Constraint-based type inference for FreezeML

Citation for published version:
Emrich, F, Stolarek, J, Cheney, J & Lindley, S 2022, 'Constraint-based type inference for FreezeML', *Proceedings of the ACM on Programming Languages*, vol. 6, no. ICFP, 111, pp. 570-595.
https://doi.org/10.1145/3547642

Digital Object Identifier (DOI):
10.1145/3547642

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Publisher's PDF, also known as Version of record

Published In:
Proceedings of the ACM on Programming Languages

Constraint-Based Type Inference for FreezeML

FRANK EMRICH, The University of Edinburgh, UK
JAN STOLAREK[*], The University of Edinburgh, UK
JAMES CHENEY[†], The University of Edinburgh, UK
SAM LINDLEY, The University of Edinburgh, UK

FreezeML is a new approach to first-class polymorphic type inference that employs term annotations to control when and how polymorphic types are instantiated and generalised. It conservatively extends Hindley-Milner type inference and was first presented as an extension to Algorithm W. More modern type inference techniques such as HM(X) and OutsideIn(X) employ constraints to support features such as type classes, type families, rows, and other extensions. We take the first step towards modernising FreezeML by presenting a constraint-based type inference algorithm. We introduce a new constraint language, inspired by the Pottier/Rémy presentation of HM(X), in order to allow FreezeML type inference problems to be expressed as constraints. We present a deterministic stack machine for solving FreezeML constraints and prove its termination and correctness.

CCS Concepts: • **Software and its engineering → Polymorphism**; **Functional languages**.

Additional Key Words and Phrases: first-class polymorphism, type inference, impredicative types, constraints

ACM Reference Format:
Frank Emrich, Jan Stolarek, James Cheney, and Sam Lindley. 2022. Constraint-Based Type Inference for FreezeML. *Proc. ACM Program. Lang.* 6, ICFP, Article 111 (August 2022), 26 pages. https://doi.org/10.1145/3547642

1 INTRODUCTION

Hindley-Milner type inference is well-studied, yet extending it to provide full support for polymorphism ("first-class" polymorphism a la System F) remains an active research topic—characterised in one recent paper as "a deep, deep swamp" [Serrano et al. 2018]. A term such as $\lambda f.f\ f$, which would be rejected by Hindley-Milner type inference, may be accepted by a type system permitting first-class polymorphism, by assigning a sufficiently polymorphic type to f, such as $\forall a.a \to a$. However, type inference for System F is undecidable [Wells 1994], meaning that some restrictions must be imposed. Choosing $\forall a.a$ as the type for f also allows the example above to type-check, but no System F type can be given to the function that subsumes both choices. A wide range of solutions has emerged to explore the resulting design space, yielding systems that go beyond System F types, employ elaborate heuristics that determine the system's behaviour, require type annotations for certain terms, or rely on additional syntax, or give up on completeness or principal typing, to name

[*] Also with Lodz University of Technology.
[†] Also with The Alan Turing Institute.

Authors' addresses: Frank Emrich, frank.emrich@ed.ac.uk, The University of Edinburgh, UK; Jan Stolarek, jan.stolarek@ed.ac.uk, The University of Edinburgh, UK; James Cheney, james.cheney@ed.ac.uk, The University of Edinburgh, UK; Sam Lindley, sam.lindley@ed.ac.uk, The University of Edinburgh, UK.

2475-1421/2022/8-ART111
https://doi.org/10.1145/3547642

a few [Garrigue and Rémy 1999; Le Botlan and Rémy 2003; Leijen 2008; Russo and Vytiniotis 2009; Serrano et al. 2020, 2018; Vytiniotis et al. 2006].

Recently, Emrich et al. [2020] proposed a new approach called FreezeML that has several desirable properties: it conservatively extends ML type inference, allows expressing arbitrary System F types and computations, and retains decidable, complete type inference. The key ingredient of FreezeML is the "freezing" operation, an annotation on term-level variables that *blocks* automatic instantiation of any quantifiers in that variable's type. FreezeML also includes let- and lambda-bindings with ascribed types (which are standard in other systems). Unlike other approaches to first-class polymorphism that err on the side of explicitness [Garrigue and Rémy 1999; Russo and Vytiniotis 2009], FreezeML uses just System F types instead of introducing different, incompatible sorts of polymorphic types.

Freezing enables the programmer to control when instantiation happens instead of requiring the type inference algorithm to guess or employ some heuristic that the programmer must then work around. For instance, suppose we have defined functions $\mathsf{single} : \forall a.a \to \mathsf{List}\ a$, which creates a singleton list, and $\mathsf{choose} : \forall a.a \to a \to a$, which returns one of its arguments (for type inference purposes it does not matter which). In FreezeML we can define $f_1\ () = \mathsf{single}\ \mathsf{choose}$ and $f_2\ () = \mathsf{single}\ \lceil \mathsf{choose} \rceil$. In the former (following the usual ML convention) both single and choose are fully instantiated before the body is generalised, hence $f_1 : \forall a.\mathsf{unit} \to \mathsf{List}\ (a \to a \to a)$. In the latter, however, instantiation of choose is frozen, hence $f_2 : \mathsf{unit} \to \mathsf{List}\ (\forall a.a \to a \to a)$.

The original presentation of FreezeML type inference was given as an extension to Algorithm W [Damas and Milner 1982]. Although Algorithm W is well-understood, many modern type inference implementations, notably Haskell, employ *constraint-based type inference* [Pottier 2014; Pottier and Rémy 2005; Vytiniotis et al. 2011] instead in which type inference is split into two stages, mediated by an intermediate logical language of *constraints* [Odersky et al. 1999; Pottier and Rémy 2005]. In the first stage, programs M are translated to constraints C such that C is solvable if and only if M is typable, and the solutions to C are the possible types of M. In the second stage, the constraint C is solved (or shown to be unsatisfiable), without further reference to M. Adopting a constraint-based inference strategy has several potential benefits over the traditional Algorithm W-style, including separating the core logic of type inference from the details of the surface language, leveraging already-known efficient techniques for constraint solver implementation, and supporting extensions such as type classes and families, subtyping, rows, units of measure, GADTs [Odersky et al. 1999; Simonet and Pottier 2007; Vytiniotis et al. 2011], etc.

One influential approach to constraint-based type inference is HM(X) [Odersky et al. 1999; Pottier and Rémy 2005], that is, Hindley-Milner type inference "parameterised over X", where X stands for a constraint domain that can be used in types. For example, if X is a theory of type equality, one obtains standard Hindley-Milner type inference HM(=); if X is a theory of row types one obtains row type inference; if X is a theory of subtyping one gets type inference with subtyping.

In this paper, we take a first step towards such a constraint-parametric system, a version of FreezeML parameterised in a constraint domain X, in the spirit of HM(X). Specifically, we introduce a constraint language for FreezeML, inspired by HM(X), in which type expressions can include arbitrary polymorphism, and which provides suitable constraints to encode type inference for FreezeML programs. (We have not yet explored parameterising the system over the constraint domain X, but even adapting FreezeML to a constraint-based approach turns out to require surmounting significant technical obstacles.) We also provide a deterministic stack machine for solving these constraints (again inspired by the presentation of constraint solving for HM(X) by Pottier and Rémy). Full correctness proofs for both contributions are included in the extended version [Emrich et al. 2022].

Formulating a suitable constraint language for FreezeML and a (provably) correct translation and sound and complete solver involves several subtleties. Handling the freeze operator itself turns out to be straightforward by adding a constraint that checks that the type of a variable exactly matches an expected type. Unification of types needs to account for polymorphism occurring anywhere in a type, and constraints need to be extended with universal quantifiers as well, in order to deal with polymorphism in ascribed types. We also add a "monomorphism constraint" to enforce FreezeML's requirement that certain types are required to be monomorphic. Finally, to deal with FreezeML's approach to the value restriction (in which let-bindings of non-values are allowed but not generalised) we introduce an additional constraint form to handle type inference of non-generalisable expressions. However, the most challenging problem is to design a constraint language and semantics that preserves the necessary invariants to ensure that FreezeML type inference remains sound, complete, and principal: specifically, to ensure that flexible type variables occurring in the inferred types of variables are always monomorphic, which is necessary in FreezeML to avoid the need to "guess polymorphism" when a polymorphic type is instantiated.

Like certain other systems [Leijen 2008; Leroy and Mauny 1993; Vytiniotis et al. 2006], typing derivations in FreezeML require principal types to be assigned to certain subterms. To the best of our knowledge, the inference algorithm shown in this paper is the first one based on constraint solving for such a type system, requiring similar principality conditions in the semantics of the constraint language. Detailed proofs of correctness are provided in the extended version [Emrich et al. 2022].

We characterise these contributions as a first step towards a longer-term goal: parameterising FreezeML type inference over other constraint domains X. This is a natural next step for future work, and would enable experimentation with combining FreezeML-style polymorphism with features found in other modern type systems, such as Haskell's type classes and families (and the numerous libraries that rely on them), higher-kinded types, and GADTs [Vytiniotis et al. 2011], row types as found in Links [Lindley and Cheney 2012], Koka [Leijen 2014] or Rose [Morris and McKinna 2019], units of measure as found in F# [Kennedy 2009] and some Haskell libraries [Gundry 2015]. To summarise, in this paper we:

- present background on FreezeML (Section 2);
- introduce a constraint language inspired by Pottier and Rémy's presentation of HM(X) and give a translation from FreezeML programs to constraints representing type inference problems (Section 3);
- present a stack machine for solving the constraints (again inspired by Pottier and Rémy's) and show that it is correct, deterministic, and terminating (Section 4);
- discuss extensions (Section 5), related and future work (Section 6), and conclude (Section 7).

2 FREEZEML

In this section we summarise the syntax and typing rules of FreezeML. (We omit the dynamic semantics, given by elaboration into System F [Emrich et al. 2020], as it is not relevant to the current paper.)

Lists as Sets. We write $\tilde{X}$ for a (possibly empty) set $\{X_1, \dots, X_n\}$ and $\overline{X}$ for a (possibly empty) sequence $X_1, \dots, X_n$. We overload comma for use as a union / concatenation operator for sets and sequences, writing $\tilde{X}, \tilde{Y}$ for the set $\{X_1, \dots, X_m, Y_1, \dots, Y_n\}$ where $\tilde{X} = \{X_1, \dots, X_m\}$ and $\tilde{Y} = \{Y_1, \dots, Y_n\}$, and writing $\overline{X}, \overline{Y}$ for the sequence $X_1, \dots, X_m, Y_1, \dots, Y_n$ where $\overline{X} = X_1, \dots, X_m$ and $\tilde{Y} = Y_1, \dots, Y_n$. Given $\overline{X}$, we may write $\tilde{X}$ for the set containing the same elements. We sometimes indicate that sets or sequences are required to be disjoint using the # relation, e.g. $\Delta \# \Delta'$ means that Δ and Δ' are disjoint.

Types. The syntax of types, instantiations, and contexts is as follows.

Type Variables	a, b, c
Type Constructors	$D ::= \to \mid \times \mid \mathsf{Int} \mid \ldots$
Types	$A, B ::= a \mid D\,\overline{A} \mid \forall a.A$
Monotypes	$S, T ::= a \mid D\,\overline{S}$
Guarded Types	$G, H ::= a \mid D\,\overline{A}$
Type Instantiation	$\delta ::= \emptyset \mid \delta[a \mapsto A]$
Type Contexts	$\Delta, \Xi ::= \cdot \mid \Delta, a$
Term Contexts	$\Gamma ::= \cdot \mid \Gamma, x : A$

Types are assembled from type variables (a, b, c) and type constructors (D). Type constructors include at least functions ($\to$), products ($\times$), and base types. FreezeML uses System F types (A, B), and the only syntactic form for expressing type polymorphism is $\forall a.A$. A type is either a type variable (a), a data types ($D\,\overline{A}$) with type constructor D and type arguments $\overline{A}$, or a polymorphic type ($\forall a.A$) that binds type variable a in type A. We consider types equal modulo alpha-renaming, but not up to reordering of quantifiers or the addition/removal of superfluous (i.e., unused) quantified variables. For example, the following types are all different: $\forall a.\forall b.a \to b$, $\forall b.\forall a.a \to b$, $\forall a.\forall b.\forall c.a \to b$. Monotypes ($S, T$) disallow any polymorphism. Guarded types (G, H) disallow polymorphism at the top-level. A type instantiation (δ) maps type variables to types. Unlike traditional presentations of ML, we explicitly track type variables in a type context (Δ). By convention we reserve Ξ for flexible type contexts which we will not need until we treat constraints in Section 3. Term contexts (Γ) ascribe types to term variables. Contexts are unordered and duplicates are disallowed. As such, we will frequently take advantage of the fact that a type context Δ is a set of type variables $\tilde{a}$ and use both notations interchangeably. This means that we impose the same disjointness conditions when writing Δ, Δ'.

Typing Judgements. FreezeML typing judgements have the form $\Delta; \Gamma \vdash M : A$, stating that term M has type A in type context Δ and term context Γ. We assume standard well-formedness judgements for types and term contexts: $\Delta \vdash A$ **ok** and $\Delta \vdash \Gamma$ **ok**, which state that only type variables in Δ can appear in A and Γ respectively. Moreover, the term well-formedness judgement $\Delta; \Gamma \vdash M$ **ok** states that all free term variables of M appear in Γ and type annotations are well-formed. This judgement also implements the scoping rules of FreezeML, where certain let bindings bring type variables in scope such that they become available in type annotations [Emrich et al. 2020]. The scoping behaviour interacts with the value restriction adopted by FreezeML, we therefore introduce $\Delta; \Gamma \vdash M$ **ok** formally when discussing let bindings later in this section.

The typing rules are given in Fig. 1. As usual, in these rules we implicitly assume that types and term contexts are well-formed with respect to the type context and that the term is well-formed with respect to the type and term context (i.e., $\Delta; \Gamma \vdash M$ **ok**). In the following running examples, we assume that the function id is in scope and has type $\forall a.a \to a$.

