Constraint-based type inference for FreezeML

FRANK EMRICH, JAN STOLAREK*, JAMES CHENEY†, and 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.

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 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

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

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

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 an appendix.

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. Besides needing to deal with polymorphism in types, 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 we require 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 1991; 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 an appendix.

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, 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], and 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 (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 $\{X_1, \ldots, X_n\}$ and $\overline{X}$ for a (possibly empty) sequence $X_1, \ldots, X_n$. We overload comma for use as a union / concatenation operator for sets and sequences, writing $\tilde{X}, \tilde{Y}$ for the set $\{X_1, \ldots, X_m, Y_1, \ldots, Y_n\}$ where $\tilde{X} = \{X_1, \ldots, X_m\}$ and $\tilde{Y} = \{Y_1, \ldots, Y_n\}$, and writing $\overline{X}, \overline{Y}$ for the sequence $X_1, \ldots, X_m, Y_1, \ldots, Y_n$ where $\overline{X} = X_1, \ldots, X_m$ and $\tilde{Y} = Y_1, \ldots, Y_n$. 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.

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

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), without the need to distinguish polymorphic types from some other sort of quantified types. 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).

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

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

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

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

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

Fig. 1. FreezeML Typing Rules.

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.

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, Δ' with types subject to restriction R using the type context Δ, Δ'', where every variable in Δ is mapped to itself. In order for this interpretation to make sense the judgement has an implicit precondition that Δ, Δ', and Δ'' are pairwise disjoint. 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.

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 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; 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 is 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 being a principal type using the predicate $\mathsf{principal}(\Delta, \Gamma, M, \Delta', A')$. It asserts that A' is the principal type of M in the context $\Delta; \Gamma$, and Δ' comprises those type variables of A' not in

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

Δ. In order for the predicate to be satisfied, A' must be a valid type of M in the context $(\Delta, \Delta'); \Gamma$ (adding Δ' to Δ allows variables from both to appear in A') and any other type of M can be obtained by instantiating the variables in Δ'.

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

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 founded 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 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.

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

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

Values disallow applications. Guarded values disallow frozen variables, and thus must have guarded type.[2]

Plain let bindings ($\mathbf{let}\ x = M\ \mathbf{in}\ N$) 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, 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]. Consider the term

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

If we generalise the principal type $a \to a$ of $\lambda x.x$ (where $a \notin \Delta$), we obtain $\forall a.a \to a$ as the type of f, which then becomes the type of the overall let term due to f being frozen in its body. If the typing rule permitted using other types of $\lambda x.x$, such as $\mathsf{Int} \to \mathsf{Int}$, then generalisation would have no effect. This would make $\mathsf{Int} \to \mathsf{Int}$ 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 $\mathsf{Int} \to \mathsf{Int}$ and $\forall a.a \to a$ 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*.

[2]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.

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

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

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 principle type of a plain let binding.

Annotated let bindings. Annotated let bindings (**let** $(x : A) = M$ **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.

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

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 $\Delta; \Gamma \vdash M$ **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 $\Delta; \Gamma \vdash M$ **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.

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

$$\frac{x \in \Gamma}{\Delta; \Gamma \vdash \lceil x \rceil \textbf{ ok}} \qquad \frac{x \in \Gamma}{\Delta; \Gamma \vdash x \textbf{ ok}} \qquad \frac{\Delta; (\Gamma, x : A) \vdash M \textbf{ ok}}{\Delta; \Gamma \vdash \lambda x.M \textbf{ ok}} \qquad \frac{\Delta \vdash A \qquad \Delta; (\Gamma, x : A) \vdash M \textbf{ ok}}{\Delta; \Gamma \vdash \lambda (x : A).M \textbf{ ok}}$$

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

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

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. The definition constraint **def** $(x : A)$ **in** C binds term variable x in C. (It also includes a side-constraint which we will return to shortly.) The polymorphic let constraint $\mathbf{let}_\star\ x = \sqcap a.C_1$ **in** C_2 and monomorphic let constraint $\mathbf{let}_\bullet\ x = \sqcap a.C_1$ **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. Monomorphism constraints $\mathsf{mono}(a)$ assert that the flexible variable a must only be instantiated with monotypes.

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 generated for the body of the lambda abstraction. An annotated lambda abstraction generates

$$\textsc{Sem-True}\ \frac{}{\Delta;\Xi;\Gamma;\delta \vdash \mathsf{true}} \qquad \textsc{Sem-And}\ \frac{\Delta;\Xi;\Gamma;\delta \vdash C_1 \qquad \Delta;\Xi;\Gamma;\delta \vdash C_2}{\Delta;\Xi;\Gamma;\delta \vdash C_1 \wedge C_2} \qquad \textsc{Sem-Equiv}\ \frac{(\Delta,\Xi) \vdash_\star A\ \mathbf{ok} \qquad (\Delta,\Xi) \vdash_\star B\ \mathbf{ok} \qquad \delta(A) = \delta(B)}{\Delta;\Xi;\Gamma;\delta \vdash A \sim B}$$

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

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

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

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

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

Fig. 3. Satisfiability judgement for constraints.

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 $\mathbf{let}\ (x : B) = M\ \mathbf{in}\ N$ 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.

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

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

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

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

Fig. 4. Well-formedness of constraints.

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

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_2 and every other one can be obtained by refining δ_m by composition.

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

The rule Sem-LetPoly then defines two subsets[3] 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.

[3] 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 $\Delta \vdash \delta' : (\Delta_\mathrm{o}, \overline{a}) \Rightarrow \cdot$. Observe that the variables in $\Delta_\mathrm{m} - (\Delta_\mathrm{o}, \overline{a})$ 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 appendix (see anonymous supplementary material).

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.

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

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

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

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

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 $\mathrm{rc}(F) \vdash \theta : \Theta \Rightarrow \Theta$, that θ is idempotent, that C is well-formed ($\mathrm{rc}(F); \mathrm{fc}(F); \mathrm{tc}(F) \vdash C\ \mathbf{ok}$), 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.

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.

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

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

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

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

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

Fig. 8. Stack well-formedness.

$$(F, \Theta, \theta, A \sim B) \rightarrow (F, \Theta', \theta' \circ \theta, \mathsf{true}) \text{ where } (\Theta', \theta') = \mathcal{U}(\mathsf{rc}(F), \Theta, \theta A, \theta B) \quad \text{(S-Eq)}$$

$$(F, \Theta, \theta, \lceil x : A \rceil) \rightarrow (F, \Theta, \theta, \mathsf{tc}(F)(x) \sim A) \quad \text{(S-Freeze)}$$

$$(F, \Theta, \theta, x \preceq A) \rightarrow (F, \Theta, \theta, \exists \overline{a}.H \sim A) \text{ where } \forall \overline{a}.H = \mathsf{tc}(F)(x) \quad \tilde{a} \# \mathsf{btv}(F) \quad \text{(S-Inst)}$$

$$(F, \Theta, \theta, \mathsf{mono}(a)) \rightarrow (F, \Theta', \theta, \mathsf{true}) \quad \text{(S-Mono)}$$
$$\text{where } \overline{b} = \mathsf{ftv}(\theta(a)) - \mathsf{rc}(F) \quad \Theta' = (\Theta - \tilde{b}) \cup \widetilde{b : \bullet} \quad \Theta' \vdash_\bullet \theta(a)\ \mathbf{ok}$$

$$(F, \Theta, \theta, C_1 \wedge C_2) \rightarrow (F :: \square \wedge C_2, \Theta, \theta, C_1) \quad \text{(S-ConjPush)}$$

$$(F :: \square \wedge C_2, \Theta, \theta, \mathsf{true}) \rightarrow (F, \Theta, \theta, C_2) \quad \text{(S-ConjPop)}$$

$$(F, \Theta, \theta, \exists a.\, C) \rightarrow (F :: \exists a, (\Theta, a : \star), \theta[a \mapsto a], C) \quad \text{(S-ExistsPush)}$$

$$(F :: f :: \exists \overline{a}, \Theta, \theta, \mathsf{true}) \rightarrow (F :: \exists \tilde{c} :: f, \Theta', \theta\restriction_{\Theta'}, \mathsf{true}) \quad \text{(S-ExistsLower)}$$
$$\text{where } f \text{ is neither a } \mathbf{let} \text{ or } \exists \text{ frame} \quad \tilde{b};\ \tilde{c} = \mathsf{partition}(\tilde{a}, \theta, \Theta) \quad \Theta' = \Theta - \tilde{b} \quad |\tilde{a}| > 0$$

