AML-F≤-REFL
$$\frac{}{\varphi \leq \varphi}$$

AML-F≤-WK
$$\frac{}{\mathrm{r} \leq \mathrm{f}}$$

AML-TY-VAR
$$\frac{\alpha ::^{\varphi_1} \kappa \in \Gamma \qquad \varphi_1 \leq \varphi_2}{\Gamma \vdash \alpha :: \kappa\ \varphi_2}$$

AML-TY-CON
$$\frac{\forall i, \Gamma \vdash \tau_i :: \mathsf{ty}\ \varphi \qquad \mathsf{arity}(\mathsf{F}) = |\overline{\tau}|}{\Gamma \vdash \overline{\tau}\ \mathsf{F} :: \kappa\ \varphi}$$

AML-TY-BOX
$$\frac{\Gamma \vdash \Psi :: \mathsf{sc} \qquad \Gamma \vdash \tau :: \mathsf{ty}\ \mathrm{r}}{\Gamma \vdash [\Psi]\tau :: \mathsf{ty}}$$

AML-TY-EMPSC
$$\frac{}{\Gamma \vdash \emptyset :: \mathsf{sc}}$$

AML-TY-SNOCSC
$$\frac{\Gamma \vdash \Psi :: \mathsf{sc} \qquad \phi \in \mathsf{dom}(\Gamma)}{\Gamma \vdash \Psi, \phi :: \mathsf{sc}}$$

AML-SCM-TY
$$\frac{\Gamma \vdash \tau :: \mathsf{ty}}{\Gamma \vdash \tau\ \mathsf{scm}}$$

AML-SCM-∀
$$\frac{\Gamma, \alpha ::^{\mathrm{f}} \kappa \vdash \sigma\ \mathsf{scm} \qquad \alpha \# \Gamma}{\Gamma \vdash \forall \alpha :: \kappa.\ \sigma\ \mathsf{scm}}$$

AML-CTX-EMP
$$\frac{}{\cdot\ \mathsf{ctx}}$$

AML-CTX-VAR
$$\frac{\Gamma\ \mathsf{ctx} \qquad \Gamma \vdash \sigma\ \mathsf{scm} \qquad x \# \Gamma}{\Gamma, x : \sigma\ \mathsf{ctx}}$$

AML-CTX-TYVAR
$$\frac{\Gamma\ \mathsf{ctx} \qquad \alpha \# \Gamma \qquad \alpha \# \Gamma}{\Gamma, \alpha ::^{\varphi} \kappa\ \mathsf{ctx}}$$

AML-CTX-EQ
$$\frac{\Gamma\ \mathsf{ctx} \qquad \Gamma \vdash \tau_1, \tau_2 :: \mathsf{ty}\ \mathrm{r} \qquad \phi \# \Gamma}{\Gamma, \phi : \tau_1 = \tau_2\ \mathsf{ctx}}$$

AML-≡REFL
$$\frac{\Gamma \vdash \tau :: \mathsf{ty}}{\Gamma \vdash \tau \equiv \tau\ [\Psi]}$$

AML-≡SYM
$$\frac{\Gamma \vdash \tau_1 \equiv \tau_2\ [\Psi]}{\Gamma \vdash \tau_2 \equiv \tau_1\ [\Psi]}$$

AML-≡TRANS
$$\frac{\Gamma \vdash \tau_1 \equiv \tau_2\ [\Psi] \qquad \Gamma \vdash \tau_2 \equiv \tau_3\ [\Psi]}{\Gamma \vdash \tau_1 \equiv \tau_3\ [\Psi]}$$

AML-≡USE
$$\frac{\phi : \tau_1 = \tau_2 \in \Gamma \qquad \phi \in \Psi}{\Gamma \vdash \tau_1 \equiv \tau_2\ [\Psi]}$$

AML-≡CONJ
$$\frac{\forall i,\ \Gamma \vdash \tau_{1i} \equiv \tau_{2i}\ [\Psi] \qquad \mathsf{arity}(\mathsf{F}) = |\overline{\tau_1}| = |\overline{\tau_2}|}{\Gamma \vdash \overline{\tau_1}\ \mathsf{F} \equiv \overline{\tau_2}\ \mathsf{F}\ \ [\Psi]}$$

AML-≡INJ
$$\frac{\Gamma \vdash \overline{\tau_1}\ \mathsf{F} \equiv \overline{\tau_2}\ \mathsf{F}\ \ [\Psi]}{\Gamma \vdash \tau_{1i} \equiv \tau_{2i}\ [\Psi]}$$

AML-≡BOX
$$\frac{\Gamma \vdash \tau_1 \equiv \tau_2\ [\Psi_1]}{\Gamma \vdash [\Psi_1]\tau_1 \equiv [\Psi_1]\tau_2\ [\Psi_2]}$$

AML-≡DIST
$$\frac{\mathsf{arity}(\mathsf{F}) > 0}{\Gamma \vdash [\Psi_1](\overline{\tau}\ \mathsf{F}) \equiv \overline{[\Psi_1]\tau}\ \mathsf{F}\ \ [\Psi_2]}$$

AML-≡M
$$\frac{\Gamma \vdash \tau :: \mathsf{ty}\ \mathrm{r}}{\Gamma \vdash [\emptyset]\tau \equiv \tau\ [\Psi]}$$

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

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

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

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

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

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

Typing The typing rules are given in Figure ??.