Variables and Instantiation. A frozen variable ($\lceil x \rceil$) can only have the exact type as given by the term environment Γ (rule VarFrozen). This means meaning that the *only* type of $\lceil \mathsf{id} \rceil$ is $\forall a.a \to a$. In contrast, plain variables (x) can be instantiated, as in algorithmic presentations of ML (rule VarPlain). In fact, plain variables are the only terms in FreezeML that eliminate polymorphic types. This means that if we have $\Gamma(x) = \forall \overline{a}.H$, then the possible types of x are all results of instantiating all $\overline{a}$ in H, using arbitrarily polymorphic types. Potential nested quantifiers inside H are not instantiated, however. As a result, for any well-formed B, the type $B \to B$ is a possible type of id, whereas $\forall a.a \to a$ is not.

$$\boxed{\Delta;\Gamma \vdash M : A}$$

$$\textsc{VarFrozen}\ \frac{x : A \in \Gamma}{\Delta;\Gamma \vdash \lceil x \rceil : A} \qquad \textsc{VarPlain}\ \frac{x : \forall\overline{a}.H \in \Gamma \qquad \Delta \vdash \delta : \tilde{a} \Rightarrow_{\star} \cdot}{\Delta;\Gamma \vdash x : \delta(H)} \qquad \textsc{App}\ \frac{\Delta;\Gamma \vdash M : A \to B \qquad \Delta;\Gamma \vdash N : A}{\Delta;\Gamma \vdash M\,N : B}$$

$$\textsc{LamPlain}\ \frac{\Delta;(\Gamma, x : S) \vdash M : B}{\Delta;\Gamma \vdash \lambda x.M : S \to B} \qquad \textsc{LamAnn}\ \frac{\Delta;(\Gamma, x : A) \vdash M : B}{\Delta;\Gamma \vdash \lambda(x : A).M : A \to B}$$

$$\textsc{LetPlain}\ \frac{\begin{array}{c}\overline{a} = \mathsf{ftv}(A') - \Delta \\ (\Delta,\overline{a},M,A') \Updownarrow A \qquad (\Delta,\tilde{a});\Gamma \vdash M : A' \qquad \Delta;(\Gamma, x : A) \vdash N : B \qquad \mathsf{principal}(\Delta,\Gamma,M,\tilde{a},A')\end{array}}{\Delta;\Gamma \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : B}$$

$$\textsc{LetAnn}\ \frac{(\tilde{a},A') = \mathsf{split}(A,M) \qquad (\Delta,\tilde{a});\Gamma \vdash M : A' \qquad \Delta;(\Gamma, x : A) \vdash N : B}{\Delta;\Gamma \vdash \mathbf{let}\ (x : A) = M\ \mathbf{in}\ N : B}$$

Fig. 1. FreezeML Typing Rules.

Formally, the VarPlain typing rule relies on an *instantiation* δ. Each instantiation is parameterised by a *restriction*[1] R which can be either monomorphic ($\bullet$) or polymorphic ($\star$), indicating whether type variables may be substituted with monotypes or arbitrary types. The instantiation judgement $\Delta \vdash \delta : \Delta' \Rightarrow_R \Delta''$ states that instantiation δ instantiates type variables in Δ' with types subject to restriction R using the type context Δ, Δ''. Variables in Δ are considered to be mapped to themselves. This means that if R is $\star$, then for all $a \in \Delta'$, $\delta(a)$ must be a well-formed type in context Δ, Δ'', which may therefore be arbitrarily polymorphic. Otherwise, if R is $\bullet$, then each such $\delta(a)$ must be a monomorphic type. Note that due to all variables $b \in \Delta$ being mapped to themselves, $\delta(b)$ is always a (monomorphic) well-formed type in context Δ, Δ'' for all such b.

Intuitively, the variables in Δ correspond to those appearing in the surrounding context, whereas Δ' corresponds to variables being instantiated and Δ'' contains new type variables appearing in the instantiation. In VarPlain, the Δ'' environment is empty, but in the principal operation discussed below, Δ'' need not be empty. In order for this interpretation to make sense the judgement has an implicit precondition that $\Delta' \# \Delta$ and $\Delta'' \# \Delta$. It is defined as follows.

$$\boxed{\Delta \vdash \delta : \Delta' \Rightarrow_R \Delta''}$$

$$\frac{}{\Delta \vdash \emptyset : \cdot \Rightarrow_R \Delta'} \qquad \frac{\Delta \vdash \delta : \Delta' \Rightarrow_R \Delta'' \qquad \Delta, \Delta'' \vdash_R A\ \mathbf{ok}}{\Delta \vdash \delta[a \mapsto A] : (\Delta', a) \Rightarrow_R \Delta''}$$

We write $\Delta \vdash_R A$ **ok** for the well-formedness judgement for types. It is standard except for the presence of R; if R is $\bullet$ then $\Delta \vdash_\bullet A$ only holds if A is a monotype S. In other words, the judgement enforces the restriction R, where any well-formed type satisfies the restriction $\star$. Therefore, the well-formedness judgement $\Delta \vdash A$ **ok** introduced earlier is a shorthand for $\Delta \vdash_\star A$ **ok**.

[1] Emrich et al. [2020] called these *kinds*, but we prefer to avoid potential confusion with other uses of this overloaded term.

Functions. Function applications ($M\,N$) are standard and oblivious to polymorphism. The parameter type A of the function M must exactly match that of the argument N, where A may be arbitrarily polymorphic. In particular, $\lceil id \rceil$ 3 is ill-typed because $\lceil id \rceil$'s type $\forall a.a \to a$ is not a function type. Conversely, id $\lceil \mathsf{id} \rceil$ has type $\forall b.b \to b$. The first occurrence of id is instantiated, by picking the type $\forall b.b \to b$ of $\lceil \mathsf{id} \rceil$ for the quantified type variable. This showcases the impredicative nature of FreezeML, with alpha-renaming performed for the sake of clarity.

Plain (i.e., unannotated) lambda abstractions ($\lambda x.M$) restrict the domain to be monomorphic. This is a simple way to keep type inference tractable, in line with other systems [Leijen 2008; Serrano et al. 2018]. Annotated lambda abstractions ($\lambda(x : A).M$) allow the domain to be polymorphic, at the cost of a type annotation. As a result, the example term $\lambda f.f\ f$ given in the introduction is rejected in FreezeML, unless f is annotated with an appropriate type. Writing $\lambda(f : \forall a.a \to a).f\ f$ yields a function of type $(\forall a.a \to a) \to (B \to B)$ for any well-formed B. The return types of both forms of lambda abstractions may be arbitrarily polymorphic: both $\lambda(f : \forall a.a \to a).f\ \lceil f \rceil$ and $\lambda x.\lceil \mathsf{id} \rceil$ yield functions with polymorphic return types.

Principality. The LetPlain rule has a *principality* side condition that requires that the type inferred for x is a principal one. Terms cannot arbitrarily be generalised in FreezeML while retaining typability. The term id has type $A \to A$ for any type A, and in particular $a \to a$ for any type variable a. However, it does not have type $\forall a.a \to a$. As in System F, there is no direct relationship between the types $\forall a.a \to a$ and $A \to A$ in FreezeML; generalisation only happens when a let-bound generalised value is encountered, and instantiation only happens if triggered by a plain variable occurrence.

The fact that FreezeML typing judgements carry type contexts specifying all in-scope type variables makes it possible to characterise principal types without universally quantifying additional type variables. Principal types are always given in their context $\Delta;\Gamma$ and may use free type variables not present in Δ. For example, the principal types of term id in the context $\Delta;\Gamma$ are exactly the types $b \to b$ for any $b \notin \Delta$. This presentation is syntax-directed, in contrast to declarative presentations of ML that allow generalisation at any point, and would typically refer to the *type scheme* $\forall a.a \to a$ (not to be confused with the corresponding System F type) as the principal type of id.

We formalise the notion of principal type using the predicate $\mathsf{principal}(\Delta, \Gamma, M, \Delta', A')$. It first asserts that A' is a possible type of M in the context $\Delta, \Delta'; \Gamma$, where Δ' includes all type variables of A' not in Δ. A minimality condition must also be satisfied: any other possible type of M can be obtained by instantiating the variables in Δ', possibly using some new type variables in context Δ'', but (crucially) *not* instantiating any of those in Δ. Formally, a *principal type* of M in the context of Δ and Γ is any pair (Δ', A') such that $\mathsf{principal}(\Delta, \Gamma, M, \Delta', A')$ holds. This is unique up to safe renaming of the variables in Δ' (that is, avoiding already-known variables in Δ) and the occurrence of superfluous variables in Δ' (i.e. variables not actually occuring in A').[2] Given a particular A' we can recover Δ', so we say *the* principal type to refer to a particular A' when Δ' is clear from context.

$$\begin{array}{l}
\mathsf{principal}(\Delta, \Gamma, M, \Delta', A') = \\
\quad \Delta, \Delta'; \Gamma \vdash M : A' \text{ and} \\
\quad (\text{for all } \Delta'', A'' \mid \text{if } \Delta, \Delta''; \Gamma \vdash M : A'' \\
\qquad\qquad\qquad\quad \text{then there exists } \delta \text{ such that} \\
\qquad\qquad\qquad\quad\ \Delta \vdash \delta : \Delta' \Rightarrow_{\star} \Delta'' \text{ and } \delta(A') = A'')
\end{array}$$

[2] We could impose minimality of Δ' and Δ'' in the definition of principal, but any superfluous variables in either context simply have no effect.

Remark 1 (Abuse of notation)**.** *Notice that the definition of* principal *refers to typing derivations in the "if" part of the condition. The reader may be concerned about whether the typing judgement is well-defined given that it appears in a negative position in the definition of* principal*. As Emrich et al. [2020] explain we can see that the definition is nevertheless well founded by indexing by untyped terms or the height of derivation trees. Likewise, proofs involving typing derivations are typically by induction on M rather than by rule induction. Although it is a slight abuse of notation, we prefer to present the typing rules using inference rule notation for ease of comparison with other systems. Formally however the rules in Figure 1 are implications that happen to hold of the typing relation, not an inductive definition of it. We refer to the extended version of Emrich et al. [2020] for the precise definition.*

Plain Let Bindings. Following ML, FreezeML adopts a syntactic value restriction [Wright 1995], distinguishing two subcategories of terms.

$$\begin{array}{lll}\text{Values} & \mathsf{Val} \ni V, W ::= & \lceil x \rceil \mid x \mid \lambda x.M \mid \lambda(x : A).M \mid \textbf{let}\ x = V\ \textbf{in}\ W \mid \textbf{let}\ (x : A) = V\ \textbf{in}\ W \\ \text{Guarded Values} & \mathsf{GVal} \ni U ::= & x \mid \lambda x.M \mid \lambda(x : A).M \mid \textbf{let}\ x = V\ \textbf{in}\ U \mid \textbf{let}\ (x : A) = V\ \textbf{in}\ U\end{array}$$

Values disallow applications. Guarded values disallow frozen variables, and thus must have guarded type.[3]

Plain let bindings ($\textbf{let}\ x = M\ \textbf{in}\ N$) generalise – subject to the value restriction – the principal type A' of M and ascribe it to x. Here, the predicate principal is used to determine the type A', using fresh variables $\overline{a}$. Note that the free type variable operator ftv returns a sequence rather than a set when applied to a type, returning variables in the order of their appearance. This reflects the fact that the order of quantifiers matters in FreezeML.

If M is a guarded value, the type A of x is then $\forall \overline{a}.A'$, performing the actual generalisation step. This is achieved using the $\Updownarrow$ auxiliary judgement that enforces the value restriction, which we return to shortly.

Generalising the principal type A' rather than an arbitrary type of M is necessary to ensure the existence of principal types in the overall system [Emrich et al. 2020]. First, recall that a principal type of an expression M with respect to Δ and Γ is formally a *pair* (Δ', A') such that $\mathsf{principal}(\Delta, \Gamma, M, \Delta', A')$ hold. Generalisation then quantifies the variables in Δ' yielding a type $\forall a_1, \ldots, a_n.A$ where $a_1, \ldots, a_n$ are in the order in which the variables first appear in A'. Now consider the term

$$\textbf{let}\ f = \lambda x.x\ \textbf{in}\ \lceil f \rceil$$

which is an example of FreezeML's "explicit generalisation" operation $\$V \equiv \textbf{let}\ f = V\ \textbf{in}\ \lceil f \rceil$ and allows capturing the generalised principal type of a value. The principal type of $\lambda x.x$ is $a \rightarrow a$ (provided $a \notin \Delta$), and by generalising this we obtain $\forall a.a \rightarrow a$ as the type of f, which then becomes the type of the overall let term due to f being frozen in its body. Note that the type $\forall a.a \rightarrow a$ is *not* the principal type of $\lambda x.x$ (formally: no pair $(\Delta'', \forall a.a \rightarrow a)$ is a principal type). If the typing rule permitted assigning other, non-principal, types to $\lambda x.x$, such as $\mathsf{Int} \rightarrow \mathsf{Int}$, then generalisation would have no effect. This would make $\mathsf{Int} \rightarrow \mathsf{Int}$ another possible type of the overall let term (as freezing a variable with a guarded or monomorphic type has no effect). However, this would mean that the overall let term has no principal type. The two types $\mathsf{Int} \rightarrow \mathsf{Int}$ and $\forall a.a \rightarrow a$ don't have a shared more general type in FreezeML, as discussed earlier.

As mentioned before, the auxiliary judgement $(\Delta, \overline{a}, M, A') \Updownarrow A$ enforces the value restriction. Given $\Delta, \tilde{a} \vdash M : A'$, the judgement determines A to be $\forall \overline{a}.A'$ if M is a guarded value. Otherwise, A is obtained from A' by instantiating all of $\tilde{a}$ with *monotypes*.