$$(F, \Theta, \theta, \forall a.C) \rightarrow (F :: \forall\, a, \Theta, \theta, C) \quad \text{(S-ForallPush)}$$

$$(F :: \forall\, a, \Theta, \theta, \mathsf{true}) \rightarrow (F, \Theta, \theta, \mathsf{true}) \text{ where } a \notin \mathsf{ftv}(\theta(\Theta)) \quad \text{(S-ForallPop)}$$

$$(F, \Theta, \theta, \mathbf{def}\ (x : A)\ \mathbf{in}\ C) \rightarrow (F :: \mathbf{def}\ (x : A), \Theta', \theta, C) \quad \text{(S-DefPush)}$$
$$\text{where } \overline{b} = \mathsf{ftv}(\theta(A)) - \mathsf{rc}(F) \quad \Theta' = (\Theta - \tilde{b}) \cup \widetilde{b : \bullet} \quad \text{for all } a \in \mathsf{ftv}(A) \mid \Theta' \vdash_\bullet \theta(a)\ \mathbf{ok}$$

$$(F :: \mathbf{def}\ (x : A), \Theta, \theta, \mathsf{true}) \rightarrow (F, \Theta, \theta, \mathsf{true}) \quad \text{(S-DefPop)}$$

$$(F, \Theta, \theta, \mathbf{let}_R\ x = \sqcap b.C_1\ \mathbf{in}\ C_2) \rightarrow (F :: \mathbf{let}_R\ x = \sqcap b.\square\ \mathbf{in}\ C_2, (\Theta, b : \star), \theta[b \mapsto b], C_1) \quad \text{(S-LetPush)}$$

$$(F :: \mathbf{let}_\star\ x = \sqcap b.\square\ \mathbf{in}\ C :: \exists \overline{a}, \Theta, \theta, \mathsf{true}) \rightarrow (F :: \exists \tilde{a}'', \Theta', \theta\restriction_{\Theta'}, \mathbf{def}\ (x : B)\ \mathbf{in}\ C) \quad \text{(S-LetPolyPop)}$$
$$\text{where } \tilde{a}';\ \tilde{a}'' = \mathsf{partition}((\tilde{a}, b), \theta, \Theta) \quad A = \theta(b) \quad \overline{c} = \mathsf{ftv}(A) \cap \tilde{a}' \quad \Theta' = \Theta - \tilde{a}' \quad B = \forall \overline{c}.A$$

$$(F :: \mathbf{let}_\bullet\ x = \sqcap b.\square\ \mathbf{in}\ C :: \exists \overline{a}, \Theta, \theta, \mathsf{true}) \rightarrow (F :: \exists (\tilde{c}, \tilde{a}''), \Theta', \theta\restriction_{\Theta'}, \mathbf{def}\ (x : A)\ \mathbf{in}\ C) \quad \text{(S-LetMonoPop)}$$
$$\text{where } \tilde{a}';\ \tilde{a}'' = \mathsf{partition}((\tilde{a}, b), \theta, \Theta) \quad A = \theta(b) \quad \overline{c} = \mathsf{ftv}(A) \cap \tilde{a}' \quad \Theta' = \Theta - (\tilde{a}' - \tilde{c})$$

Fig. 9. Constraint solving rules.

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 $(\forall \Delta :: \exists \Xi, \Theta, \theta, \mathsf{true})$ from

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

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

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

$\mathcal{U}(\Delta, \Theta, D\,\overline{A}, D\,\overline{B}) =$
  let $(\Theta_1, \theta_1) = (\Theta, \theta_{\mathrm{id}})$
  let $n = \mathsf{arity}(D)$
  for $i \in 1...n$
    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 $(\Theta_{n+1}, \theta_{n+1})$

$\mathcal{U}(\Delta, \Theta, \forall a.A, \forall b.B) =$
  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 (Θ_1, θ')

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

Fig. 10. Unification algorithm.

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 of no variable further down in the stack depending 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.

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 $\bigwedge_{a \in \mathsf{ftv}(A) - \mathrm{rc}(F)} \mathsf{mono}(a)$ 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 $\mathbf{let}_R\ x = \sqcap b.C_1$ **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 $\tilde{a}'$ and $\tilde{a}''$. 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 $\tilde{a}'$ (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 $\tilde{a}''$ must be preserved and are lowered in the stack. Note that $\tilde{a}''$ may or may not contain b.

The type A for x is then determined by generalising $\theta(b)$. The variables $\overline{c}$ to be generalised are obtained from taking those free type variables of $\theta(b)$ that also appear in $\tilde{a}'$, meaning that no flexible variable in a lower frame depends on them. In particular, if Ξ denotes the free unification variables appearing in $\theta(\mathrm{tc}(F))$, we have $\Xi \# \tilde{a}' \supseteq \overline{c}$. Recall that ftv applied to a type yields an ordered sequence, and we assume that the ordering is preserved under intersection. 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:

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

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:

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

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*

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

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*

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

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* $a \# \Delta$. *If* $(\cdot, \cdot, \emptyset, \forall\Delta.\, \exists a.\, \mathbf{def}\ \Gamma\ \mathbf{in}\ [\![M : a]\!]) \to^* (\forall\Delta :: \exists\ (a, \tilde{b}), \Theta, \theta, \mathsf{true})$ *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* $a \# \Delta$. *If* $\Delta; \Gamma \vdash M : A$ *then there exist* $\Xi, \Theta, \theta, \delta$ *such that* $(\cdot, \cdot, \emptyset, \forall\Delta\,.\exists a\,.\mathbf{def}\ \Gamma\ \mathbf{in}\ [\![M : a]\!]) \to^* (\forall\Delta :: \exists\ \Xi, \Theta, \theta, \mathsf{true})$ *and* $A = \delta(\theta(a))$.

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 unifying a flexible variable b and a type A, the rank of b and of all free flexible variables of A are then updated to the minimum rank of all variables in this set. The rank of any rigid variable a in A is updated to the new rank of b if it is lower than the previous rank of a.

We may then use the following updated version of the function partition in the rules S-LetPolyPop and S-LetPolyPop in Figure 9:

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

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.

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 $\tilde{a}'$ are determined by taking exactly those of rank $\mathsf{rank}(s)$ (the set $\tilde{a}''$ 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.

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 $a \in \mathrm{ftv}(A)$. 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 are used for the sake of efficiency, to avoid duplication of constraints when instantiating. In constrast, our system uses term variables as a means of keeping first-class polymorphism tractable — we only allow instantiation of term variables, rather than type variables. By ensuring that all polymorphism in the term context is fully known, we guarantee that we do not instantiate unknown polymorphism.

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 bub 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. QuickLook requires only small modifications to existing Haskell-style type inference, including extensions such as qualified types and GADTs, but (like other recent proposals) does not support let-bound polymorphism nor come with a formal completeness result.

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).

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.

REFERENCES

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2006. Links: Web Programming Without Tiers. In *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures (Lecture Notes in Computer Science, Vol. 4709)*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever (Eds.). Springer, 266–296. https://doi.org/10.1007/978-3-540-74792-5_12

Luís Damas and Robin Milner. 1982. Principal Type-Schemes for Functional Programs. In *POPL*. ACM Press, 207–212.

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.

Jacques Garrigue and Didier Rémy. 1999. Semi-Explicit First-Class Polymorphism for ML. *Inf. Comput.* 155, 1-2 (1999), 134–169.

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.

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.

George Kuan and David MacQueen. 2007. Efficient type inference using ranked type variables. In *ML*. ACM, 3–14.

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

Daan Leijen. 2008. HMF: simple type inference for first-class polymorphism. In *ICFP*. ACM, 283–294.

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

Xavier Leroy and Michel Mauny. 1991. Dynamics in ML. In *FPCA*. Springer, 406–426.

Sam Lindley and James Cheney. 2012. Row-based effect types for database integration. In *TLDI*. ACM, 91–102.

Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *J. Funct. Program.* 18, 1 (2008), 1–13. https://doi.org/10.1017/S0956796807006326

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.

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.

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.

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.

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

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.

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

A PROOFS FOR SECTION 3.4

The proofs of Theorems 1 and 2 proceed via mutual induction on the structure of the term M. Both proofs use the following lemma, but only on subterms of the term M in question.

Lemma 9. *Let* $\Delta' = \mathsf{ftv}(A) - \Delta$ *and* $\Delta; \Gamma \vdash M$ **ok**. *Then* $\mathsf{principal}(\Delta, \Gamma, M, \Delta', A)$ *iff* $\mathsf{mostgen}(\Delta, (a), \Gamma, [\![M : a]\!], \Delta', [a \mapsto A])$.