Variables (x) are typed as usual. If a variable has a polymorphic type, the standard ML instantiation rule applies. The instantiation relation $\Gamma \vdash \sigma \leq \sigma'$ is defined as follows:

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

This relation is mostly standard, adapted to account for type equivalence and scope polymorphism. An interesting consequence of our instantiation relation is that the following rule is derivable in AML:

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

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

A type-level equality may be introduced via reflexivity using the unique constructor **Refl** with type scheme $\forall \alpha.\ \alpha = \alpha$. Pattern matching on equalities using **match** $e_1 : \tau_1 = \tau_2$ **with Refl** $\rightarrow e_2$ can eliminate the equality witness e_1 of type $\tau_1 = \tau_2$, adding it as an implicit equality to the context Γ while type checking the body e_2; the witness e_1 must have the structure of $\tau_1 = \tau_2$ but may have additional scopes – we formalize this using annotations, as discussed below. Since the equation is only available in the scope of e_2, it must not be present in the return type σ. We ensure this by allocating a fresh name for the equality $\phi \# \Gamma$ and ensuring ϕ does not occur in σ, this is done so by using the well-formedness judgement $\Gamma \vdash \sigma\ \mathsf{scm}$. If the return type fails to satisfy this condition, then we say the equation (ϕ) has escaped its scope.

Annotations represent an explicit loss of sharing of scopes between types. This loss of sharing of scopes permits us to eliminate ambivalence in our return type, thereby preventing scope escapes. The types τ_1, τ_2 need not be identical to τ but must instead be consistent instances of τ with differing scopes. We express this by defining a scope insertion function $\lceil \tau \rceil$ for rigid types, which produces a context of scope variables $\mathcal{S}$ and a type τ' with scopes inserted at leaves of the type. Scopes at inner nodes can be inferred via the distributivity of scopes.

$$\begin{aligned}
\lceil \alpha \rceil &= \varsigma \rhd [\varsigma]\alpha && \text{fresh } \varsigma \\
\lceil \mathsf{F} \rceil &= \varsigma \rhd [\varsigma]\mathsf{F} && \text{fresh } \varsigma \\
\lceil \overline{\tau}\ \mathsf{F} \rceil &= \mathcal{S}_1, \ldots, \mathcal{S}_n \rhd \overline{\tau'}\ \mathsf{F} && \text{where } \lceil \tau_i \rceil = \mathcal{S}_\rangle \rhd \tau_i'
\end{aligned}$$

We write $\forall \lceil \tau \rceil$ for $\forall \mathcal{S}.\ \tau'$ where $\lceil \tau \rceil = \mathcal{S} \rhd \tau'$. This type scheme describes a set of instances

that are equivalent to τ after erasing their scopes. We say such instances are consistent instances of τ.

Function applications $e_1\ e_2$ are standard and oblivious to scopes. The parameter type τ_1 of the function e_1 must be equal to that of the argument e_2. In particular, in the presence of scopes, if e_1 have the type $[\Psi]\tau$ where τ and $\Gamma \vdash \tau \equiv \tau_1 \to \tau_2\ [\Psi]$, then the argument e_2 must have the type $[\Psi]\tau_1$ and the result of the application has the type $[\Psi]\tau_2$. This behaviour matches the propagation of scopes for application in the following example:

```
let propagate_scope (type a) (w : (a, int -> int) eq) (f : a) =
 match w with Refl (* a = int -> int *) ->
 f 1
```

Naming `a = int -> int` as ϕ, the result has the type $[\phi]\mathbf{int}$. This escapes the scope of the `match` expression, resulting in the following error (with `-principal` enabled): ...

Functions $\lambda x.\ e$ are standard, binding monomorphic types to x in the body of e. The annotated form $\lambda(x : \tau).\ e$ is syntactic sugar for $\lambda x.\ \mathbf{let}\ x = (x : \tau)\ \mathbf{in}\ e$, permitting scope polymorphic types to be bound to x in e. The use of scope polymorphism is cruicial to avoid scope escape errors when using x in a context with a scope. For example, if `f` was not annotated with `(_ : a)` in the above example, then a scope escape would occur even if the return type was annotated. This is because the type of `f` would have to match $[\phi]\mathbf{int}$, thus leaking the equation ϕ.

Let bindings $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$ assign a polymorphic type σ to x in the scope of e_2. Generalizing the type of an expression using flexible variables is standard.