[3] The only guarded value with a top-level polymorphic type is a plain variable x of type $\forall a_1, \ldots, a_n.a_i$. This special case is handled gracefully by FreezeML.

$$\boxed{(\Delta, \overline{a}, M, A') \Updownarrow A}$$

$$\frac{M \in \mathsf{GVal}}{(\Delta, \overline{a}, M, A') \Updownarrow \forall \overline{a}.A'} \qquad \frac{\Delta' = \tilde{a} \qquad \Delta \vdash \delta : \Delta' \Rightarrow_{\bullet} \cdot \qquad M \notin \mathsf{GVal}}{(\Delta, \overline{a}, M, A') \Updownarrow \delta(A')}$$

As is well-known, type inference for System F is undecidable, even with nontrivial restrictions [Pfenning 1993; Wells 1994]. The condition to instantiate monomorphically is one of several design choices in FreezeML's to keep type inference decidable and tractable. Along with the monomorphic restriction on the arguments to plain lambda abstractions, FreezeML ensures that polymorphism can only ever appear in the term context if it was written explicitly by a programmer in a type annotation or inferred as a principal type of a plain let binding.

Annotated Let Bindings. Annotated let bindings ($\mathbf{let}\ (x : A) = M\ \mathbf{in}\ N$) also generalise, subject to the value restriction, but ascribe the type A to x. The splitting operation $\mathsf{split}(A, M)$ enforces the value restriction for annotated let terms. It decomposes A into a collection of top-level quantifiers and another type. The first component of the returned pair is maximal if M is a guarded value and empty otherwise due to the value restriction.

$$\mathsf{split}(\forall \overline{a}.H, M) = \begin{cases} (\tilde{a}, H) & \text{if } M \in \mathsf{GVal} \\ (\cdot, \forall \overline{a}.H) & \text{otherwise} \end{cases}$$

It is also important to note that in the generalising case (i.e. when the let-bound expression is a guarded value U), the top-level quantifiers in type annotations are in scope and can be used in U (e.g. in other type annotations). This is reflected in the split operation which returns these variables in its first argument. In contrast, in the non-generalising case where M is not a guarded value, these variables are not in scope in M. Since M's type is not being generalised, the only way it can end up with a polymorphic type is by referencing (frozen) variables with polymorphic types.

Note that this scoping behaviour also needs to be reflected in the term well-formedness judgement $\Delta; \Gamma \vdash M\ \mathbf{ok}$ mentioned earlier. To this end, the well-formedness rule for annotated let bindings also uses the split operation, as shown in Figure 2. The judgement $\Delta; \Gamma \vdash M\ \mathbf{ok}$ only requires the presence of a binding for all free term variables, but ignores the associated types. As a result, the rules for unannotated lambda functions and let bindings add arbitrary types A to the the term context.

$$\boxed{\Delta; \Gamma \vdash M\ \mathbf{ok}}$$

$$\frac{x \in \Gamma}{\Delta; \Gamma \vdash \lceil x \rceil\ \mathbf{ok}} \qquad \frac{x \in \Gamma}{\Delta; \Gamma \vdash x\ \mathbf{ok}} \qquad \frac{\Delta; (\Gamma, x : A) \vdash M\ \mathbf{ok}}{\Delta; \Gamma \vdash \lambda x.M\ \mathbf{ok}} \qquad \frac{\begin{array}{c}\Delta \vdash A\ \mathbf{ok} \\ \Delta; (\Gamma, x : A) \vdash M\ \mathbf{ok}\end{array}}{\Delta; \Gamma \vdash \lambda (x : A).M\ \mathbf{ok}}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash M\ \mathbf{ok} \\ \Delta; \Gamma \vdash N\ \mathbf{ok}\end{array}}{\Delta; \Gamma \vdash M\, N\ \mathbf{ok}} \qquad \frac{\Delta; \Gamma \vdash M\ \mathbf{ok} \qquad \Delta; (\Gamma, x : A) \vdash N\ \mathbf{ok}}{\Delta; \Gamma \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N\ \mathbf{ok}}$$

$$\frac{\Delta \vdash A\ \mathbf{ok} \qquad (\Delta', A') = \mathsf{split}(A, M) \qquad (\Delta, \Delta'); \Gamma \vdash M\ \mathbf{ok} \qquad \Delta; (\Gamma, x : A) \vdash N\ \mathbf{ok}}{\Delta; \Gamma \vdash \mathbf{let}\ (x : A) = M\ \mathbf{in}\ N\ \mathbf{ok}}$$

Fig. 2. Well-formedness of terms.

3 CONSTRAINT LANGUAGE

In this section, we present the constraint language and a function for generating typing constraints from terms. Following Pottier and Rémy [2005], our constraint language uses both term variables and type variables. Following Emrich et al. [2020], we distinguish rigid and flexible type variables. The former arise in the object language from universal quantification. The latter are used to represent unknown types.

The syntax and satisfiability judgement for constraints is given in Figure 3. The judgement $\Delta; \Xi; \Gamma; \delta \vdash C$ states that in rigid type context Δ, flexible type context Ξ, term context Γ, using instantiation δ, the constraint C is satisfied. Note that rigid and flexible type contexts follow the same grammar, but we use the convention that Δ is used for rigid variables, whereas Ξ contains flexible ones. Therefore, using both environments in the judgement allows us to distinguish the flexible variables in scope from those that are rigid. In the judgement $\Delta; \Xi; \Gamma; \delta \vdash C$ we implicitly assume that the term environment Γ is well-formed and contains no flexible variables ($\Delta \vdash \Gamma$ **ok**) and that type instantiations close over the flexible type variables ($\Delta \vdash \delta : \Xi \Rightarrow_\star \cdot$).

To support composition of constraints we start with the always true constraint (true) and conjunction ($C_1 \wedge C_2$). The equality constraint $A \sim B$ asserts that A and B are equivalent. The frozen constraint $\lceil x : A \rceil$ asserts that x has type A. The instance constraint $x \preceq A$ asserts that top-level quantifiers of x's type can be instantiated to yield A. The universal constraint $\forall a.C$ binds rigid type variable a in C. The existential constraint $\exists a.C$ binds flexible type variable a in C. Monomorphism constraints $\mathsf{mono}(a)$ assert that the flexible variable a must only be instantiated with monotypes. The definition constraint $\mathbf{def}\ (x : A)\ \mathbf{in}\ C$ binds term variable x in C. (It also includes a side-constraint which we will return to shortly.) The polymorphic let constraint $\mathbf{let}_\star\ x = \sqcap a.C_1\ \mathbf{in}\ C_2$ and monomorphic let constraint $\mathbf{let}_\bullet\ x = \sqcap a.C_1\ \mathbf{in}\ C_2$ are used to bind x in C_2, subject to the restrictions imposed on a in C_1. The two forms differ in how the type of x is obtained from solving C_1 for a: either by generalisation ($\star$) or monomorphic instantiation ($\bullet$). These constraints are somewhat involved, so we defer a full explanation until we present the constraint-generation function.

We consider constraints equivalent modulo alpha-renaming of all binders, of both type and term variables.

3.1 Constraint Generation

We now introduce the function $[\![M : A]\!]$, which translates a term M and type A to a constraint. The only free type variables in the resulting constraint are those appearing in A and type annotations in M. Assuming that M is well-formed under Δ and Γ ($\Delta; \Gamma \vdash M$ **ok**) and that A is well-formed under Δ, Ξ, the constraint $[\![M : A]\!]$ is well-formed under Δ, Ξ and Γ ($\Delta; \Xi; \Gamma \vdash [\![M : A]\!]$ **ok**). The latter judgement is given in Figure 4. Note that this judgement ignores the types in Γ and uses it to track bound term variables, just like the well-formedness judgement on terms introduced in Section 2.

If Ξ is empty (i.e., A contains no flexible variables) then this constraint is satisfiable in context $\Delta; \Gamma$ if and only if M has type A in context $\Delta; \Gamma$. However, if A does contain flexible variables, then the models of $[\![M : A]\!]$ are exactly those that instantiate A to valid types of M. We formalise these properties in Section 3.4. Concretely, we perform type inference for M by choosing A to be a single flexible variable.

The function $[\![-]\!]$ is defined in Figure 5.

Frozen variables and plain variables generate the corresponding atomic constraints. An application generates an existential constraint that binds a fresh flexible type variable for the argument type. A plain lambda abstraction generates a constraint that binds fresh flexible type variables for argument and return types and uses a definition constraint to bind the argument in the constraints

$$
\begin{array}{rl}
C ::= & \mathsf{true} \mid C \wedge C \mid A \sim B \mid \lceil x : A \rceil \mid x \preceq A \mid \forall a.C \mid \exists a.C \mid \mathsf{mono}(a) \\
\mid & \mathbf{def}\ (x : A)\ \mathbf{in}\ C \mid \mathbf{let}_\star\ x = \sqcap a.C\ \mathbf{in}\ C \mid \mathbf{let}_\bullet\ x = \sqcap a.C\ \mathbf{in}\ C
\end{array}
$$

$$
\textsc{Sem-True}\quad \frac{}{\Delta;\Xi;\Gamma;\delta \vdash \mathsf{true}}
\qquad
\textsc{Sem-And}\quad \frac{\Delta;\Xi;\Gamma;\delta \vdash C_1 \qquad \Delta;\Xi;\Gamma;\delta \vdash C_2}{\Delta;\Xi;\Gamma;\delta \vdash C_1 \wedge C_2}
$$

$$
\textsc{Sem-Equiv}\quad \frac{\begin{array}{c}(\Delta,\Xi) \vdash_\star A\ \mathbf{ok} \qquad (\Delta,\Xi) \vdash_\star B\ \mathbf{ok} \\ \delta(A) = \delta(B)\end{array}}{\Delta;\Xi;\Gamma;\delta \vdash A \sim B}
$$

$$
\textsc{Sem-Freeze}\quad \frac{\Gamma(x) = \delta(A)}{\Delta;\Xi;\Gamma;\delta \vdash \lceil x : A \rceil}
\qquad
\textsc{Sem-Instance}\quad \frac{\begin{array}{c}\Delta' = \tilde{a} \qquad \Delta \vdash \delta' : \Delta' \Rightarrow_\star \cdot \\ \Gamma(x) = \forall \overline{a}.H \qquad \delta'(H) = \delta(A)\end{array}}{\Delta;\Xi;\Gamma;\delta \vdash x \preceq A}
$$

$$
\textsc{Sem-Forall}\quad \frac{(\Delta,a);\Xi;\Gamma;\delta \vdash C}{\Delta;\Xi;\Gamma;\delta \vdash \forall a.C}
\qquad
\textsc{Sem-Exists}\quad \frac{\Delta;(\Xi,a);\Gamma;\delta[a \mapsto A] \vdash C}{\Delta;\Xi;\Gamma;\delta \vdash \exists a.C}
$$

$$
\textsc{Sem-Mono}\quad \frac{\Delta \vdash_\bullet \delta(a)\ \mathbf{ok}}{\Delta;\Xi;\Gamma;\delta \vdash \mathsf{mono}(a)}
\qquad
\textsc{Sem-Def}\quad \frac{\begin{array}{c}\text{for all } a \in \mathsf{ftv}(A) - \Delta \ \mid\ \Delta;\Xi;\Gamma;\delta \vdash \mathsf{mono}(a) \\ \Delta;\Xi;(\Gamma, x : \delta A);\delta \vdash C\end{array}}{\Delta;\Xi;\Gamma;\delta \vdash \mathbf{def}\ (x : A)\ \mathbf{in}\ C}
$$

$$
\textsc{Sem-LetPoly}\quad \frac{\begin{array}{c}\mathsf{mostgen}(\Delta,(\Xi,a),\Gamma,C_1,\Delta_{\mathrm{m}},\delta_{\mathrm{m}}) \\ \Delta_{\mathrm{o}} = \mathsf{ftv}(\delta_{\mathrm{m}}(\Xi)) - \Delta \qquad \overline{b} = \mathsf{ftv}(\delta_{\mathrm{m}}(a)) - \Delta, \Delta_{\mathrm{o}} \\ \Delta \vdash \delta' : \Delta_{\mathrm{o}} \Rightarrow_\bullet \cdot \qquad A = \delta'(\delta_{\mathrm{m}}(a)) \\ (\Delta,\tilde{b});(\Xi,a);\Gamma;\delta[a \mapsto A] \vdash C_1 \qquad \Delta;\Xi;(\Gamma, x : \forall \overline{b}.A);\delta \vdash C_2\end{array}}{\Delta;\Xi;\Gamma;\delta \vdash \mathbf{let}_\star\ x = \sqcap a.C_1\ \mathbf{in}\ C_2}
$$

$$
\textsc{Sem-LetMono}\quad \frac{\begin{array}{c}\mathsf{mostgen}(\Delta,(\Xi,a),\Gamma,C_1,\Delta_{\mathrm{m}},\delta_{\mathrm{m}}) \\ \Delta \vdash \delta' : \Delta_{\mathrm{m}} \Rightarrow_\bullet \cdot \qquad A = \delta'(\delta_{\mathrm{m}}(a)) \\ \Delta;(\Xi,a);\Gamma;\delta[a \mapsto A] \vdash C_1 \qquad \Delta;\Xi;(\Gamma, x : A);\delta \vdash C_2\end{array}}{\Delta;\Xi;\Gamma;\delta \vdash \mathbf{let}_\bullet\ x = \sqcap a.C_1\ \mathbf{in}\ C_2}
$$

Fig. 3. Satisfiability judgement for constraints.

generated for the body of the lambda abstraction. An annotated lambda abstraction generates a similar constraint to a plain lambda abstraction, but the argument type is fixed by the type annotation. The remaining four cases of $[\![-]\!]$ account for the four different combinations arising from the two choices between plain or annotated and between guarded value or not. An annotated let binding **let** $(x : B) = M$ **in** N generates a conjunction of constraints: one for M and the other for N. Following the definition of split in the LetAnn rule in Figure 1, if M is a guarded value U then its type can be generalised to obtain B as witnessed by the universal constraints. Notice in particular that the quantified type variables introduced in the annotation are in scope in U in the sub-constraint $\forall \overline{a}.[\![U : H]\!]$. Otherwise the types must match on the nose without any generalisation, and in this case the quantified variables are not in scope in M.