The proof of Lemma 9 in turn uses both theorems directly.

We proceed by collecting auxiliary lemmas (including Lemma 9) in Appendix A.1. The two subsequent subsections contain the proofs of Theorem 1, Theorem 2, respectively.

A.1 Auxiliary Lemmas

Lemma 10. *Let* $M \in \mathsf{GVal}$ *and* $\mathsf{principal}(\Delta, \Gamma, M, \Delta', A)$. *Then* A *is a guarded type.*

Proof. The only way for A to be a top-level polymorphic type of a guarded value M is if M is a plain (i.e, not frozen) variable x of type $\forall a_0, \ldots, a_n.a_i$. However, the *principal* type of x is a for some fresh polymorphic variable $a \in \Delta'$, which is a guarded type. □

Lemma 11 (Well-formedness of constraint translation)**.** *Let* $\Delta; \Gamma \vdash M$ **ok** *and* $(\Delta, \Xi) \vdash A$ **ok**. *Then* $\Delta; \Xi; \Gamma \vdash [\![M : A]\!]$ **ok** *holds.*

Proof. By induction on structure of M. We observe that the only free type variables of $[\![M : A]\!]$ are those appearing freely in A and in type annotations appearing in M. By $\Delta; \Gamma \vdash M : A$ we have that all such free type variables in the annotations in M are rigid variables from Δ. Hence, the only free unification variables of $[\![M : A]\!]$ are those in A.

□

Lemma 12 (Satisfiability implies well-formedness)**.** *If* $\Delta; \Xi; \Gamma; \delta \vdash C$ *then* $\Delta; \Xi; \Gamma \vdash C$ **ok**.

Proof. We observe that the rules Sem-Equiv, Sem-Freeze, Sem-Inst Sem-Def all (explicitly or implicitly) require the types found in the constraint under consideration to be well-formed in the context $\Delta; \Xi$. The rule Sem-Mono imposes $a \in (\Delta, \Xi)$. Finally, the rules Sem-Freeze, Sem-Inst require $x \in \Gamma$. □

See the note at the beginning of Appendix A, for an explanation of the dependencies between Theorem 1, Theorem 2 and the Lemma 9.

Lemma 9. *Let* $\Delta' = \mathsf{ftv}(A) - \Delta$ *and* $\Delta; \Gamma \vdash M$ **ok**. *Then* $\mathsf{principal}(\Delta, \Gamma, M, \Delta', A)$ *iff* $\mathsf{mostgen}(\Delta, (a), \Gamma, [\![M : a]\!], \Delta', [a \mapsto A])$.

Proof. Recall the definitons of principal and mostgen:

$$
\begin{aligned}
&\mathsf{principal}(\Delta, \Gamma, M, \Delta', A) = \\
&\quad \Delta, \Delta'; \Gamma \vdash M : A \ \text{ and} && \textbf{(1)} \\
&\quad (\text{for all } \Delta'', A'' \mid \text{if } \Delta, \Delta''; \Gamma \vdash M : A'' \\
&\qquad\qquad \text{then there exists } \delta \text{ such that} && \textbf{(2)} \\
&\qquad\qquad \Delta \vdash \delta : \Delta' \Rightarrow_\star \Delta'' \text{ and } \delta(A) = A'')
\end{aligned}
$$

$$\begin{aligned}&\mathsf{mostgen}(\Delta, a, \Gamma, \delta, [\![M : a]\!], \Delta', [a \mapsto A]) = \\ &\quad (\Delta, \Delta'); a; \Gamma; [a \mapsto A] \vdash [\![M : a]\!] \text{ and} && \textbf{(3)} \\ &\quad (\text{for all } \Delta'', \delta'' \mid \text{if } (\Delta, \Delta''); a; \Gamma; \delta'' \vdash [\![M : a]\!] \\ &\qquad\qquad \text{then there exists } \delta \text{ such that} && \textbf{(4)} \\ &\qquad\qquad \Delta \vdash \delta : \Delta' \Rightarrow_\star \Delta'' \text{ and } \delta'' = \delta \circ [a \mapsto A])\end{aligned}$$

⇒ We apply Theorem 1 to (1), immediately yielding the desired property (3).
To show (4), we assume $(\Delta, \Delta''); a; \Gamma; \delta'' \vdash [\![M : a]\!]$, which implies that $\delta'' = [a \mapsto A'']$ for some A''.
By Theorem 2 this gives us $(\Delta, \Delta''); \Gamma \vdash M : A''$. According to (2), there exists a δ with the desired properties.

⇐ We apply Theorem 2 to (3), which gives us satisfaction of property (1).
To show (4), we assume $\Delta, \Delta''; \Gamma \vdash M : A''$. Theorem 1 then gives us $(\Delta, \Delta''); a; \Gamma; [a \mapsto A''] \vdash [\![M : a]\!]$. According to (4) there exists an appropriate δ.

□

A.2 Proof of Theorem 1

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.*

Proof. By structural induction on M, focusing on the let cases.

Case let $x = M'$ **in** N'**, where** $M' \in \mathsf{GVal}$ The derivation of $\Delta; \Gamma \vdash M : A$ has the following form, for some $B, B', \overline{a}$.

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

By $M' \in \mathsf{GVal}$ we have $B = \forall \overline{a}.B'$ and $[\![M : a]\!] = \mathbf{let}_\star\ x = \sqcap b.[\![M' : b]\!]\ \mathbf{in}\ [\![N' : a]\!]$. We assume w.l.o.g. that $b \#(\Delta, \tilde{a})$. By induction we have $(\Delta, \tilde{a}); b; \Gamma; [b \mapsto B'] \vdash [\![M' : b]\!]$ and $\Delta; a; (\Gamma, x : B); [a \mapsto A] \vdash [\![N' : a]\!]$. We can weaken the former to $(\Delta, \tilde{a}); (a, b); \Gamma; [a \mapsto A, b \mapsto B'] \vdash [\![M' : b]\!]$ By Lemma 9, $\mathsf{principal}(\Delta, \Gamma, M', \tilde{a}, B')$ implies $\mathsf{mostgen}(\Delta, b, \Gamma, [\![M' : b]\!], \tilde{a}, [b \mapsto B'])$. According to Lemma 18 we can weaken this to $\mathsf{mostgen}(\Delta, (a, b), \Gamma, [\![M' : b]\!], (\tilde{a}, c), [a \mapsto c, b \mapsto B'])$ for some fresh c.
Let $\Delta_\mathrm{o} \coloneqq c$ and $\delta' \coloneqq [c \mapsto \mathsf{unit}]$ and $\delta_\mathrm{m} = [a \mapsto c, b \mapsto B']$. Due to $a \notin \mathsf{ftv}(B')$ we have $\delta'(B') = B'$. We can then derive the following

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

Case let $x = M'$ **in** N'**, where** $M' \notin \mathsf{GVal}$ We have a derivation of the same shape as in the previous case. However, by $M' \notin \mathsf{GVal}$ we have $B = \delta(B')$ and $[\![M : a]\!] = \mathbf{let}_\bullet\ x = \sqcap b.[\![M' : b]\!]\ \mathbf{in}\ [\![N' : a]\!]$, for some δ such that $\Delta \vdash \delta : \tilde{a} \Rightarrow_\bullet \cdot$.
We define Let Δ_o and δ_m as in the previous case and extend δ to δ' by setting $\delta'(a) = c$. This implies $B = \delta(B') = \delta'(B') = \delta(\delta_\mathrm{m}(b))$.