Generalization using rigid variables with an explicit quantifier $\Lambda\alpha.\ e$ requires the rigid generalizable variable α cannot occur in 'dangerous' positions in the type scheme. A dangerous position is one that is under a (non-trivial) scope. We formally define this as follows:

$$\begin{aligned}
\mathsf{dangerous}(\alpha) &= \emptyset \\
\mathsf{dangerous}(\overline{\tau}\ \mathsf{F}) &= \bigcup_i \mathsf{dangerous}(\tau_i) \\
\mathsf{dangerous}([\Psi]\tau) &= \mathsf{fv}(\tau) \\
\mathsf{dangerous}(\forall\alpha :: \kappa.\ \sigma) &= \mathsf{dangerous}(\sigma) \setminus \{\alpha\}
\end{aligned}$$

This is because AML has unrestricted instaniation (and type schemes), thus permitting a generalizable rigid variable under a non-trivial scope would result in a well-formedness issue since scoped ambivalent types $[\Psi]\tau$ are only well-formed if τ is rigid.

Additionally, without this condition, a loss in principality could occur. We demonstrate this by first considering the type of `coerce`, defined below:

```
let coerce = fun (type a b) w x ->
```

```
let x = (x : a) in
match (w : (a, b) eq) with Refl ->
(x : b)
```

Without the restriction on generalizable variables, the most general type that could be inferred for `coerce` is

$$\forall \varsigma_1, \varsigma_2, \varsigma_3, \varsigma_4, \alpha, \beta.\ ([\varsigma_1]\alpha = [\varsigma_2]\beta) \to [\varsigma_3]\alpha \to [\varsigma_4]\beta$$

The issue with this inferred type is that the additional scopes lose some of the sharing that Garrigue and Rémy's system enforces. This could result in some morally ambivalent programs being well-typed in our system. For example:

```
let shoud_not_typecheck (type a) (w : (a, int) eq) (x : a) =
  match w with Refl ->
  let y = if true then x else 0 in
  coerce Refl y
```

We note that `y` has the type $[\phi : \alpha = \mathbf{int}]\alpha$ (or $[\phi : \alpha = \mathbf{int}]\mathbf{int}$), yet we can infer **int** or α for the result of `should_not_typecheck`. This amounts to a loss in principality. Our notion of 'dangerous' variables encodes that in order to generalize α, β, the type system must ensure α, β do not occur under scopes. With this constraint, the principal type for `coerce` is:

$$\forall \alpha, \beta.\ (\alpha = \beta) \to \alpha \to \beta$$

4.2.1 Ambivalent constraints

We briefly explain how to extend our constraint-based approach presented in Section ?? with support for scoped ambivalent types and type annotations.

For type annotations, we must provide support for explicit universal quantification:

$$C ::= \ldots \mid \forall \alpha :: \kappa.\ C$$

The semantics is given by:

$$\textsc{Sat-Forall}\ \frac{\Phi, \alpha ::^{\mathrm{r}} \kappa \vdash C \qquad \alpha \# \Phi}{\Phi \vdash \forall \alpha :: \kappa.\ C}$$

While the universal quantification constraint is sufficient for typing the $\Lambda\alpha.\ e$ construct, to permit *linear complexity* for type checking (and constraint generation) we extend **let**

constraints with universally quantified variables $\overline{\alpha}$, yielding *rigid constraint abstractions.* ...

The constraint generation rule for

4.3 Constraint solving

4.4 Discussion

Our previous work [??] explored a constraint-based approach for the type inference of GADTs using ambivalent types, introducing the novel notion of *ambivalent constraints.* The constraint language used a less restricted form of *rigid implication constraints* and incorporated ambivalent types into the type algebra. In parallel, Martinot, also focused on this area, going further and providing a formal specification for our constraint solver.

Unfortunately, both works are flawed. Our stratification of the model for our constraints into types and ambivalent types (sets of types) didn't correctly represent 'flexization', a process used by Garrigue and Remy, that converts a rigid type variable into a flexible one (thereby subtly allowing one to instantiate this flexible variable with an ambivalent type). As such the following program is well-typed for our inference algorithms but ill-typed in Garrigue and Rémy's specification

Discuss Chore and pointwise inference – their lack of principality for trade off with precision

Discuss Thomas refis's work and attribute the idea of scopes (in the implementation's sense) to him.

4.5 Future work

Integration of omnidirectional type inference for 'deep' GADTs / removing unnecessary annots

Removing the dangerous condition if possible.

Match branch annotation propagation using omnidirectional type inference .

Modal types here can also be used to detect the equational scope escape in staged meta-programming

5 Thesis proposal

- Implicit parameters - Modules and first-class modules, based on Russo's work, using the latest formalisation of OCaml's modules - Structural polymorphism - Rank(1) polymorphism - Constraint-based inference with algebraic subtyping might be nice. Other tricky features such as intersection types / how it works with algebraic subtyping for suspended constraints - Labelled arguments

5.1 Timeline

Bibliography

[1] 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`.

[2] 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`.

[3] 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`.

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

[5] 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`.

[6] 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`.

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

[8] 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`.

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

[10] 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`.

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

[12] 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`.

[13] 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.

[14] 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`.

[15] 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`.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.