$$\frac{}{\Delta;\Xi;\Gamma \vdash \mathsf{true}\ \mathbf{ok}} \qquad \frac{a \in (\Delta,\Xi)}{\Delta;\Xi;\Gamma \vdash \mathsf{mono}(a)\ \mathbf{ok}} \qquad \frac{\Delta;\Xi;\Gamma \vdash C_1\ \mathbf{ok} \qquad \Delta;\Xi;\Gamma \vdash C_2\ \mathbf{ok}}{\Delta;\Xi;\Gamma \vdash C_1 \wedge C_2\ \mathbf{ok}}$$

$$\frac{\Delta;(\Xi,a);\Gamma \vdash C\ \mathbf{ok}}{\Delta;\Xi;\Gamma \vdash \exists a.C\ \mathbf{ok}} \qquad \frac{(\Delta,a);\Xi;\Gamma \vdash C\ \mathbf{ok}}{\Delta;\Xi;\Gamma \vdash \forall a.C\ \mathbf{ok}} \qquad \frac{(\Delta,\Xi) \vdash A\ \mathbf{ok} \qquad (\Delta,\Xi) \vdash B\ \mathbf{ok}}{\Delta;\Xi;\Gamma \vdash A \sim B\ \mathbf{ok}}$$

$$\frac{x \in \Gamma \qquad (\Delta,\Xi) \vdash A\ \mathbf{ok}}{\Delta;\Xi;\Gamma \vdash x \preceq A\ \mathbf{ok}} \qquad \frac{x \in \Gamma \qquad (\Delta,\Xi) \vdash A\ \mathbf{ok}}{\Delta;\Xi;\Gamma \vdash \lceil x : A \rceil\ \mathbf{ok}}$$

$$\frac{(\Delta,\Xi) \vdash A\ \mathbf{ok} \qquad \Delta;\Xi;(\Gamma, x : A) \vdash C\ \mathbf{ok}}{\Delta;\Xi;\Gamma \vdash \mathbf{def}\ (x : A)\ \mathbf{in}\ C\ \mathbf{ok}} \qquad \frac{\begin{array}{c}\Delta;(\Xi,a);\Gamma \vdash C_1\ \mathbf{ok} \\ \Delta;\Xi;(\Gamma, x : A) \vdash C_2\ \mathbf{ok}\end{array}}{\Delta;\Xi;\Gamma \vdash \mathbf{let}_R\ x = \sqcap a.C_1\ \mathbf{in}\ C_2\ \mathbf{ok}}$$

Fig. 4. Well-formedness of constraints.

$$\begin{array}{rcll}
[\![\lceil x \rceil : A]\!] & = & \lceil x : A \rceil & \\
[\![x : A]\!] & = & x \preceq A & \\
[\![M\,N : A]\!] & = & \exists a_1.([\![M : a_1 \to A]\!] \wedge [\![N : a_1]\!]) & \\
[\![\lambda x.M : A]\!] & = & \exists a_1, a_2.(a_1 \to a_2 \sim A \wedge \mathbf{def}\ (x : a_1)\ \mathbf{in}\ [\![M : a_2]\!]) & \\
[\![\lambda (x : B).M : A]\!] & = & \exists a_1.B \to a_1 \sim A \wedge \mathbf{def}\ (x : B)\ \mathbf{in}\ [\![N : a_1]\!] & \\
[\![\mathbf{let}\ (x : \forall \overline{a}.H) = U\ \mathbf{in}\ N : A]\!] & = & (\forall \overline{a}.[\![U : H]\!]) \ \wedge\ \mathbf{def}\ (x : \forall \overline{a}.H)\ \mathbf{in}\ [\![N : A]\!] & \\
[\![\mathbf{let}\ (x : B) = M\ \mathbf{in}\ N : A]\!] & = & [\![M : B]\!] \ \wedge\ \mathbf{def}\ (x : B)\ \mathbf{in}\ [\![N : A]\!] & (\text{if } M \notin \mathsf{GVal}) \\
[\![\mathbf{let}\ x = U\ \mathbf{in}\ N : A]\!] & = & \mathbf{let}_\star\ x = \sqcap a.[\![U : a]\!]\ \mathbf{in}\ [\![N : A]\!] & \\
[\![\mathbf{let}\ x = M\ \mathbf{in}\ N : A]\!] & = & \mathbf{let}_\bullet\ x = \sqcap a.[\![M : a]\!]\ \mathbf{in}\ [\![N : A]\!] & (\text{if } M \notin \mathsf{GVal})
\end{array}$$

Fig. 5. Translation from terms to constraints.

3.2 Def Constraints

The side condition in the Sem-Def rule ensures that the argument type can only be instantiated with a monomorphic type. In general, the side condition preserves the invariant that no undetermined (or "guessed") polymorphism exists in the term context Γ. This is crucial to ensure the existence of most general solutions for our constraint language. Consider the constraint $\mathbf{def}\ (x : a)\ \mathbf{in}\ x \preceq b \wedge c \sim (a \to b)$ with free flexible variables a, b, c. Without the extra condition on def constraints, different solutions could for instance include $c \mapsto (\mathsf{Int} \to \mathsf{Int})$ or $c \mapsto ((\forall a.a) \to \mathsf{Int})$ for c. However, there is no more general solution subsuming both. Note that the monomorphism condition on def constraints does not impose the type annotation to be monomorphic itself, it only imposes conditions on free flexible variables appearing within it. Consequently, the constraint $\mathbf{def}\ (x : (\forall a.a) \to b)\ \mathbf{in}\ \mathsf{true}$ is satisfiable as long as the flexible variable b is instantiated monomorphically. We cannot avoid the monomorphism condition imposed on def constraints simply by using mono constraints. The constraint $\mathsf{mono}(b) \wedge \mathbf{def}\ f(x : (\forall a.a) \to b)\ \mathbf{in}\ \mathsf{true}$ would be equivalent to the previous one in terms of its solutions, even if we dropped the monomorphism condition built into def constraints. However, this system would exhibit the same lack of most general solutions for def constraints discussed earlier, showing that the monomorphism condition needs to be imposed on def constraints directly.

3.3 Let Constraints

Plain let bindings are translated to let constraints. A plain let binding of a guarded value **let** $x = U$ **in** N generates a polymorphic let constraint. In general, such a polymorphic let constraint $\mathbf{let}_\star\ x = \sqcap a.C_1$ **in** C_2 binds the flexible variable a in C_1, much like an existential constraint. The type assigned to x in C_2 is then obtained by generalising type variables appearing in the solution for a.

We make several observations motivating the overall semantics of let constraints.

Need to Generalise Principal Solution. We first observe that let *constraints* require a principality condition similar to the one imposed on plain let *terms.* Consider the constraint C defined as **let** $x = \sqcap a.\exists b.a \sim (b \to b)$ **in** $\lceil x : c \rceil$, appearing in a rigid context Δ. It has a single free type variable c and we refer to its first subconstraint (i.e., $\exists b.a \sim (b \to b)$) as C_1 in the following. Allowing arbitrary solutions for a in C_1 to be generalised to yield the type for x would lead to the following pathological situation.

For any well-formed type A, $[a \mapsto (A \to A)]$ is a model of C_1. As usual, we must not generalise any type variables already bound in the surrounding scope, namely those variables in Δ. However, we may generalise fresh variables appearing in A. This means that if we choose $[a \mapsto (\mathsf{Int} \to \mathsf{Int})]$ there is nothing to generalise and we have $x : (\mathsf{Int} \to \mathsf{Int})$ in C_2 , whereas $[a \mapsto (b' \to b')]$ for some fresh b' does allow us to generalise, meaning that we have $x : (\forall b'.b' \to b')$ in C_2. Any solution of the overall constraint must use the type of x for c. This leads to a problem very similar to the one discussed for let terms in Section 2: the two solutions $[c \mapsto (\mathsf{Int} \to \mathsf{Int})]$ and $[c \mapsto (\forall b'.b' \to b')]$ of C would have no shared most general solution in our system.

We avoid this problem by demanding that only the most general solution for a in C_1 must be generalised to yield the type for x in C_2. In our example, this means choosing $[a \mapsto (b' \to b')]$, where b' is fresh, which means that in our example only $[c \mapsto (\forall b'.b' \to b')]$ is a valid solution of the overall let constraint.

The rule Sem-LetPoly in Figure 3 enforces this using the premise $\mathsf{mostgen}(\Delta, (\Xi, a), \Gamma, C_1, \Delta_\mathrm{m}, \delta_\mathrm{m})$, which asserts that δ_m is the most general model of C_1 in the context $\Delta; \Xi; \Gamma$. Here, Δ_m contains fresh variables that are used in place of flexible type variables for which no further substitution/solution is currently known. Note that this premise (and subsequently, δ_m) is independent from the ambient instantiation δ; the latter does not appear as an argument. The predicate mostgen is defined as follows, stating that δ_m is a model of C_1 and every other one can be obtained by refining δ_m by composition.

$$\begin{array}{l}\mathsf{mostgen}(\Delta, \Xi, \Gamma, C, \Delta_\mathrm{m}, \delta_\mathrm{m}) = \\ \quad (\Delta, \Delta_\mathrm{m}); \Xi; \Gamma; \delta_\mathrm{m} \vdash C \ \text{ and} \\ \quad (\text{for all } \Delta'', \delta'' \mid \text{if } (\Delta, \Delta''); \Xi; \Gamma; \delta'' \vdash C \\ \qquad\qquad\qquad \text{then there exists } \delta' \text{ such that} \\ \qquad\qquad\qquad \Delta \vdash \delta' : \Delta_\mathrm{m} \Rightarrow_\star \Delta'' \text{ and } \delta' \circ \delta_\mathrm{m} = \delta'')\end{array}$$

The rule Sem-LetPoly then defines two subsets[4] of Δ_m: The variables in Δ_o are those appearing in the range of δ_m restricted to Ξ (i.e., not considering the mapping for a in δ_m). This means that the variables in Δ_o are related to the *outer* context, namely by being part of the instantiations of the variables Ξ in the surrounding scope.

The rule then determines the variables $\overline{b}$ to be generalised as the flexible ones appearing freely in $\delta_\mathrm{m}(a)$ (i.e., the most general solution for a) and disregarding the variables from Δ_o, as the latter variables are related to the outer scope Ξ.

[4] In general, Δ_m may contain useless variables not appearing in the codomain of δ_m. Otherwise, if all variables in Δ_m appear in the range of δ_m, then Δ_m and $\overline{b}$ denote a partitioning of Δ_m.

Safe Interaction with Outer Scope. We have discussed that the rule Sem-LetPoly forces solutions for constraints of the form $\mathbf{let}_\star\ x = \sqcap a.C_1\ \mathbf{in}\ C_2$ to use the most general solution for a in C_1 – using fresh rigid variables Δ_m – and quantifying over variables $\overline{b} \subseteq \Delta_\mathrm{m}$ to obtain the type for x in C_2.

We now show how flexible variables from the outer scope that appear in C_1 may influence the type of x and how we prevent this from introducing undetermined polymorphism in the term context. Consider the constraint $\exists a.\mathbf{let}_\star\ x = \sqcap b.a \sim b\ \mathbf{in}\ C_2$ appearing in rigid context Δ. The semantics of $\exists$ constraints (cf. Sem-Exists in Figure 3) necessitates choosing a type B for a such that $\Delta \vdash B\ \mathbf{ok}$. The first subconstraint of the let constraint then equates a and b, making any kind of generalisation impossible when determining the type of x (i.e., the type of x is just B without further quantification). However, this means that the choice of B influences the polymorphism of x, meaning that the constraint above may introduce undetermined polymorphism in the term context if arbitrarily polymorphic types were permitted for a. Thus we must restrict the possible choices for B. The rule Sem-LetPoly does so by imposing a relationship between the most general solution $\delta_\mathrm{m}(a)$ for a and the type A actually chosen as the instantiation of a. In our example above, each most general solution δ_m of C_1 has the form $[a \mapsto c, b \mapsto c]$, where $c \in \Delta_\mathrm{m}$. Therefore, we have $\Delta_\mathrm{o} = c$ and $\overline{b}$ is empty. The rule Sem-LetPoly then imposes that the actual type A for a results from monomorphically instantiating all non-generalisable variables in $\delta_\mathrm{m}(a)$ (namely, Δ_o). In the example above, this means that a (and therefore also b) must be instantiated with a monotype. Observe that in general, $\delta'(\delta_\mathrm{m}(a))$ may not be a feasible choice for A for any well-formed monomorphic instantiation δ'. Consider the constraint:

$$\exists a.a \sim (\mathsf{Int} \to \mathsf{Int}) \wedge (\mathbf{let}_\star\ x = \sqcap b.\exists c.a \sim b \wedge a \sim (c \to c)\ \mathbf{in}\ C_2)$$

Here, $\delta'(\delta_\mathrm{m}(a))$ may yield any type of the form $S \to S$ (recall that S denotes monotypes). However, the premise $(\Delta, \tilde{b}); (\Xi, a); \Gamma; \delta[a \mapsto A] \vdash C_1$ of Sem-LetPoly forces A to be compatible with any prior choices made by the ambient instantiation δ. In our examples, this ensures that $A = (\mathsf{Int} \to \mathsf{Int})$.

Monomorphic Let Constraints. To accommodate the value restriction, the function $[\![-]\!]$ translates a plain let binding of a term M that is not a guarded value, $\mathbf{let}\ x = M\ \mathbf{in}\ N$, to a monomorphic let constraint of the form $\mathbf{let}_\bullet\ x = \sqcap a.C_1\ \mathbf{in}\ C_2$. The only difference between a polymorphic let constraint and a monomorphic one is that all variables that would be generalised by the former are instantiated monomorphically by the latter.