We also extend δ to δ'' by setting $\delta''(a') = a'$ for all $a' \in \Delta$.
Using similar as in the previous case we get $\mathsf{mostgen}(\Delta, (a,b), \Gamma, [\![M' : b]\!], (\tilde{a}, c), \delta_{\mathrm{m}})$, $\Delta; a; (\Gamma, x : B); [a \mapsto A] \vdash [\![N' : a]\!]$, and $(\Delta, \tilde{a}); a; \Gamma; [a \mapsto A, b \mapsto B'] \vdash [\![M' : b]\!]$. Applying Lemma 19 to the latter using δ'' yields $\Delta; (a,b); \delta''(\Gamma); [a \mapsto \delta''(A), b \mapsto \delta''(B')] \vdash [\![M' : b]\!]$, which is equivalent to $\Delta; (a,b); \Gamma; [a \mapsto A, b \mapsto B] \vdash [\![M' : b]\!]$,
We can then derive the following

$$\frac{\begin{array}{c}\mathsf{mostgen}(\Delta, (a,b), \Gamma, [\![M' : b]\!], (\tilde{a}, c), \delta_{\mathrm{m}}) \\ \Delta \vdash \delta' : (\tilde{a}, c) \Rightarrow_\bullet \cdot \qquad B = \delta'(\delta_{\mathrm{m}}(a)) \\ \Delta; (a,b); \Gamma; \delta[b \mapsto B] \vdash [\![M' : b]\!] \qquad \Delta; a; (\Gamma, x : B); \delta \vdash [\![N' : a]\!]\end{array}}{\Delta; a; \Gamma; [a \mapsto A] \vdash \mathbf{let}_\bullet\ x = \sqcap b.[\![M' : b]\!]\ \mathbf{in}\ [\![N' : a]\!]}$$

Case $\mathbf{let}(x : B) = M'\ \mathbf{in}\ N'$**, where** $M' \in \mathsf{GVal}$**:** Let $\overline{a}, H$ such that $B = \forall \overline{a}.H$.
The derivation of $\Delta; \Gamma \vdash M' : A$ has the following form, for some Δ', A'

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

By $M' \in \mathsf{GVal}$, we have $\Delta' = \tilde{a}$, $B' = H$ and $[\![M : A]\!] = (\forall \overline{a}.[\![M' : H]\!]) \wedge \mathbf{def}\ (x : B)\ \mathbf{in}\ [\![N' : A]\!]$. Let b be fresh. By induction, we have $(\Delta, \Delta'); b; \Gamma; [b \mapsto B'] \vdash [\![M' : b]\!]$ and $\Delta; a; (\Gamma, x : B); [a \mapsto A] \vdash [\![N' : A]\!]$. By Lemma 20 we can substitute the former to $(\Delta, \Delta'); \cdot; \Gamma; \emptyset \vdash [\![M' : H]\!]$.
Recall that an implicit precondition of $\Delta; (\Gamma, x : B) \vdash N' : A$ we have $\Delta \vdash \Gamma\ \mathbf{ok}$, which implies $\Delta \vdash B\ \mathbf{ok}$. Therefore, we have that $\mathsf{ftv}(B) - \Delta$ is empty and $B[a/A] = B$.
We can now derive the desired judgement.

$$\frac{\dfrac{\dfrac{(\Delta, \Delta'); a; \Gamma; [a \mapsto A] \vdash [\![M' : H]\!]}{\vdots}}{\Delta; a; \Gamma; [a \mapsto A] \vdash (\forall \overline{a}.[\![M' : H]\!])} \qquad \dfrac{\begin{array}{c}\text{for all } a \in \mathsf{ftv}(B) - \Delta \mid \Delta; \Xi; \Gamma; \delta \vdash \mathsf{mono}(a) \\ \Delta; \Xi; (\Gamma, x : B[a/A]); \delta \vdash C\end{array}}{\Delta; a; \Gamma; [a \mapsto A] \vdash \mathbf{def}\ (x : B)\ \mathbf{in}\ [\![N' : A]\!]}}{\Delta; a; \Gamma; [a \mapsto A] \vdash (\forall \overline{a}.[\![M' : H]\!]) \wedge \mathbf{def}\ (x : B)\ \mathbf{in}\ [\![N' : A]\!]}$$

Case $\mathbf{let}(x : A) = M'\ \mathbf{in}\ N'$**, where** $M' \notin \mathsf{GVal}$**:** Analogous to previous case, with $\Delta' = \emptyset$, $B' = B$.

□

A.3 Proof of Theorem 2

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* $\Delta; \Gamma \vdash M : \delta(a)$.

Proof. We prove the following, slightly more general property by induction on M: If $\Delta; \Gamma \vdash M\ \mathbf{ok}$ and $\Delta; \Xi \vdash A\ \mathbf{ok}$ and $\Delta; \Xi; \Gamma; \delta \vdash [\![M : A]\!]$, then $\Delta; \Gamma \vdash M : \delta(A)$.

Note that by the implicit preconditions of $\Delta; \Xi; \Gamma; \delta \vdash [\![M : A]\!]$ we have $\Delta \# \Xi$, $\Delta \vdash \Gamma\ \mathbf{ok}$, and $\Delta \vdash \delta : \Xi \Rightarrow_\star \cdot$. The latter implies $\Delta \vdash \delta(A)\ \mathbf{ok}$.

We focus on the let cases.

Case *Let* $x = M'$ **in** N', $M' \in \mathrm{GVal}$**:** We have $\Delta; \Xi; \Gamma; \delta \vdash \mathbf{let}_\star\ x = \sqcap b.[\![M' : b]\!]$ **in** $[\![N' : A]\!]$. The derivation of this has the following form for some $B, \overline{a}, \Delta_\mathrm{m}, \Delta_\mathrm{o}, \delta_\mathrm{m}$ and δ':

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

By induction, this gives us $(\Delta, \tilde{a}); \Gamma \vdash M' : B$ and $\Delta; (\Gamma, x : \forall \overline{a}.B) \vdash N' : \delta(A)$. By $M' \in \mathrm{GVal}$, we have $\Delta \vdash M'$ and $\Delta \vdash \Gamma$. According to Lemma 11 this implies $\Delta, (a), \Gamma \vdash [\![M' : a]\!]$ **ok** (i.e., a is the only free flexible variable of $[\![M' : a]\!]$). This means that $[\![M' : a]\!]$ leaves all variables of Ξ entirely unconstrained. Therefore, we have that δ_m maps all $a \in \Xi$ to pairwise different variables c from Δ_m and $c \notin \mathsf{ftv}(\delta_(a))$. This implies $\overline{a} = \mathsf{ftv}(\delta_\mathrm{m}(b)) - \Delta$ (i.e., removing Δ_o has no effect) and $B = \delta_\mathrm{m}(b)$ (i.e., applying δ' to $\delta_\mathrm{m}(b)$ has not effect).
Consequently, we may strengthen $\mathsf{mostgen}(\Delta, (\Xi, b), \Gamma, [\![M' : b]\!], \Delta_\mathrm{m}, \delta_\mathrm{m})$ to $\mathsf{mostgen}(\Delta, b, \Gamma, [\![M' : b]\!], \tilde{a}, [b \mapsto \delta_\mathrm{m}(b)])$. We may then apply Lemma 9 to the latter, yielding $\mathsf{principal}(\Delta, \Gamma, M', \tilde{a}, B)$.
Finally, Lemma 10 shows us that B is a guarded type, and we may refer to it as H.
We can therefore derive the desired property $\Delta; \Gamma \vdash \mathbf{let}\ x = M'\ \mathbf{in}\ N' : A$ as follows:

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

Case *Let* $x = M'$ **in** N' **if** $M' \notin \mathrm{GVal}$**:** This case is largely analogous to the previous one.
This time we have a derivation of the form:

$$\frac{\begin{array}{c}\mathsf{mostgen}(\Delta, (\Xi, a), \Gamma, [\![M' : b]\!], \Delta_\mathrm{m}, \delta_\mathrm{m}) \\ \Delta \vdash \delta' : \Delta_\mathrm{m} \Rightarrow_\bullet \cdot \qquad B = \delta'(\delta_\mathrm{m}(b)) \\ \Delta; (\Xi, a); \Gamma; \delta[b \mapsto B] \vdash [\![M' : b]\!] \qquad \Delta; \Xi; (\Gamma, x : B); \delta \vdash [\![N' : A]\!]\end{array}}{\Delta; \Theta; \Gamma; \delta \vdash \mathbf{let}_\bullet\ x = \sqcap b.[\![M' : b]\!]\ \mathbf{in}\ [\![N' : A]\!]}$$

Let $A' \coloneqq \delta_\mathrm{m}(b)$ and $\overline{a} \coloneqq \mathsf{ftv}(A') - \Delta$. By $\mathsf{mostgen}(\Delta, (\Xi, a), \Gamma, [\![M' : b]\!], \Delta_\mathrm{m}, \delta_\mathrm{m})$ we have $(\Delta, \Delta_\mathrm{m}); (\Xi, a); \Gamma; \delta_\mathrm{m} \vdash [\![M' : b]\!]$. By induction, this implies $(\Delta, \Delta_\mathrm{m}); \Gamma \vdash M' : A'$, which we can strengthen to $(\Delta, \tilde{a}); \Gamma \vdash M' : A'$
We obtain $\Delta; (\Gamma, x : B) \vdash \delta(A)$ directly by applying the induction hypothesis to $\Delta; \Xi; (\Gamma, x : B); \delta \vdash [\![N' : A]\!]$. Likewise, we obtain $\mathsf{principal}(\Delta, \Gamma, M', \tilde{a}, A')$ using the same reasoning as in the previous case.
Finally, we observe that $\Delta \vdash \delta'\restriction_{\tilde{a}} \Rightarrow \cdot$ holds and $B = \delta'\restriction_{\tilde{a}}(A')$. Therefore, we have $(\Delta, \overline{a}, M, A')\ \Updownarrow\ B$

Thus, we can derive the following:

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

Case *Let* $(x : B) = M'$ **in** N' **if** $M' \in \mathrm{GVal}$**:** Let $\overline{b}$ and H such that $B = \forall \overline{b}.H$.
We then have $\Delta; \Xi; \Gamma; \delta \vdash (\forall \overline{b}.[\![M' : H]\!]) \wedge \mathbf{def}\ (x : B)\ \mathbf{in}\ [\![N' : A]\!]$. The derivation tree of this contains derivations for $(\Delta, \tilde{b}); \Xi; \Gamma; \delta \vdash [\![M' : H]\!]$ and $\Delta; \Xi; (\Gamma, x : \delta(B)); \delta \vdash [\![N' : A]\!]$.