The rule Sem-LetMono in Figure 3 achieves this by instantiating all of Δ_m monomorphically to obtain A from $\delta_\mathrm{m}(a)$. An equivalent, yet slightly more verbose version of Sem-LetMono highlighting the symmetry between Sem-LetPoly and Sem-LetMono could define Δ_o and $\overline{b}$ just like the former rule, and then impose $\Delta \vdash \delta' : (\Delta_\mathrm{o}, \overline{b}) \Rightarrow_\bullet \cdot$. Observe that the variables in $\Delta_\mathrm{m} - (\Delta_\mathrm{o}, \overline{b})$ are irrelevant in Sem-LetPoly.

3.4 Metatheory

We can now formalise the relationship between terms and the constraints obtained from them. Firstly, if M has type A, then $[\![M : a]\!]$ is satisfiable by a substitution that maps a to A.

Theorem 1 (Constraint generation is sound with respect to the typing judgement). *Let* $\Delta; \Gamma \vdash M : A$ *and* $a \# \Delta$. *Then* $\Delta; a; \Gamma; [a \mapsto A] \vdash [\![M : a]\!]$ *holds.*

Secondly, if a constraint $[\![M : a]\!]$ is satisfied using an instantiation δ, then $\delta(a)$ is a valid type for M.

Theorem 2 (Constraint generation is complete with respect to the typing judgement). *If* $\Delta; \Gamma \vdash M\ \mathbf{ok}$ *and* $\Delta; a; \Gamma; \delta \vdash [\![M : a]\!]$, *then* $\Delta; \Gamma \vdash M : \delta(a)$.

Both properties are proved by structural induction on M; proof details are provided in the extended version [Emrich et al. 2022].

4 CONSTRAINT SOLVING

We present a stack machine for solving constraints in our language, similar to the HM(X) solver by Pottier and Rémy [2005]. Our machine is defined in terms of a transition relation on states of the form (F, Θ, θ, C), consisting of a *stack* F, a *restriction context* Θ, a *type substitution* θ, and an *in-progress constraint* C, each of which we elaborate on below.

Stacks. In a state (F, Θ, θ, C), C denotes the constraint to be solved next.

The stack F denotes the context in which C appears, containing bindings for type variables (rigid and flexible) and term variables that may appear in C. Further, the stack indicates how to continue after C has been solved. Our stack machine operates on closed states, meaning that the stack contains bindings for all free variables of C.

Formally, stacks are built from stack frames as follows.

$$\begin{array}{ll} \text{Frames} & f ::= \square \wedge C \mid \forall\, a \mid \exists\, a \mid \mathbf{let}_R\ x = \sqcap a.\square \text{ in } C \mid \mathbf{def}\ (x : A) \\ \text{Stacks} & F ::= \cdot \mid F :: f \end{array}$$

The different forms of stack frames directly correspond to those constraints with at least one sub-constraint. The overall stack can then be seen as a constraint with a hole in which C is plugged. We use holes $\square$ in frames for constraints with two sub-constraints and store the second sub-constraint to which we must return after solving the first one.

Restriction Contexts and Type Substitutions. The components Θ and θ of a state (F, Θ, θ, C) encode the *unification context*. Their syntax is defined as follows.

$$\begin{array}{ll} \text{Restriction Contexts} & \Theta ::= \cdot \mid \Theta, a : R \\ \text{Type Substitutions} & \theta ::= \emptyset \mid \theta[a \mapsto A] \\ \text{States} & s ::= (F, \Theta, \theta, C) \end{array}$$

The restriction context Θ contains exactly the flexible variables bound by the stack F and stores the restriction imposed on each such variable. Again, restrictions R determine which types a flexible variable may be unified or instantiated with: monomorphic only ($\bullet$) or arbitrary polymorphic types ($\star$).

Type substitutions θ are similar to type instantiations δ. However, they apply only to flexible variables, their codomain may contain flexible variables, and must respect the restriction imposed on each individual variable in the domain. Note that this is in contrast to instantiations, where $\Delta \vdash \delta : \Delta' \Rightarrow_R \Delta''$ fixes a single restriction R for all variables in the domain of δ.

To this end, we formalise what it means for a type A to obey a restriction R using the judgement $\Delta; \Theta \vdash_R A$ **ok**, shown in Figure 6. Rigid variables are monomorphic. Flexible variables have their restriction determined by the restriction context. The restriction of a data type is determined inductively. A universally quantified type is polymorphic. Every monomorphic type is also a polymorphic type. Observe that the well-formedness judgement $\Delta \vdash_R A$ **ok** used in Section 3 can now be considered as a shorthand for $\Delta; \cdot \vdash_R A$ **ok**.

We can now formally state what it means for a substitution θ to be well-formed, mapping flexible variables in Θ' to well-formed types over variables from Δ, Θ via the judgement $\Delta \vdash \theta : \Theta' \Rightarrow \Theta$, which is also shown in Figure 6. As for substitutions, we additionally require that $\Theta \# \Delta$ and $\Theta' \# \Delta$ (however, Θ and Θ' need not be disjoint). In summary, this means that in any solver state, the substitution θ contains the current knowledge about unification variables, respecting the restrictions imposed by Θ.

$$\boxed{\Delta;\Theta \vdash_R A\ \mathbf{ok}}$$

$$\frac{a \in \Delta}{\Delta;\Theta \vdash_\bullet a} \qquad \frac{a : R \in \Theta}{\Delta;\Theta \vdash_R a} \qquad \frac{\begin{array}{c}\mathsf{arity}(D) = n \\ \Delta;\Theta \vdash_R A_1\ \mathbf{ok} \\ \cdots \\ \Delta;\Theta \vdash_R A_n\ \mathbf{ok}\end{array}}{\Delta;\Theta \vdash_R D\,\overline{A}\ \mathbf{ok}} \qquad \frac{(\Delta, a);\Theta \vdash_R A\ \mathbf{ok}}{\Delta;\Theta \vdash_\star \forall a.A\ \mathbf{ok}} \qquad \frac{\Delta;\Theta \vdash_\bullet A\ \mathbf{ok}}{\Delta;\Theta \vdash_\star A\ \mathbf{ok}}$$

$$\boxed{\Delta \vdash \theta : \Theta' \Rightarrow \Theta}$$

$$\frac{}{\Delta \vdash \emptyset : \cdot \Rightarrow \Theta} \qquad \frac{\Delta \vdash \theta : \Theta' \Rightarrow \Theta \qquad \Delta;\Theta \vdash_R A\ \mathbf{ok}}{\Delta \vdash \theta[a \mapsto A] : (\Theta', a : R) \Rightarrow \Theta}$$

Fig. 6. Well-formedness of types and substitutions.

We write $\mathsf{bv}(F)$ and $\mathsf{btv}(F)$ for the term variables and type variables (flexible or rigid) bound by F, respectively. Moreover, we write $\mathsf{rc}(F)$, $\mathsf{fc}(F)$, and $\mathsf{tc}(F)$ for the rigid context, flexible context, and term context synthesised from a stack F, respectively. The latter operators consider $\forall$ frames ($\mathsf{rc}(F)$), let and $\exists$ frames ($\mathsf{fc}(F)$), and def frames ($\mathsf{tc}(F)$), as shown in Figure 7.

$$\mathsf{rc}(F) = \begin{cases} \cdot & \text{if } F = \cdot \\ \mathsf{rc}(F'), a & \text{if } F = F' :: \forall a \\ \mathsf{rc}(F') & \text{otherwise } (F = F' :: \_) \end{cases} \qquad \mathsf{fc}(F) = \begin{cases} \cdot & \text{if } F = \cdot \\ \mathsf{fc}(F'), a & \text{if } F = F' :: \exists a \text{ or} \\ & \quad F = F' :: \mathbf{let}_R\ x = \sqcap a.\square\ \mathbf{in}\ C_2 \\ \mathsf{fc}(F') & \text{otherwise } (F = F' :: \_) \end{cases}$$

$$\mathsf{tc}(F) = \begin{cases} \cdot & \text{if } F = \cdot \\ \mathsf{tc}(F'), (x : A) & \text{if } F = F' :: \mathbf{def}\ (x : A) \\ \mathsf{tc}(F') & \text{otherwise } (F = F' :: \_) \end{cases}$$

Fig. 7. Extracting components from stacks.

In order for a state (F, Θ, θ, C) to be well-formed ($\vdash (F, \Theta, \theta, C)\ \mathbf{ok}$), we require that $\mathsf{rc}(F) \vdash \theta : \Theta \Rightarrow \Theta$, that θ is idempotent, that C is well-formed ($\mathsf{rc}(F);\mathsf{fc}(F);\mathsf{tc}(F) \vdash C\ \mathbf{ok}$), and that F is well-formed with respect to Θ ($\Theta \vdash F\ \mathbf{ok}$). The latter judgement is defined in Figure 8. In addition to basic well-formedness conditions on the involved types and constraints, the judgement $\Theta \vdash F\ \mathbf{ok}$ imposes the following invariants: all type and term variables bound by F are pairwise disjoint and all free type variables appearing in annotations on def constraints are monomorphic. Moreover, Θ must contain exactly the flexible variables bound by F.

Remark 2 (Idempotent substitutions). *Our requirement that θ be idempotent (i.e. $\theta \circ \theta = \theta$) concretely means that each binding $a \mapsto A$ in θ either maps a to itself (an "undetermined variable") or to a type A whose flexible variables are all undetermined. This has some helpful consequences, for example in the definition of* partition*, discussed later (Remark 3).*

To check the well-formedness of constraints embedded in stack frames, the corresponding rules of $\Theta \vdash F\ \mathbf{ok}$ in Figure 8 synthesise term contexts from the stack under consideration. As with earlier well-formedness judgements, the judgement $\Theta \vdash F\ \mathbf{ok}$ checks that all term variables are in scope, but ignores the associated types.

$\boxed{\Theta \vdash F\ \mathbf{ok}}$

$$\frac{}{\cdot \vdash \cdot\ \mathbf{ok}} \qquad \frac{\mathrm{rc}(F); \Theta; \mathrm{tc}(F) \vdash C\ \mathbf{ok} \qquad \Theta \vdash F\ \mathbf{ok}}{\Theta \vdash F :: \square \wedge C\ \mathbf{ok}}$$

$$\frac{\Theta \vdash F\ \mathbf{ok} \qquad a \notin \mathrm{btv}(F)}{\Theta \vdash F :: \forall\ a\ \mathbf{ok}} \qquad \frac{\Theta \vdash F\ \mathbf{ok} \qquad a \notin \mathrm{btv}(F)}{(\Theta, a : R) \vdash F :: \exists\ a\ \mathbf{ok}}$$

$$\frac{\begin{array}{c}\text{for all } a \in \mathrm{ftv}(A) - \mathrm{rc}(F) \mid (a : \bullet) \in \Theta \\ x \notin \mathrm{bv}(F) \qquad \Theta \vdash F\ \mathbf{ok}\end{array}}{\Theta \vdash F :: \mathbf{def}\ (x : A)\ \mathbf{ok}} \qquad \frac{\begin{array}{c}\mathrm{rc}(F); \Theta; (\mathrm{tc}(F), x : A) \vdash C\ \mathbf{ok} \\ x \notin \mathrm{bv}(F) \qquad \Theta \vdash F\ \mathbf{ok}\end{array}}{(\Theta, a : R) \vdash F :: \mathbf{let}_{R'}\ x = \sqcap a.\square \text{ in } C\ \mathbf{ok}}$$

Fig. 8. Stack well-formedness.