By $\Delta; Gamma \vdash M$ **ok** we have $\Delta \vdash B$ (i.e., B contains no flexible variables). Therefore, $\delta(B) = B$ and $\delta(H) = H$. By induction, this then gives us $(\Delta, \tilde{a}); \Gamma \vdash M' : H$ and $\Delta; (\Gamma, x : B) \vdash N' : \delta(A)$.
Hence, we can derive

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

Case *Let* $(x : B) = M'$ **in** N' **if** $M' \notin \mathrm{GVal}$ This case is similar to the previous one: We have $\Delta; \Xi; \Gamma; \delta \vdash [\![M' : B]\!] \ \wedge\ \mathbf{def}\ (x : B)\ \mathbf{in}\ [\![N' : A]\!]$ this time and $\delta(B) = B$ due to $\Delta; \Gamma \vdash M$ **ok**.
We get $\Delta; \Gamma \vdash M' : B$ and $\Delta; (\Gamma, x : B) \vdash N' : \delta(A)$ by induction and can derive

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

□

B PROOFS FOR SECTION 4.2

We again proceed by collection auxiliary lemmas in Appendix B.1 before proofing each theorem from Section 4.2 in an individual subsection.

B.1 Auxiliary Lemmas

Lemma 13 (Well-Ordering on States)**.** *There exists a strict well-ordering* $<$ *on the set* St *of stack machine states such that for all* $s, s' \in \mathrm{St}$ *with* $s \longrightarrow s'$ *we have* $s' < s$.

Proof. First, we define the size of a constraint C, denoted $|C|$, s.t.

$$\begin{aligned}
|\mathsf{true}| &= 0 \\
|\mathsf{mono}(a)| &= 1 \\
|A \sim B| &= 1 \\
|\lceil x : A \rceil| &= 2 \\
|x \preceq A| &= 2 \\
|\exists a.C| &= 1 + |C| \\
|\forall a.C| &= 1 + |C| \\
|\mathbf{def}\ (x : A)\ \mathbf{in}\ C| &= 1 + |C| \\
|C_1 \wedge C_2| &= 1 + |C_1| + |C_2| \\
|\mathbf{let}_R\ x = \sqcap a.C_1\ \mathbf{in}\ C_2| &= 3 + |C_1| + |C_2|
\end{aligned}$$

Next, we define $\mathsf{insts}(C)$ to be the number of instantiation (sub-)constraints in C.
We now define the function $|\cdot|$ that maps states to elements of $\mathbb{N}_0 \times \mathbb{N}_0 \times \mathbb{N}_0 \times \mathbb{N}_0$:

$$|(f_0 :: \cdots :: f_n, \Theta, \theta, C)| = (\mathsf{insts}(C),\ |F[C]|,\ |C|,\ \max_{0 \le i \le n}\ f_i \text{ is an } \exists \text{ frame })$$

We observe that the lexicographic ordering $<_{lex}$ on tuples from $\mathbb{N}_0 \times \mathbb{N}_0 \times \mathbb{N}_0 \times \mathbb{N}_0$ constitutes a well-ordering on such tuples and we will show below that for each step $s \longrightarrow s'$ we have that $|s'| <_{lex} |s|$ holds. However, the function $|\cdot|$ on states is surjective, which implies that defining $s' < s$ iff $|s'| <_{lex} |s|$ would *not* yield a total order on states. Hence, let $<_{ord}$ be some arbitrary strict well-order on states. We then define

$$s' < s \ \text{ iff } \ |s'| <_{lex} |s| \text{ or } |s'| = |s| \text{ and } s' <_{ord} s$$

which is indeed a well-ordering.

It remains to show that each step of the stack machine produces a smaller state w.r.t. $|\cdot|$. Hence, assume $s \longrightarrow s'$, where $s = (F, \Theta, \theta, C)$ and $s' = (F', \Theta', \theta', C')$.

- If the step is the result of applying the rule S-Eq, S-Freeze, or S-Mono, then we have $F = F'$ and $|C'| < |C|$, yielding $|s'| <_{lex} |s|$ via the second component of the tuples.
- If the step is the result of applying S-Inst, we have $\mathsf{insts}(C') = \mathsf{insts}(C) - 1$ and we have $|s'| <_{lex} |s|$ via the first component of the tuples.
- If the step is the result of applying the rule S-ConjPop, S-ForallPop, or S-DefPop, we have $\mathsf{insts}(C) = \mathsf{insts}(C')$ and $|F'[C']| < |F[C]|$, yielding $|s'| <_{lex} |s|$ via the second component of the tuple.
- If the step is the result of applying the rule S-ConjPush, S-ExistsPush, S-ForallPush, S-DefPush, or S-LetPush, we have $\mathsf{insts}(C) = \mathsf{insts}(C')$ and $F[C] = F'[C']$, but $|C'| < |C|$, yielding $|s'| <_{lex} |s|$ via the third component of the tuples.
- If S-ExistsLower got applied, let $\tilde{c}$ and $\tilde{a}$ be defined as in the rule, $F = f_0 :: \ldots f_n$, and $l = |\tilde{a}|$. Note that the rule imposes $l > 0$ and we have $C = C' = \mathsf{true}$.
  We observe that $\tilde{c}$ is a subset of $\tilde{a}$. If $|\tilde{a}| > |\tilde{c}|$, we have $|F'[C']| < |F[C]|$ because the set of frames of F' is a strict subset of the frames in F. We then obtain $|s'| <_{lex} |s|$ immediately via the second component of the tuples (the first component remains unchanged). Otherwise, we have that the two sets are equal. In that case we have $|F[C]| = |F'[C']|$ (as there is merely a reordering of stack frames happening) and $|C| = |C'| = 0$. The resulting stack F' is of the form $f'_0 :: \ldots f'_n$, where $f'_n = f_{n-l}$ (not an $\exists$ frame), and f'_{n-1} is an $\exists$ frame.
  Together, we have $|s| = (\mathsf{insts}(C), |F[C]|, 0, n)$, and $|s'| = (\mathsf{insts}(C'), |F[C]|, 0, n-1)$, and $\mathsf{insts}(C) = \mathsf{insts}(C')$, yielding $|s'| <_{lex} |s|$.
- If rule S-LetPolyPop was applied, we use the following reasoning: F is of the form $F_0 :: \mathsf{let}_\star\ x = \sqcap c.\square\ \mathsf{in}\ \hat{C} :: \exists \tilde{a}$ and C is true. This yields

$$\begin{aligned} |F[C]| &= |F_0[\mathbf{let}_\star\ x = \sqcap c.\exists \tilde{a}.\mathsf{true}\ \mathbf{in}\ \hat{C}]| \\ &= |\mathbf{let}_\star\ x = \sqcap c.\mathsf{true}\ \mathbf{in}\ \mathsf{true}| + |\exists \tilde{a}.\mathsf{true}| + |\hat{C}| + |F_0[\mathsf{true}]| \\ &= 3 + |\tilde{a}| + |\hat{C}| + |F_0[\mathsf{true}]|, \end{aligned}$$

  Let $\tilde{a}''$ be defined as in the rule. We have

$$|F'[C']| = |F_0[\exists \tilde{a}''.\mathbf{def}\ (x : A)\ \mathbf{in}\ \hat{C}] = 1 + |\tilde{a}''| + |\hat{C}| + |F_0[\mathsf{true}]|$$

  Proving $F'[C'] < F[C]$ is therefore equivalent to proving

$$\begin{aligned} & 1 + |\tilde{a}''| + |\hat{C}| + |F_0[\mathsf{true}]| < 3 + |\tilde{a}| + |\hat{C}| + |F_0[\mathsf{true}]| \\ \text{equiv.}\quad & \tilde{a}'' < 2 + \tilde{a}. \end{aligned}$$

  We observe that $\tilde{a}''$ is a strict subset of $(\tilde{a}, c)$ and hence $|\tilde{a}''| < |\tilde{a}| + 2$, meaning that the inequality above holds.
  We have $\mathsf{insts}(C) = \mathsf{insts}(C')$, and therefore $|s'| <_{lex} |s|$ via the second component of the tuples.
- The reasoning for rule S-LetMonoPop is analogous to the previous case. The only change is that we need to observe that $(\tilde{c}, \tilde{a}'')$ is a subset of $(\tilde{a}, c)$.

□

The following lemma states how solutions for constraints $\mathfrak{U}(\Theta, \theta)$ look like. The lemma is somewhat specialised for the specific places where we use it, by introducing an extra type context Δ' and an existential quantifier around $\mathfrak{U}(\Theta, \theta)$.

Lemma 14. *Let* $\Delta \vdash \theta : \Theta \Rightarrow \Theta$ *and* $\Delta, \Delta' \vdash \Gamma$ **ok** *and* $\mathsf{ftv}(\Theta) = \Xi, \tilde{a}$.

Then we have

$$(\Delta, \Delta'); \Xi; \Gamma; \delta \vdash \exists \tilde{a}.\mathfrak{U}(\Theta, \theta)$$
$$\textit{iff}$$
$$\textit{there exists } \theta' \textit{ such that } \Delta, \Delta' \vdash \theta' : \Theta \Rightarrow \cdot \textit{ and } \delta = (\theta' \circ \theta)\restriction_{\Xi}.$$

Proof. By definition, the subconstraint $\mathfrak{U}(\theta)$ of $\mathfrak{U}(\Theta, \theta)$ contains constraints of the form $a \sim \theta(a)$ for all a in Θ. These constraints are satisfied by exactly those substitutions δ refining θ. In addition, the $\bullet$ subconstraints in $\mathfrak{U}(\Theta)$ are satisfied iff the refinement respects the restrictions in Θ. □

The next lemma states how *most general* solutions of constraints $\mathfrak{U}(\Theta, \theta)$ look like.

Lemma 15. *Let the following conditions hold:*

- $\Delta \vdash \theta : \Theta \Rightarrow \Theta$
- $\mathsf{ftv}(\Theta) = \Xi, \tilde{a}$
- $\Delta \vdash \Gamma$ **ok**
- $\overline{b} \approx \mathsf{ftv}(\theta) - \Delta$

Then we have

$$\mathsf{mostgen}((\Delta, \Delta'), \Xi, \Gamma, \exists \tilde{a}.\mathfrak{U}(\Theta, \theta), \Delta_{\mathrm{m}}, \delta_{\mathrm{m}})$$
$$\textit{iff}$$
$$\textit{there exists } \overline{c} \subseteq \Delta_{\mathrm{m}} \textit{ s.t. } \delta_{\mathrm{m}} = ([\overline{b} \mapsto \overline{c}] \circ \theta)\restriction_{\Xi}$$

Proof. Follows directly from Lemma 14 and the observation that the most general solution of $\mathfrak{U}(\Theta, \theta)$ is the one that maps all flexible variables in $\mathsf{ftv}(\theta)$ to fresh, pairwise disjoint rigid variables. Observe that by assumption $\Delta \vdash \theta : \Theta \Rightarrow \Theta$ and the fact that rigid variables are considered monomorphic we have $\Delta \vdash [\overline{b} \mapsto \overline{c}] \circ \theta : \Theta \Rightarrow \cdot$ as well.

□

Lemma 16. *Let C_1, C_2, Δ, Ξ, δ and F be given and let the following condition hold: for all Δ', δ' we have $(\Delta, \mathsf{rc}(F), \Delta'); (\Xi, \mathsf{fc}(F)); \delta'(\mathsf{tc}(F)); \delta' \vdash C_1$ iff $(\Delta, \mathsf{rc}(F), \Delta'); (\Xi, \mathsf{fc}(F)); \delta'(\mathsf{tc}(F)); \delta' \vdash C_2$.*
Then we have have $\Delta; \Xi; \cdot; \delta \vdash F[C_1]$ iff $\Delta; \Xi; \cdot; \delta \vdash F[C_2]$.

Proof. By induction on structure of F, observing that any judgement involving a constraint with subconstraint C_1 can be replaced by a corresponding judgement involving C_2 (and vice versa). □

Lemma 17 (Stack machine steps preserve well-formedness of states)**.** *If s* **ok** *and $s \longrightarrow s'$ then s'* **ok**.

Proof. By case analysis over which stack machine rule was applied. □

Lemma 18 (Weakening mostgen)**.** *Let* $\mathsf{mostgen}(\Delta, \Xi, \Gamma, C, \Delta_{\mathrm{m}}, \delta_{\mathrm{m}})$ *and* $(\Delta, \Xi, \Delta_{\mathrm{m}})\ \#(\overline{a}, \overline{b})$*. Then we have* $\mathsf{mostgen}(\Delta, (\Xi, \tilde{a}), \Gamma, C, (\Delta_{\mathrm{m}}, \overline{b}), \delta_{\mathrm{m}}[\overline{a} \mapsto \overline{b}])$.

Proof. By $\mathsf{mostgen}(\Delta, \Xi, \Gamma, C, \Delta_{\mathrm{m}}, \delta_{\mathrm{m}})$ we have $(\Delta, \Delta_{\mathrm{m}}); \Xi; \Gamma; \delta_{\mathrm{m}} \vdash C$ and therefore $(\Delta, \Delta_{\mathrm{m}}); \Xi; \Gamma \vdash C$ **ok** (cf. Lemma 12). This means that none of the variables in $\overline{a}$ are constrained in any way by C, meaning that the most general solution maps them to pairwise disjoint variables, like $\overline{b}$. □

Lemma 19 (Refinement)**.** *Let $\Xi \,\#\, \Delta'$. If $\Delta; \Xi; \Gamma; \delta \vdash C$ and $\Delta' \vdash \delta' : \Delta \Rightarrow_{\bullet} \cdot$ then $\Delta'; \Xi; \delta'(\Gamma); \delta' \circ \delta \vdash C$.*

Proof. By structural induction on C, observing that for all $a \in \Xi, R \in \{\bullet, \star\}$ we have $\Delta \vdash_R \delta(a)$ **ok** iff $\Delta' \vdash_R \delta'(\delta(a))$ **ok**. □

Lemma 20 (Substitution)**.** *If $\Delta; (\Xi, a); \Gamma; \delta[a \mapsto A] \vdash C$ then $\Delta; \Xi; \Gamma; \delta \vdash C[a/A]$.*

Proof. By structural induction on C. Observe that the substitution does not interfere with generalisation: The variable a is already in scope and therefore not subject to generalisation by any let constraint within C. □

B.2 Proof of Theorem 3

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

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

Proof. We carry out the proof by case analysis of the stack machine reduction rules. Let s be the state before, and s' the state after the step.

We focus on the let rules.

- Case S-LetPolyPop: We have that s is of the form $(\forall\Delta :: \exists\Xi :: F :: \mathbf{let}_\star\ x = \sqcap a.\square\ \mathbf{in}\ C :: \exists\tilde{a}, \Theta_0, \theta_0, \mathsf{true})$ for some F, C, and $\tilde{a}$ and assume that the following conditions imposed by S-LetPolyPop hold:

$$\begin{aligned}
&\tilde{a'};\ \tilde{a''} = \mathsf{split}((\tilde{b}, a), \theta_0, \Theta_0)\\
&A = \theta_0(a)\\
&\overline{c}\ = \mathsf{ftv}(A) \cap \tilde{a'}\\
&B = \forall\overline{c}.A\\
&\Theta_1 = \Theta_0 - \tilde{a'}\\
&\theta_1 = \theta_0 \restriction_{\Theta_1}
\end{aligned}$$

  We need to show

$$\begin{gathered}
\Delta; \Xi; \cdot; \delta \vdash F :: \mathbf{let}_\star\ x = \sqcap a.\square\ \mathbf{in}\ C :: \exists\tilde{a}[\mathsf{true} \wedge \mathfrak{U}(\Theta_0, \theta_0)]\\
\text{iff}\\
\Delta; \Xi; \cdot; \delta \vdash F :: \exists\tilde{a''}[\mathbf{def}\ (x : B)\ \mathbf{in}\ C \wedge \mathfrak{U}(\Theta_1, \theta_1)],
\end{gathered}$$

  which is equivalent to

$$\begin{gathered}
\Delta; \Xi; \cdot; \delta \vdash F[C_0']\\
\text{iff}\\
\Delta; \Xi; \cdot; \delta \vdash F[C_1']
\end{gathered}$$

  if we define $C_0' := \mathbf{let}_\star\ x = \sqcap a.\exists\tilde{a}.\mathfrak{U}(\Theta_0, \theta_0)\ \mathbf{in}\ C$ and $C_1' := \exists\tilde{a''}.(\mathbf{def}\ (x : B)\ \mathbf{in}\ C) \wedge \mathfrak{U}(\Theta_1, \theta_1)$.