$$\begin{array}{rcll}
(F, \Theta, \theta, A \sim B) & \longrightarrow & (F, \Theta', \theta' \circ \theta, \mathsf{true}) \text{ where} (\Theta', \theta') = \mathcal{U}(\mathrm{rc}(F), \Theta, \theta A, \theta B) & \text{(S-Eq)} \\
(F, \Theta, \theta, \lceil x : A \rceil) & \longrightarrow & (F, \Theta, \theta, \mathrm{tc}(F)(x) \sim A) & \text{(S-Freeze)} \\
(F, \Theta, \theta, x \preceq A) & \longrightarrow & (F, \Theta, \theta, \exists \overline{a}.H \sim A) \text{ where } \forall \overline{a}.H = \mathrm{tc}(F)(x) \quad \tilde{a} \# \mathrm{btv}(F) & \text{(S-Inst)} \\
(F, \Theta, \theta, \mathsf{mono}(a)) & \longrightarrow & (F, \Theta', \theta, \mathsf{true}) & \text{(S-Mono)} \\
\multicolumn{4}{l}{\quad \text{where } \overline{b} = \mathrm{ftv}(\theta(a)) - \mathrm{rc}(F) \quad \Theta' = (\Theta - \tilde{b}) \cup \widetilde{b : \bullet} \quad \Theta' \vdash_\bullet \theta(a)\ \mathbf{ok}} \\
(F, \Theta, \theta, C_1 \wedge C_2) & \longrightarrow & (F :: \square \wedge C_2, \Theta, \theta, C_1) & \text{(S-ConjPush)} \\
(F :: \square \wedge C_2, \Theta, \theta, \mathsf{true}) & \longrightarrow & (F, \Theta, \theta, C_2) & \text{(S-ConjPop)} \\
(F, \Theta, \theta, \exists a. C) & \longrightarrow & (F :: \exists a, (\Theta, a : \star), \theta[a \mapsto a], C) & \text{(S-ExistsPush)} \\
(F :: f :: \exists \overline{a}, \Theta, \theta, \mathsf{true}) & \longrightarrow & (F :: \exists \tilde{c} :: f, \Theta', \theta\restriction_{\Theta'}, \mathsf{true}) & \text{(S-ExistsLower)} \\
\multicolumn{4}{l}{\quad \text{where } f \text{ is neither a } \mathbf{let} \text{ or } \exists \text{ frame} \quad \tilde{b};\ \tilde{c} = \mathrm{partition}(\tilde{a}, \theta, \Theta) \quad \Theta' = \Theta - \tilde{b} \quad |\tilde{a}| > 0} \\
(F, \Theta, \theta, \forall a.C) & \longrightarrow & (F :: \forall\ a, \Theta, \theta, C) & \text{(S-ForallPush)} \\
(F :: \forall\ a, \Theta, \theta, \mathsf{true}) & \longrightarrow & (F, \Theta, \theta, \mathsf{true}) \text{ where } a \notin \mathrm{ftv}(\theta(\Theta)) & \text{(S-ForallPop)} \\
(F, \Theta, \theta, \mathbf{def}\ (x : A) \textbf{ in } C) & \longrightarrow & (F :: \mathbf{def}\ (x : A), \Theta', \theta, C) & \text{(S-DefPush)} \\
\multicolumn{4}{l}{\quad \text{where } \overline{b} = \mathrm{ftv}(\theta(A)) - \mathrm{rc}(F) \quad \Theta' = (\Theta - \tilde{b}) \cup \widetilde{b : \bullet} \quad \text{for all } a \in \mathrm{ftv}(A) \mid \Theta' \vdash_\bullet \theta(a)\ \mathbf{ok}} \\
(F :: \mathbf{def}\ (x : A), \Theta, \theta, \mathsf{true}) & \longrightarrow & (F, \Theta, \theta, \mathsf{true}) & \text{(S-DefPop)} \\
(F, \Theta, \theta, \mathbf{let}_R\ x = \sqcap b.C_1 \textbf{ in } C_2) & \longrightarrow & (F :: \mathbf{let}_R\ x = \sqcap b.\square \textbf{ in } C_2, (\Theta, b : \star), \theta[b \mapsto b], C_1) & \text{(S-LetPush)} \\
(F :: \mathbf{let}_\star\ x = \sqcap b.\square \textbf{ in } C :: \exists \overline{a}, \Theta, \theta, \mathsf{true}) & \longrightarrow & (F :: \exists \widetilde{a''}, \Theta', \theta\restriction_{\Theta'}, \mathbf{def}\ (x : B) \textbf{ in } C) & \text{(S-LetPolyPop)} \\
\multicolumn{4}{l}{\quad \text{where } \widetilde{a'};\ \widetilde{a''} = \mathrm{partition}((\tilde{a}, b), \theta, \Theta) \quad A = \theta(b) \quad \overline{c} = \mathrm{ftv}(A) \cap \widetilde{a'} \quad \Theta' = \Theta - \widetilde{a'} \quad B = \forall \overline{c}.A} \\
(F :: \mathbf{let}_\bullet\ x = \sqcap b.\square \textbf{ in } C :: \exists \overline{a}, \Theta, \theta, \mathsf{true}) & \longrightarrow & (F :: \exists (\tilde{c}, \widetilde{a''}), \Theta', \theta\restriction_{\Theta'}, \mathbf{def}\ (x : A) \textbf{ in } C) & \text{(S-LetMonoPop)} \\
\multicolumn{4}{l}{\quad \text{where } \widetilde{a'};\ \widetilde{a''} = \mathrm{partition}((\tilde{a}, b), \theta, \Theta) \quad A = \theta(b) \quad \overline{c} = \mathrm{ftv}(A) \cap \widetilde{a'} \quad \Theta' = \Theta - (\widetilde{a'} - \tilde{c})}
\end{array}$$

Fig. 9. Constraint solving rules.

4.1 Stack Machine Rules

We now introduce the rules of the constraint solver itself (Figure 9). These rules are deterministic in the sense that at most one rule applies at any point. Moreover, after each step the resulting state

```
$\mathcal{U}(\Delta, \Theta, a, a) =$
  return $(\Theta, \theta_{\mathrm{id}})$

$\mathcal{U}(\Delta, \Theta, D\,\overline{A}, D\,\overline{B}) =$
  let $(\Theta_1, \theta_1) = (\Theta, \theta_{\mathrm{id}})$
  let $n = \mathsf{arity}(D)$
  for $i \in 1 \ldots n$
    let $(\Theta_{i+1}, \theta_{i+1}) =$
      let $(\Theta', \theta') = \mathcal{U}(\Delta, \Theta_i, \theta_i(A_i), \theta_i(B_i))$
      return $(\Theta', \theta' \circ \theta_i)$
  return $(\Theta_{n+1}, \theta_{n+1})$

$\mathcal{U}(\Delta, (\Theta, a : R), a, A) =$
$\mathcal{U}(\Delta, (\Theta, a : R), A, a) =$
  let $\Theta_1 = \mathsf{demote}(R, \Theta, \mathsf{ftv}(A) - \Delta)$
  assert $\Delta, \Theta_1 \vdash_R A$ ok
  return $(\Theta_1, \theta_{\mathrm{id}}[a \mapsto A])$

$\mathcal{U}(\Delta, \Theta, \forall a.A, \forall b.B) =$
  assume fresh $c$
  let $(\Theta_1, \theta') = \mathcal{U}((\Delta, c), \Theta, A[c/a], B[c/b])$
  assert $c \notin \mathsf{ftv}(\theta')$
  return $(\Theta_1, \theta')$
```

$$\begin{aligned} \mathsf{demote}(\star, \Theta, \Delta) &= \Theta \\ \mathsf{demote}(\bullet, \cdot, \Delta) &= \cdot \\ \mathsf{demote}(\bullet, (\Theta, a : R), \Delta) &= \mathsf{demote}(\bullet, \Theta, \Delta), a : \bullet \quad (a \in \Delta) \\ \mathsf{demote}(\bullet, (\Theta, a : R), \Delta) &= \mathsf{demote}(\bullet, \Theta, \Delta), a : R \quad (a \notin \Delta) \end{aligned}$$

Fig. 10. Unification algorithm.

is unique up to the names of binders added to the stack and the order of adjacent existential frames in the frame (e.g., a step may yield either $F :: \exists a :: \exists b :: \ldots$ or $F :: \exists b :: \exists a :: \ldots$). A constraint is satisfiable in a given context if the machine reaches a state of the form $(\forall\Delta :: \exists\Xi, \Theta, \theta, \mathrm{true})$ from an initial configuration built from C and the context. From such a final configuration we can also read off a most general solution for the constraint. If the constraint is unsatisfiable, the machine gets stuck before reaching such a final state. We formalise the properties of the solver in Section 4.2.

Unification. The rule S-Eq in Figure 9 handles equality constraints of the form $A \sim B$. We apply θ to both types as this may refine the types prior to invoking the unification procedure $\mathcal{U}$. It remains unchanged as compared to the original type inference system for FreezeML based on Algorithm W [Emrich et al. 2020]. The unification algorithm is largely standard, supporting unification of polymorphic types without reordering of quantifiers or the removal/addition of unneeded quantifiers, as per FreezeML's notion of type equality. It returns updated versions of the restriction context and substitution, named Θ' and θ'. The algorithm is sound, complete, and yields most general unifiers [Emrich et al. 2020, Theorem 4 and 5].

One notable feature is the unification algorithm's treatment of restrictions. The restriction context Θ' returned by the algorithm contains the same flexible variables as the original Θ, but some variables therein may have been demoted from a polymorphic restriction to a monomorphic one. Unifying a flexible, monomorphic variable a with a type A only succeeds if making all free flexible variables in A monomorphic makes A itself monomorphic. Therefore, assuming $a : \bullet, b : \star \in \Theta$, unifying a with $b \to b$ yields $(b : \bullet) \in \Theta'$, whereas unification of a with $\forall c.c \to c$) fails.

The unification algorithm is shown in Figure 10. On each invocation, the first applicable clause is used; θ_{id} denotes the identity substitution on Θ.

Basic Constraints. The rules for constraints $\lceil x : A \rceil$ and $x \preceq A$ yield corresponding equality constraints. For instantiation constraints, the solver instantiates all top-level quantifiers $\overline{a}$ of x's type by existentially quantifying them. Note that the rule imposes picking variables $\overline{a}$ that are fresh with respect to the bound type variables (rigid and flexible) of F. In both rules, $\mathrm{tc}(F)$ denotes the term context synthesised from all **def** constraints in F.

A monomorphism constraint $\mathsf{mono}(a)$ is handled by demoting all flexible variables in $\theta(a)$. The step fails if doing so does not make $\theta(a)$ a monomorphic type. Note that demoting the involved restrictions means that later unification steps cannot make a polymorphic — even if, say, $\theta(a) = a$ holds at the time of applying S-Mono, recording $(a : \bullet)$ in Θ' ensures that it stays monomorphic.

The rules S-ConjPush and S-ConjPop handle conjunctions. When encountering $C_1 \wedge C_2$, the first rule pushes a corresponding frame on the stack. Once C_1 is solved and the state's in-progress constraint becomes true, the latter rule pops this frame from the stack and continues solving C_2.

Binding of Type Variables. When encountering $\exists a.C$ or $\forall a.C$, a corresponding frame is added by the rules S-ExistsPush and S-ForallPush, respectively.

In general, once the in-progress constraint in a state becomes true and the topmost stack frame binds a flexible variable a, the binding frame is either moved downwards in the stack, generalised when handling **let** constraints, or dropped if no variable further down in the stack depends on a. Note that lowering binders of flexible variables in the stack this way effectively increases the syntactic scope of the bound variable as it moves outwards in the constraint representing the stack. The rule S-ExistsLower implements part of this lowering mechanism; it acts on stacks of the form $F :: f :: \exists\overline{a}$, a shorthand for $F :: f :: \exists a_0 :: \cdots :: \exists a_n$. The rule requires that f is neither a let frame (whose rules treat adjacent existentials directly) or another existential frame (to make the rule deterministic by making $\overline{a}$ exhaustive). It uses the helper function partition, which returns a tuple and is defined as follows.

$$\mathsf{partition}(\Xi, \theta, \Theta) = \Xi'; \Xi'' \text{ where } \Xi', \Xi'' = \Xi \text{ and for all } a \in \Xi : a \in \Xi'' \text{ iff } a \in \mathsf{ftv}(\theta\restriction_{(\mathsf{ftv}(\Theta)-\Xi)})$$

It partitions Ξ into two sets Ξ' and Ξ'' such that the latter contains exactly those variables appearing in the range of θ restricted to the flexible variables bound further down in the stack (i.e., θ restricted to $\mathsf{ftv}(\Theta) - \Xi$). This formalises the notion that no variable further down in the stack depends on a variable in Ξ'. Therefore, the bindings for Ξ' can simply be removed altogether; the bindings for Ξ'' must be kept and are lowered within the stack.

Remark 3 (Idempotence and partition)**.** *If we did not require substitutions to be idempotent, the existing definitions would break in subtle ways (or need to be made more complicated). For example,. let $a, b, c \in \Theta$ and $\Xi = \{a, b\}$ and $\theta = [a \mapsto a, b \mapsto a, c \mapsto b]$. Note that this θ is not idempotent since $\theta(\theta(c))) = a \neq b = \theta(c)$. The definition of* $\mathsf{partition}(\Xi, \theta, \Theta)$ *would then yield $\Xi' = \{a\}$ and $\Xi'' = \{b\}$, even though c depends on both variables.*

When popping a frame $\forall a$ from the stack, the rule S-ForallPop checks that a does not escape its scope by still being present in the range of θ. Note that together with the lowering of existentials mentioned before, removing the unneeded variables $\tilde{b}$ in S-ExistsLower is not simply an optimisation, but necessary for completeness. Consider the state $(F :: \forall a :: \exists b, \Theta, \theta, \mathsf{true})$ where $\theta(b) = a$. If this variable was lowered by S-ExistsLower – yielding a new state $(F :: \exists b :: \forall a, \Theta, \theta, \mathsf{true})$ – instead of being removed, this would cause S-ForallPop to erroneously detect an escaping quantifier.

Binding of Term Variables. Similarly to the other *Push rules, S-DefPush moves a constraint **def** $(x : A)$ **in** C to the stack and makes C the next in-progress constraint. However, it also forces all flexible variables found in $\theta(A)$ to be monomorphic and checks that doing so does not make the substitution ill-formed. This ill-formedness would arise if the substitution maps one of the type variables to be monomorphised to a polymorphic type. The monomorphisation is crucial in order to maintain the invariant that the term context does not contain unknown polymorphism in the form of unrestricted (i.e., polymorphic) unification variables. Note that the checks performed by S-DefPush are equivalent to adding $\bigwedge_{a \in \mathsf{ftv}(A)-\mathsf{rc}(F)} \mathsf{mono}(a)$ as a conjunct to C, but doing so may create an ill-formed intermediate state before failing when solving one of the mono constraints. The rule S-DefPop is the counterpart of S-DefPush and simply pops the **def** frame.

S-LetPush handles constraints $\mathbf{let}_R\ x = \sqcap b.C_1\ \mathbf{in}\ C_2$ by adding a stack frame and bringing b into scope while solving C_1. Once C_1 has been solved, the rules S-LetPolyPop and S-LetMonoPop handle the different semantics of $\mathbf{let}_{\bullet}$ and $\mathbf{let}_{\star}$ regarding how they determine the type of x. We first consider the former rule. Note that the rule is applicable with zero or more existential frames on top of the let frame, binding $\tilde{a}$, followed by the actual let frame. These existential frames are either the result of existential constraints at the top-level of the original constraint C_1 (the first subconstraint of the **let** constraint under consideration), or were lowered while solving C_1.

Similarly to S-ExistsLower, the variables $\tilde{a}$ and b are partitioned by the partition function into $\widetilde{a'}$ and $\widetilde{a''}$. Note that we include b here because $\mathbf{let}_R\ x = \sqcap b.C_1\ \mathbf{in}\ C_2$ binds b existentially in C_1. By definition of partition, we again have that no unification variable bound below the **let** frame depends on any of the variables in $\widetilde{a'}$ (as indicated by a not appearing in the image of θ restricted to the variables bound in the lower frames). Similarly to S-ExistsLower, the variables $\widetilde{a''}$ must be preserved and are lowered in the stack. Note that $\widetilde{a''}$ may or may not contain b.

The type A for x is then determined by generalising $\theta(b)$. The variables $\bar{c}$ to be generalised are obtained from taking those free type variables of $\theta(b)$ that also appear in $\widetilde{a'}$. Recall that ftv applied to a type yields an ordered sequence, and we assume that the ordering is preserved under intersection.

By definition, $\widetilde{a'}$ contains those variables from $\tilde{a}, b$ that do not appear in the codomain of θ restricted to the variables from lower stack frames (i.e. no such variable from a lower stack frame directly depends on a variable in $\widetilde{a'}$). Nevertheless, there may be variables $a \in \widetilde{a'}$ such that $\theta(a) = B$ where B contains (or is) a variable from a lower frame (i.e. from $\Theta - \tilde{a}, b$), meaning that a must not be generalised. However, due to the idempotency of θ, we have that $\theta(a) = a$ for all $a \in \theta(b)$. In other words, intersecting $\widetilde{a'}$ with the free type variables of $\theta(b)$ evokes that $\bar{c}$ only contains "undetermined" variables mapped to themselves that are not referenced by lower stack frames, either.

Note that rewriting the let frame to a def constraint also evokes that solving the latter monomorphises any flexible variables in A that were not generalised. This reflects the monomorphic instantiation imposed in the semantics of $\mathbf{let}_{\star}$ constraints (cf. δ' in Sem-PolyPop in Figure 3).

The only difference between S-LetPolyPop and S-LetMonoPop is that while the latter rule also determines the variables $\tilde{c}$, it does not generalise them. This means that the resulting type A assigned to x contains the unification variables $\tilde{c}$ freely. Therefore, these variables must kept in scope and are existentially quantified further down in stack after the rule is applied.

4.2 Metatheory

Our goal is to state a preservation property along the lines that stepping from state s_0 to s_1 implies that some representation of s_0 as a constraint is equivalent to s_1's constraint representation. To this end, we first define how to represent the unification context, comprising Θ and θ, as a constraint. Given Θ and θ, we define:

$$\begin{aligned}
\mathfrak{U}(\Theta) &= \textstyle\bigwedge_{(a:\bullet)\in\Theta}\ \mathsf{mono}(a) \\
\mathfrak{U}(\theta) &= \textstyle\bigwedge_{a\in\mathsf{ftv}(\Theta)}\ a \sim \theta(a) \\
\mathfrak{U}(\Theta,\theta) &= \mathfrak{U}(\Theta) \wedge \mathfrak{U}(\theta)
\end{aligned}$$

Using $\mathfrak{U}$, we may now represent a state (F, Θ, θ, C) as $F[C \wedge \mathfrak{U}(\Theta, \theta)]$, where the $F[-]$ operator plugs a constraint into the stack's innermost hole:

$$\begin{array}{rcl}
\cdot[C] & = & C \\
(F :: \square \wedge C_2)[C_1] & = & F[C_1 \wedge C_2] \\
(F :: \forall\, a)[C] & = & F[\forall a.C] \\
(F :: \exists\, a)[C] & = & F[\exists a.C] \\
(F :: \mathbf{let}_R\, x = \sqcap a.\square \text{ in } C_2)[C_1] & = & F[\mathbf{let}_R\, x = \sqcap a.C_1 \text{ } \mathbf{in} \text{ } C_2] \\
(F :: \mathbf{def}\ (x : A))[C] & = & F[\mathbf{def}\ (x : A) \text{ } \mathbf{in} \text{ } C]
\end{array}$$

Note that if the state is closed (i.e., F binds all variables free in C) the resulting constraint $F[C]$ is closed, too. In order to reason about constraints that are satisfied by non-empty instantiations, we assume that there are some rigid and flexible contexts Δ and Ξ quantified by the bottom-most stack frames that remain unchanged by the step. Therefore, we consider the satisfiability of constraints before and after the step by an instantiation δ with $\Delta \vdash \delta : \Xi \Rightarrow_\star \cdot$.

Theorem 3 (Preservation). *If* $(\forall\Delta :: \exists\Xi :: F_0, \Theta_0, \theta_0, C_0)$ **ok** *and*

$$(\forall\Delta :: \exists\Xi :: F_0, \Theta_0, \theta_0, C_0) \longrightarrow (\forall\Delta :: \exists\Xi :: F_1, \Theta_1, \theta_1, C_1)$$

then

$$\Delta; \Xi; \cdot; \delta \vdash F_0[C_0 \wedge \mathfrak{U}(\Theta_0, \theta_0)] \ \ \textit{iff} \ \ \Delta; \Xi; \cdot; \delta \vdash F_1[C_1 \wedge \mathfrak{U}(\Theta_1, \theta_1)]$$

This preservation property is inspired by a similar one holding for HM(X) [Pottier and Rémy 2005, Lemma 10.6.9].

The following progress property states that given a well-formed, non-final state whose representation as a formula is satisfiable, the stack machine can take a step.

Theorem 4 (Progress). *Let* (F, Θ, θ, C) **ok** *and* $F[C] \neq \forall\Delta.\exists\Xi.\mathsf{true}$ *for all* Δ, Ξ*. Further, let* $\cdot; \cdot; \cdot; \emptyset \vdash F[C \wedge \mathfrak{U}(\Theta, \theta)]$*. Then there exists a state* s_1 *such that* $(F, \Theta, \Gamma, \theta, C) \longrightarrow s_1$*.*

Termination is another crucial property.

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

The proof relies on the existence of a well-ordering $<$ on states such that $s \longrightarrow s'$ implies $s' < s$. We observe that the well-ordering cannot simply be defined based on the syntactic size of the in-progress constraint of each state before and after the step, even when plugging the constraint into each state's stack. For example, the rule S-Inst in Figure 9 may introduce an arbitrary number of nested existential constraints. Other rules such as S-ExistsLower may simply reorder stack frames. Therefore, given a state (F, Θ, θ, C), the well-ordering not only takes the size of $F[C]$ and C into account but also the number of instantiation constraints in C and the position of the right-most existential frame in F.

We use the syntax **def** Γ **in** C to denote a series of nested def constraints with C as the innermost constraint, where each def constraints performs a binding from Γ. We now state the overall correctness of the solver as follows: A constraint C is satisfiable in context $\Delta; \Xi; \Gamma$ using instantiation δ if and only if the solver reaches a final state from the input constraint $\forall\Delta.\exists\Xi.$**def** Γ **in** C and δ is a refinement of the substitution θ returned by the solver. Here, a "refinement" of θ is simply a composition with θ.

Theorem 6 (Correctness of constraint solver). *Let $\Delta \vdash \Gamma$ **ok** and $\Delta; \Xi; \Gamma \vdash C$ **ok**. Then we have*

$$\begin{array}{l} \Delta; \Xi; \Gamma; \delta \vdash C \\ \textit{iff} \\ \textit{there exist } \Theta, \theta', \theta, \Xi' \textit{ s.t.} \\ \quad (\cdot, \cdot, \emptyset, \forall\Delta.\, \exists\Xi.\, \mathbf{def}\ \Gamma\ \mathbf{in}\ C) \longrightarrow^* (\forall\Delta :: \exists\ (\Xi, \Xi'), \Theta, \theta, \mathsf{true})\ \textit{and} \\ \quad \Delta \vdash \theta' : \Theta \Rightarrow \cdot\ \textit{and} \\ \quad (\theta' \circ \theta)\restriction_{\Xi} = \delta. \end{array}$$

Even though θ' acts like an instantiation (its codomain only contains rigid variables), it is crucial for it to be a substitution, meaning that it respects the individual restrictions in Θ. An instantiation δ' in place of θ' may violate the restrictions in Θ and introduce polymorphism in places where the type system prohibits it, which would make the right-to-left direction of the theorem invalid. Also note the domain of θ is (Ξ, Ξ'), whereas that of δ is Ξ. Thus, we restrict θ to Ξ when relating it to δ.

Observe that Theorem 6 also states that our solver finds most general solutions: The instantiation θ returned by the solver is independent from δ. Together with the deterministic nature of our solver, this means that any such δ can be obtained from θ.

For the purposes of type-checking, we may now relate the correctness of the solver to constraints resulting from the translation function $[\![-]\!]$ introduced in Section 3.1. If the solver succeeds on the translation of some term M in some context $\Delta; \Gamma$, then the term is well-typed in context $\Delta; \Gamma$ for any well-formed refinement of $\theta(a)$, where a is the placeholder variable used for the type of M.

Theorem 7 (Constraint-based typechecking is sound). *Let $\Delta \vdash \Gamma$ and $\Delta; \Gamma \vdash M$ **ok** and $a \mathbin{\#} \Delta$. If $(\cdot, \cdot, \emptyset, \forall\Delta.\, \exists a.\, \mathbf{def}\ \Gamma\ \mathbf{in}\ [\![M : a]\!]) \longrightarrow^* (\forall\Delta :: \exists\ (a, \tilde{b}), \Theta, \theta, \mathsf{true})$ and $\Delta \vdash \theta' : \Theta \Rightarrow \cdot$ then $\Delta; \Gamma \vdash M : (\theta' \circ \theta)(a)$.*

Conversely, if M has type A, then A can be obtained from instantiating $\theta(a)$.

Theorem 8 (Constraint-based typechecking is complete and most general). *Let $a \mathbin{\#} \Delta$. If $\Delta; \Gamma \vdash M : A$ then there exist $\Xi, \Theta, \theta, \delta$ such that $(\cdot, \cdot, \emptyset, \forall\Delta\,.\exists a\,.\mathbf{def}\ \Gamma\ \mathbf{in}\ [\![M : a]\!]) \longrightarrow^* (\forall\Delta :: \exists\ \Xi, \Theta, \theta, \mathsf{true})$ and $A = \delta(\theta(a))$.*

Remark 4 (mono(−) constraints)**.** *The constraint* $\mathsf{mono}(a)$ *constrains a type variable to be instantiated only with a monotype. Such constraints are not produced during constraint generation and appear in the constraint solving rules only in the rule S-Mono which solves them immediately by checking the current instantiation of a and constraining the free variables of a to be monomorphic. Moreover, because* $\mathbf{def}\ (x : A)\ \mathbf{in}\ C$ *constraints also require A to be monomorphic, we considerd leaving out the* $\mathsf{mono}(a)$ *constraint form altogether and considering it syntactic sugar for* $\mathbf{def}\ (x : a)\ \mathbf{in}\ \mathsf{true}$. *The main reason we have not done this is to simplify the presentation, particularly in the statement and proof of the above results, in order to ensure a simple translation of monomorphism information latent in Θ back into explicit constraints.*

An alternative design we considered would be to attach monomorphism restrictions to existential quantifiers, and drop the monomorphism requirement on **def** *constraints. However, we were not able to find a way to make this work that retains most general solutions for* **def** *constraints, as illustrated in Section 3.2.*

5 DISCUSSION

In this section we discuss two extensions: using ranks for efficiency and unordered quantification. We also compare our approach more directly to Pottier and Rémy's presentation of HM(X).

5.1 Using Ranks

In our solver, the lowering of existential frames in the stack as well as generalisation are controlled by the free type variables in the image of the substitution θ in the state under consideration. Both mechanisms depend on the partition function. A more efficient implementation associates a *rank* with each unification variable [Kuan and MacQueen 2007; Rémy 1992], which can then be used instead of checking what free type variables appear in certain types in the context.

Implementing ranks for our solver requires a similar mechanism to the one described for the HM(X) solver by Pottier and Rémy [2005]; ranks are orthogonal to the support for first-class polymorphism in our system.

We briefly outline how to adapt their mechanism to implement the escape check our solver performs for $\forall$ quantifiers. To this end we associate a rank with each flexible *and* rigid variable b in a given state s, denoted $\mathsf{rank}(b)$. We define $\mathsf{rank}(s)$ as the number of let frames plus the number of $\forall$ frames appearing in F. When the solver encounters a binder for type variable b in state s, the variable's rank is then initialised to be $\mathsf{rank}(s)$. When the unifier detects that some flexible variable a should be substituted with some type A, then the ranks of all variables in A are updated to be no larger than $\mathsf{rank}(a)$. The escape check in the rule S-ForallPop, applied to state $s = (F :: \forall b, \Theta, \theta, \mathsf{true})$, can then be performed by checking that $\mathsf{rank}(b) = \mathsf{rank}(s)$ holds. To partition the variables in Ξ using ranks in the rules S-LetPolyPop and S-LetPolyPop in Figure 9 we use the following modified version of the function partition:

$$\mathsf{partition}'(\Xi, s) = \Xi'; \Xi'' \text{ where } \Xi', \Xi'' = \Xi \text{ and } (\text{for all } a \in \Xi \mid a \in \Xi'' \text{ iff } a \in \mathsf{rank}(a) < \mathsf{rank}(s))$$

Eschewing existential frames. To avoid the need for (inefficiently) lowering existential frames in the stack by swapping with one non-existential frame at a time (as for example in S-ExistsLower), we may optimise the solver further by not carrying individual existential frames in the stack at all. Instead, each state s contains $\mathsf{rank}(s)$ sets of type variables of that rank. We may then remove the rule S-ExistsLower altogether; in S-Let*Pop the variables $\widetilde{a'}$ are determined by taking exactly those of rank $\mathsf{rank}(s)$ (the set $\widetilde{a''}$ isn't needed anymore in this setting).

5.2 Unordered FreezeML

So far we have considered a syntactic equational theory on types that equates quantified types up to alpha-equivalence only and does not allow for any reordering of quantifiers or the removal/addition of unused ones. This is in line with the original presentation of FreezeML [Emrich et al. 2020].

However, this is not a fundamental requirement of the system. We may define *Unordered FreezeML*, a variant of FreezeML where quantifiers are unordered, by redefining equality of types to allow $\forall ab.a \to b = \forall ba.a \to b = \forall abc.a \to b$ and consider ftv to return sets of variables rather than sequences. The typing rules of Unordered FreezeML can than be obtained from Figure 1 by replacing every occurrence of $\overline{a}$ by $\tilde{a}$. Likewise, type inference for Unordered FreezeML can be performed using a stack machine using the same rules as shown in Figure 9. The only change is to replace the unification algorithm $\mathcal{U}$ with an alternative one ignoring the order of quantifiers as well as unnecessary ones. This is of course not trivial because unification can no longer assume that when it encounters a $\forall$ on one side of an equation, the other side must be a $\forall$ binding the same (modulo alpha-equivalence) variable. Unification must be modified to handle the case where one side $\forall$-binds an unused variable (e.g. $\forall a.int \sim int$) or where bindings must be reordered (e.g. $\forall a, b, c.a \to b \to c \sim \forall a, b, c.c \to b \to a$). The point is that this complexity seems to be confined to unification and the rest of the system is unchanged.