  We can prove this equivalence directly using Lemma 16. In order to apply this lemma, we need to show that the following holds: For all $\hat{\Delta}, \hat{\delta}$ we have $(\Delta, \mathsf{rc}(F), \hat{\Delta}); (\Xi, \mathsf{fc}(F)); \hat{\delta}(\mathsf{tc}(F)); \hat{\delta} \vdash C_0'$ iff $(\Delta, \mathsf{rc}(F), \hat{\Delta}); (\Xi, \mathsf{fc}(F)); \hat{\delta}(\mathsf{tc}(F)); \hat{\delta} \vdash C_1'$

  To this end, suppose $\hat{\Delta}$, and $\hat{\delta}$ are given. Let $\Delta' := \Delta, \mathsf{rc}(F), \hat{\Delta}$ and $\Xi' := \Xi, \mathsf{fc}(F)$ and $\Gamma' := \hat{\delta}(\mathsf{tc}(F))$.

  $\Longrightarrow$: We assume $\Delta'; \Xi'; \Gamma'; \hat{\delta} \vdash C_1'$ **(1)**. The derivation of this must have the following form, for some $\Delta_\mathrm{m}, \delta_\mathrm{m}, \Delta_\mathrm{o}, \overline{b}, \delta', A'$:

$$\frac{\begin{gathered}
\mathsf{mostgen}(\Delta', (\Xi', a), \Gamma', \exists\tilde{a}.\mathfrak{U}(\Theta_0, \theta_0), \Delta_\mathrm{m}, \delta_\mathrm{m})\\
\Delta_\mathrm{o} = \mathsf{ftv}(\delta_\mathrm{m}(\Xi')) - \Delta' \qquad \overline{b} = \mathsf{ftv}(\delta_\mathrm{m}(a)) - \Delta', \Delta_\mathrm{o}\\
\Delta' \vdash \delta' : \Delta_\mathrm{o} \Rightarrow_\bullet \cdot \qquad A' = \delta'(\delta_\mathrm{m}(a))\\
(\Delta', \tilde{b}); (\Xi', a); \Gamma'; \hat{\delta}[a \mapsto A'] \vdash \exists\tilde{a}.\mathfrak{U}(\Theta_0, \theta_0) \qquad \Delta'; \Xi'; (\Gamma', x : \forall\overline{b}.A'); \hat{\delta} \vdash C
\end{gathered}}{\Delta'; \Xi'; \Gamma'; \hat{\delta} \vdash \mathbf{let}_\star\ x = \sqcap a.\exists\tilde{a}.\mathfrak{U}(\Theta_0, \theta_0)\ \mathbf{in}\ C}$$

  We can assume w.l.o.g. that the variables in Δ_m are fresh.

  By s **ok** and s' **ok**, we have $\mathsf{ftv}(\Theta_0) = (\Xi, a, \tilde{a})$ as well as idempotency of θ_0 and θ_1.

  Let $\Xi_\mathrm{f} := \mathsf{ftv}(\theta_0) - \Delta'$, which implies $\Xi_\mathrm{f} \subseteq \Theta$. By Lemma 15 there exitsts a bijection δ_r such that $\Delta_\mathrm{m} \vdash \delta_\mathrm{r} : \Xi_\mathrm{f} \Rightarrow_\bullet \cdot$ and $\delta_\mathrm{m} = (\delta_\mathrm{r} \circ \theta_0) \restriction_{(\Xi', a)}$ **(2)**.

We obseve $\delta_{\mathrm{r}}(\tilde{a}'') \subseteq \Delta_{\mathrm{o}}$. We can now define $\hat{\delta}''$ such that for all $b \in (\Xi', \tilde{a}'')$ we have

$$\hat{\delta}''(b) = \begin{cases} \hat{\delta}(b) & \text{if } b \in \Xi' \\ \delta'(\delta_{\mathrm{r}}(b)) & \text{if } b \in \tilde{a}'' \end{cases}$$

which yields $\Delta' \vdash \hat{\delta}'' : (\Xi', \tilde{a}'') \Rightarrow_\star \cdot$ **(3)**.

Next, we show $\delta_{\mathrm{r}}(\overline{c}) = \overline{b}$ **(4)**:

$$\begin{array}{lcl}
\delta_{\mathrm{r}}(\overline{c}) & = & \delta_{\mathrm{r}}(\mathrm{ftv}(\theta_0(a)) \cap \tilde{a}') \\
& = & \delta_{\mathrm{r}}(\mathrm{ftv}(\theta_0(a)) - \Xi' - \tilde{a}'' - \Delta') \\
& & \quad (\text{by } \mathrm{ftv}(\theta_0(a)) \subseteq \Delta', \Xi', \tilde{a}', \tilde{a}'') \\
& = & \delta_{\mathrm{r}}(\mathrm{ftv}(\theta_0(a)) - \mathrm{ftv}(\theta_0(\Xi')) - \tilde{a}'' - \Delta') \\
& & \quad (\text{by } \theta_0(b) = b \text{ for all } b \in \mathrm{ftv}(\theta(a)) \text{ and } \tilde{a}' \# \mathrm{ftv}(\theta(\Xi')) \subseteq \Delta', \Xi', \tilde{a}'') \\
& = & \delta_{\mathrm{r}}(\mathrm{ftv}(\theta_0(a)) - \mathrm{ftv}(\theta_0(\Xi')) - \Delta') \\
& & \quad (\text{by } \tilde{a}'' \subseteq \mathrm{ftv}(\theta_0(\Xi'))) \\
& = & \mathrm{ftv}(\delta_{\mathrm{m}}(a)) - \mathrm{ftv}(\delta_{\mathrm{m}}(\Xi')) - \Delta' \\
& & \quad (\text{by (2)}) \\
& = & \mathrm{ftv}(\delta_{\mathrm{m}}(a)) - \Delta', \Delta_{\mathrm{o}} \\
& = & \overline{b}
\end{array}$$

We now show $\hat{\delta}''(c) = \delta'(\delta_{\mathrm{r}}(c))$ for all $c \in \mathrm{ftv}(\theta_0(a)) \cap (\Xi', \tilde{a}'')$ **(5)**. First, we observe that for all such $c \in \tilde{a}''$ this holds by definition of $\hat{\delta}''$. Next, by $(\Delta', \tilde{b}); (\Xi', a); \Gamma'; \hat{\delta}' \vdash \exists \tilde{a}.\mathfrak{U}(\Theta_0, \theta_0)$ (see derivation of (1)) and Lemma 14 we have that there exists δ_{s} such that $(\Delta', \tilde{b}) \vdash \delta_{\mathrm{s}} : (\Xi', a, \tilde{a}) \Rightarrow_\star \cdot$ and $\hat{\delta}' = (\delta_{\mathrm{s}} \circ \theta_0)\restriction_{(\Xi', a)}$ **(6)**. We have $\theta_0(b) = b$ for all $b \in \mathrm{ftv}(\theta_0)$ and therefore $\delta_{\mathrm{s}}(b) = \hat{\delta}'(b)$ **(7)** for any such b. Using this, we observe

$$\begin{array}{lll}
& \hat{\delta}'(a) & \\
= & \delta_{\mathrm{s}}(\theta_0((a)) & (\text{by (6)}) \\
= & A' & (\text{by def. of } \hat{\delta}') \\
= & \delta'(\delta_{\mathrm{m}}(a)) & (\text{by def. of } A') \\
= & \delta'(\delta_{\mathrm{r}}(\theta_0(a))) & (\text{by (2)})
\end{array}$$

This implies that for all $b \in \mathrm{ftv}(\theta_0(a)) \cap \Xi'$ we have $\delta'(\delta_{\mathrm{r}}(b)) = \delta_{\mathrm{s}}(b) \overset{(7)}{=} \hat{\delta}'(b) = \hat{\delta}''(b)$ and therefore (5) holds.