5.3 Comparison with HM(X) Solver by Pottier and Rémy

The solver presented in this section is inspired by the one presented by Pottier and Rémy [2005] for HM(X), adding support for first-class polymorphism in the style of FreezeML.

Additional notable differences include the following:

- In the HM(X) solver, configurations carry a collection U of unification constraints. It can be interpreted as a conjunction built from a subset of the constraint language with additional well-formedness restrictions. This means that when extending the constraint language, the definition of constraint permitted in U can be adapted accordingly.
  Our solver represents the unifier context with two separate components Θ and θ. This is mostly for the purpose of making the system more similar to the original type inference algorithm of FreezeML. Our system already provides a mechanism for representing the unification context as a constraint, in the form of $\mathfrak{U}(\Theta, \theta)$, defined in Section 4.2 for the purposes of our meta-theory. It would be straightforward to define unification contexts in our solver in terms of $\mathfrak{U}(\Theta, \theta)$ (or a more structured representation thereof using multi-equations) instead of Θ and θ.
- The solver presented by Pottier and Rémy supports recursive types by allowing the solver state to contain constraints of the form $a \sim A$, where $A \neq a$ and $a \in \mathsf{ftv}(A)$. In our system, a corresponding state with $\theta(a) = A$ for the same A would be ill-formed, as we require θ to be idempotent.
  We consider support for recursive types as orthogonal to the issue of supporting first-class polymorphism, but our reliance on the idempotency of θ would require adding explicit μ types for handling recursive types.
- The HM(X) solver implements several optimisations, for example mechanisms to reduce the number of type variables present in states. Similar optimisations could be performed by our solver, but we eschew them for the sake of brevity, including the usage of ranks described in Section 5.1.
- In the HM(X) solver, def constraints and term variables efficiently handle sharing the work of type inference for let-generalized values, but they are not strictly necessary: def constraints can be eliminated by a form of constraint inlining. Doing so would not change the results of type inference but would be disastrous for performance. In constrast, our system uses term variables as a means of keeping first-class polymorphism tractable. Term variables are used as placeholders for polymorphic types that we may instantiate, which is why we require that the polymorphism in these types is always fully determined. Flexible type variables can also become bound to arbitrarily polymorphic types (e.g. during type inference for $id\ \lceil id \rceil$) but the quantifiers occurring in these types are never instantiated during constraint solving, they can only be unified with other quantifiers. By ensuring that all polymorphism in the term context is fully known, we guarantee that we do not instantiate unknown polymorphism. On the other hand, it does not appear possible to eliminate def constraints via substitution or to define let constraints in terms of def, as in Pottier and Rémy work.

Remark 5 (Itches we haven't been able to scratch yet). *An alternative system without term variables in the constraint language should be possible, in which case instantiation of quantifiers occuring in types bound to type variables would need to take place. This would require an additional mechanism to guarantee that the polymorphism of those type variables that could be instantiated in this version is fixed. We haven't adopted this alternative design as we consider the current design that uses term variables to be closer to Pottier and Rémy's work and the original version of FreezeML.*

It also may be possible to recover the property that let and def constraints canbe "expanded away". One possibility (suggested by a reviewer) is to seek a suitable adjustment of the notion of constraint *inlining that accounts for frozen constraints, perhaps by generalizing them to permit a local type constraint ($\lceil x : [C]A \rceil$). As discussed in 6, allowing for full constrained types in our setting poses challenges, and may also encounter similar issues to those encountered in equational reasoning about FreezeML terms, which was considered briefly by Emrich et al. [2020]. This general issue deserves further investigation.*

Further discussion of these two issues, which are interrelated, is the extended version [Emrich et al. 2022].

6 RELATED WORK

Constraint-based type inference for Hindley-Milner and related systems has a long history [Wand 1987]. Some of the most relevant systems include qualified types [Jones 1994], HM(X) [Odersky et al. 1999], OutsideIn(X) [Vytiniotis et al. 2011], and GI [Serrano et al. 2018] which present increasingly sophisticated techniques for solving (generalisations of) constraints generated from ML or Haskell-like programs. Our work differs in building on HM(X) as presented by Pottier and Rémy [2005], while adapting it to support first-class polymorphism based on the FreezeML approach. On the other hand, constraint-based FreezeML does not so far support constraint solving parameterised by an arbitrary constraint domain X, and extending it to support this is a natural but nontrivial next step. In particular, FreezeML uses exactly System F types, rather than the type schemes with constraint components of the form $\forall \overline{a}.C \Rightarrow A$ found in HM(X). We have compared our constraint solver to the HM(X) solver by Pottier and Rémy in Section 5.3.

FreezeML is also related to PolyML as explained by Emrich et al. [2020]. Unlike FreezeML, PolyML uses two different sorts of polymorphic types: ML-like type schemes and first-class polymorphic types. The latter may only be introduced with explicit type annotations. As a result, the conditions to pick most general solutions in the semantics of certain constraints in our language are not necessary in PolyML.

The type system of GI [Serrano et al. 2018] uses carefully crafted rules for n-ary function applications, determining when arguments' types may be generalised or instantiated. It does not perform let generalisation. Its type inference system is built on constraint solving, using a different approach towards restricting polymorphism. It syntactically distinguishes three sorts of unification variables, which may only be instantiated with monomorphic, guarded, or fully polymorphic types. While our solver determines the order of constraint solving using a stack, their system allows individual rules to be blocked until progress has been made elsewhere, for example waiting until a fully polymorphic variable has been substituted with a more concrete type.

QuickLook [Serrano et al. 2020] combines Hindley-Milner style type inference with bidirectional type inference in a subtle way, and when typechecking applications of polymorphically typed variables, performs a "quick look" at all of the arguments; this amounts to a sound but shallow analysis whether there is a unique type instantiation (possibly involving polymorphism). If there is a unique type instantiation then that instantiation is chosen, otherwise quantified variables are instantiated with monomorphic flexible variables. Type inference for QuickLook follows a two-stage approach: all first-class polymorphism is resolved at constraint generation time, and the actual constraint solver does not have to find solutions for polymorphic type variables. Consequently, their constraint language and solver are completely standard and oblivious to first-class polymorphism. Thus, QuickLook requires only small modifications to existing Haskell-style type inference, including extensions such as qualified types and GADTs, but (like other recent proposals such as GI) does not support let-bound polymorphism nor come with a formal completeness result. In an appendix the authors discuss approaches to supporting let generalisation; one is to use let constraints in the style of Pottier and Rémy, and that is what we do.

Some aspects of our solver are reminiscent of the approach taken in Type Inference in Context by Gundry et al. [2010], though their approach performs type inference as a traversal of source language terms rather than introducing an intermediate constraint language. We are interested in adapting their approach to FreezeML type inference, particularly leveraging the insight that type inference monotonically increases knowledge about possible solutions (reflected in the structure of their contexts).

Returning to the motivation for this work mentioned in the introduction, it is a natural to ask what obstacles remain to generalizing our system to handle an arbitrary constraint domain (the "X" in HM(X)). The immediate obstacle is how to handle constrained or qualified types $C \Rightarrow A$ which are considered equivalent up to reordering constraints C. Such types need not induce, and depending on the theory X such types may have equivalent forms with different numbers of quantified types. Adopting the unordered quantification approach in Section 5.2 could help with this, but we leave this and the investigation of generalizing to a "FreezeML(X)" to future work.

7 CONCLUSIONS

Emrich et al. [2020] recently introduced FreezeML, a new approach to ML-style type inference that supports the full power of System F polymorphism using type and term annotations to control instantiation and generalisation of polymorphic types. Their initial type inference algorithm was a straightforward extension of Algorithm W. We have introduced Constraint FreezeML, an alternative constraint-based presentation of FreezeML type inference, opening up many possibilities for extending FreezeML in the future. We extended the constraint language of HM(X) with suitable constraints, equipped with a semantics and translation from FreezeML programs to constraints that encode type inference problems, and presented a deterministic, terminating state machine for solving the constraints. Several potential next steps are opened by this work, including generalising to support arbitrary constraint domains (the "X" in HM(X)), implementing the solver efficiently using ranks, and considering recursive types and higher kinds.

ACKNOWLEDGMENTS

This work was supported by ERC Consolidator Grant Skye (grant number 682315) and by an ISCF Metrology Fellowship grant provided by the UK government's Department for Business, Energy and Industrial Strategy (BEIS). Lindley is supported by UKRI Future Leaders Fellowship "Effect Handler Oriented Programming" (MR/T043830/1).

REFERENCES

Luís Damas and Robin Milner. 1982. Principal Type-Schemes for Functional Programs. In *POPL*. ACM Press, 207–212. https://doi.org/10.1145/582153.582176

Frank Emrich, Sam Lindley, Jan Stolarek, James Cheney, and Jonathan Coates. 2020. FreezeML: Complete and Easy Type Inference for First-class Polymorphism. In *PLDI*. ACM, 423–437. Extended version available at https://doi.org/10.48550/arXiv.2004.00396.

Frank Emrich, Jan Stolarek, James Cheney, and Sam Lindley. 2022. *Constraint-based type inference for FreezeML (extended version)*. Technical Report. https://doi.org/10.48550/arXiv.2207.09914 arXiv:2207.09914.

Jacques Garrigue and Didier Rémy. 1999. Semi-Explicit First-Class Polymorphism for ML. *Inf. Comput.* 155, 1-2 (1999), 134–169. https://doi.org/10.1006/inco.1999.2830

Adam Gundry. 2015. A typechecker plugin for units of measure: domain-specific constraint solving in GHC Haskell. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, Ben Lippmeier (Ed.). ACM, 11–22. https://doi.org/10.1145/2804302.2804305

Adam Gundry, Conor McBride, and James McKinna. 2010. Type Inference in Context. In *MSFP@ICFP*. ACM, 43–54. https://doi.org/10.1006/inco.1999.2830

Mark P. Jones. 1994. A Theory of Qualified Types. *Sci. Comput. Program.* 22, 3 (1994), 231–256. https://doi.org/10.1016/0167-6423(94)00005-0

Andrew Kennedy. 2009. Types for Units-of-Measure: Theory and Practice. In *CEFP (Lecture Notes in Computer Science, Vol. 6299)*. Springer, 268–305. https://doi.org/10.1007/978-3-642-17685-2_8

George Kuan and David MacQueen. 2007. Efficient type inference using ranked type variables. In *ML*. ACM, 3–14. https://doi.org/10.1145/1292535.1292538

Didier Le Botlan and Didier Rémy. 2003. ML^{F}: raising ML to the power of System F. In *ICFP*. ACM, 27–38. https://doi.org/10.1145/944705.944709

Daan Leijen. 2008. HMF: simple type inference for first-class polymorphism. In *ICFP*. ACM, 283–294. https://doi.org/10.1145/1411204.1411245

Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. In *MSFP (EPTCS, Vol. 153)*. 100–126. https://doi.org/10.4204/EPTCS.153.8

Xavier Leroy and Michel Mauny. 1993. Dynamics in ML. *J. Funct. Program.* 3, 4 (1993), 431–463. https://doi.org/10.1017/S0956796800000848

Sam Lindley and James Cheney. 2012. Row-based effect types for database integration. In *TLDI*. ACM, 91–102. https://doi.org/10.1145/2103786.2103798

J. Garrett Morris and James McKinna. 2019. Abstracting extensible data types: or, rows by any other name. *Proc. ACM Program. Lang.* 3, POPL (2019), 12:1–12:28. https://doi.org/10.1145/3290325

Martin Odersky, Martin Sulzmann, and Martin Wehr. 1999. Type Inference with Constrained Types. *Theory Pract. Object Syst.* 5, 1 (1999), 35–55.

Frank Pfenning. 1993. On the Undecidability of Partial Polymorphic Type Reconstruction. *Fundam. Inform.* 19, 1/2 (1993), 185–199.

François Pottier. 2014. Hindley-Milner Elaboration in Applicative Style: Functional Pearl. In *ICFP*. ACM, 203–212. https://doi.org/10.1145/2628136.2628145

François Pottier and Didier Rémy. 2005. *The Essence of ML Type Inference.* MIT Press, Chapter 10, 389–489.

Didier Rémy. 1992. *Extension of ML Type System with a Sorted Equational Theory on Types.* Technical Report RR-1766. Institut National de Recherche en Informatique et en Automatique.

Claudio V. Russo and Dimitrios Vytiniotis. 2009. QML: Explicit First-class Polymorphism for ML. In *ML*. ACM, 3–14.

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

Alejandro Serrano, Jurriaan Hage, Dimitrios Vytiniotis, and Simon Peyton Jones. 2018. Guarded impredicative polymorphism. In *PLDI*. ACM, 783–796. https://doi.org/10.1145/3192366.3192389

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

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

Dimitrios Vytiniotis, Stephanie Weirich, and Simon L. Peyton Jones. 2006. Boxy types: inference for higher-rank types and impredicativity. In *ICFP*. ACM, 251–262. https://doi.org/10.1145/1159803.1159838

Mitchell Wand. 1987. A simple algorithm and proof for type inference. *Fundamenta Informaticae* (1987).

J. B. Wells. 1994. Typability and Type-Checking in the Second-Order lambda-Calculus are Equivalent and Undecidable. In *LICS*. IEEE Computer Society, 176–185. https://doi.org/10.1109/LICS.1994.316068

Andrew K. Wright. 1995. Simple Imperative Polymorphism. *LISP Symb. Comput.* 8, 4 (1995), 343–355.