We now show that $\hat{\delta}''(B)$ is alpha-equivalent to $\forall \tilde{b}.A'$:

$$\begin{array}{lcl}
\hat{\delta}''(B) & = & \hat{\delta}''(\forall \overline{c}.\theta_0(a)) \\
& = & \forall \overline{c}.\hat{\delta}''(\theta_0(a)) \\
& & \quad (\text{due to (3)}, \overline{c} \# \Xi', \tilde{a}'' \text{ and } \overline{c} \# \Delta', \Delta_{\mathrm{m}}) \\
& = & \forall \overline{b}.\left(\hat{\delta}''(\theta_0(a))[\overline{c} \mapsto \overline{b}]\right) \\
& & \quad (\text{due to (4)}, \overline{c} \# \overline{b}) \\
& = & \forall \overline{b}.\delta'(\delta_{\mathrm{r}}(\theta_0(a))) \\
& & \quad (\text{due to (2), (5)}, \hat{\delta}''(b) = b \text{ for all } b \in \overline{b}) \\
& = & \forall \overline{b}.\delta'(\delta_{\mathrm{m}}(a)) \\
& = & \forall \overline{b}.A'
\end{array}$$

This means that by $\Delta'; \Xi'; (\Gamma', x : \forall \overline{b}.A'); \hat{\delta} \vdash C$ (see derivation of (1)) we also have $\Delta'; \Xi'; (\Gamma', x : \hat{\delta}''(B)); \hat{\delta} \vdash C$, which we can weaken to $\Delta'; (\Xi', \tilde{a}''); (\Gamma', x : \hat{\delta}''(B)); \hat{\delta}'' \vdash C$ **(8)**.

We now show that for all $b \in \mathsf{ftv}(B) - \Delta'$ we have $\Delta'; \Xi'; \Gamma'; \hat{\delta}'' \vdash \mathsf{mono}(b)$ **(9)**: We have $\mathsf{ftv}(B) - \Delta' = \mathsf{ftv}(\forall \overline{c}.\theta_0(a)) - \Delta' \subseteq \Xi, \tilde{a}''$. By (5) we have $\hat{\delta}''(b) = \delta'(\delta_\mathrm{r}(b))$ for any such b. By $\Delta' \vdash \delta' : \Delta_\mathrm{o} \Rightarrow_\bullet \cdot$ and δ_r being a bijection on variables, we have $\Delta' \vdash_\bullet \hat{\delta}''(b)$ **ok**.

We now show $\Delta'; (\Xi', \tilde{a}''); \Gamma'; \hat{\delta}'' \vdash \mathfrak{U}(\Theta_1, \theta_1)$ **(10)**: Recall that per (6), we have $\hat{\delta}' = (\delta_\mathrm{s} \circ \theta_0)\restriction_{(\Xi', a)}$, where $\hat{\delta}'$ and $\hat{\delta}''$ coincide on Ξ'. Further, $\theta_0(b) = b$ for all $b \in \tilde{a}''$. We can therefore apply Lemma 14 to $\hat{\delta}''$ to obtain (10).

Using (8), (9), and (10) we can now derive $\Delta'; \Xi'; \Gamma'; \hat{\delta} \vdash C_1'$ as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Delta'; (\Xi', \tilde{a}''); (\Gamma', x : \hat{\delta}''B); \hat{\delta}'' \vdash C}{\text{for all } b \in \mathsf{ftv}(B) - \Delta' \;\mid\; \Delta'; \Xi'; \Gamma'; \hat{\delta}'' \vdash \mathsf{mono}(b)}}{\Delta'; (\Xi', \tilde{a}''); \Gamma'; \hat{\delta}'' \vdash \mathbf{def}\ (x : B)\ \mathbf{in}\ C} \qquad \Delta'; (\Xi', \tilde{a}''); \Gamma'; \hat{\delta}'' \vdash \mathfrak{U}(\Theta_1, \theta_1)}{\Delta'; (\Xi', \tilde{a}''); \Gamma'; \hat{\delta}'' \vdash (\mathbf{def}\ (x : B)\ \mathbf{in}\ C) \wedge \mathfrak{U}(\Theta_1, \theta_1)}}{\vdots}}{\Delta'; \Xi'; \Gamma'; \hat{\delta} \vdash \exists \tilde{a}''.(\mathbf{def}\ (x : B)\ \mathbf{in}\ C) \wedge \mathfrak{U}(\Theta_1, \theta_1)}$$

$\Longleftarrow$: We assume $\Delta'; \Xi'; \Gamma'; \hat{\delta} \vdash C_2'$ **(11)**. The derivation of this must have the following form for some $\hat{\delta}''$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Delta'; (\Xi', \tilde{a''}); (\Gamma', x : \hat{\delta}''B); \hat{\delta}'' \vdash C}}{\text{for all } b \in \mathsf{ftv}(B) - \Delta' \;\mid\; \Delta'; \Xi'; \Gamma'; \hat{\delta}'' \vdash \mathsf{mono}(b)} \qquad \dfrac{\vdots}{\Delta'; (\Xi', \tilde{a}''); \Gamma'; \hat{\delta}'' \vdash \mathfrak{U}(\Theta_1, \theta_1)}}{\Delta'; (\Xi', \tilde{a}''); \Gamma'; \hat{\delta}'' \vdash (\mathbf{def}\ (x : B)\ \mathbf{in}\ C) \wedge \mathfrak{U}(\Theta_1, \theta_1)}}{\vdots}}{\Delta'; \Xi'; \Gamma'; \hat{\delta} \vdash \exists \tilde{a}''.(\mathbf{def}\ (x : B)\ \mathbf{in}\ C) \wedge \mathfrak{U}(\Theta_1, \theta_1)}$$

This immedately gives us $\Delta' \vdash \hat{\delta}'' : (\Xi', \tilde{a}'') \Rightarrow_\star \cdot$ **(12)**.

Let Ξ_f be defined as in the $\Rightarrow$ case and let δ_r be a bijection from Ξ_f to fresh variables. Further, let $\Delta_\mathrm{m} \coloneqq \mathsf{ftv}(\delta_\mathrm{r})$ and $\delta_\mathrm{m} \coloneqq (\delta_\mathrm{r} \circ \theta_0)\restriction_{(\Xi', a)}$. By Lemma 15 we then have $\mathsf{mostgen}(\Delta', (\Xi', a), \Gamma', \exists \tilde{a}.\mathfrak{U}(\Theta_0, \theta_0), \Delta_\mathrm{m}, \delta_\mathrm{m})$. We may now define Δ_o and $\overline{b}$ as in the $\Rightarrow$ case, making each of them a subset of Δ_m.

We now define δ' for all $b \in \Delta_\mathrm{o}$ as follows:

$$\delta'(b) = \begin{cases} \hat{\delta}''(\delta_\mathrm{r}^{-1}(b)) & \text{if } \delta_\mathrm{r}^{-1}(b) \in \mathsf{ftv}(\theta(a)) - \tilde{a}' - \Delta' \\ \mathsf{unit} & \text{otherwise} \end{cases}$$

We have $\tilde{a'} \subseteq \overline{c}$ and therefore $\mathsf{ftv}(B) = \mathsf{ftv}(\forall \tilde{c}.\theta(a)) = \mathsf{ftv}(\theta(a)) - \tilde{a}'$. Thus, by the definition above and the second premise of the derivation of $\Delta'; (\Xi', \tilde{a}''); \Gamma'; \hat{\delta}'' \vdash \mathbf{def}\ (x : B)\ \mathbf{in}\ C$, we have $\Delta' \vdash \delta' : \Delta_\mathrm{o} \Rightarrow_\bullet \cdot$.

The definition of δ' immedately yields $\hat{\delta}''(c) = \delta'(\delta_\mathrm{r}(c))$ for all $c \in \mathsf{ftv}(\theta_0(a)) \cap (\Xi', \tilde{a}'')$. (which we called (5) in the $\Rightarrow$ direction). We define $A' \coloneqq \delta'(\delta_\mathrm{m}(a))$. Together with (12), we

can use the same reasoning as in the $\Rightarrow$ case to obtain $\overline{b} = \delta_{\mathrm{r}}(\overline{c})$ and the alpha-equivalence of and $\hat{\delta}''(B)$ and $\forall \overline{b}.A'$.

Hence, $\Delta'; (\Xi', \tilde{a}''); (\Gamma', x : \hat{\delta}''B); \hat{\delta}'' \vdash C$ (see derivation of (11)) is equivalent to $\Delta'; (\Xi', \tilde{a}''); (\Gamma', x : \forall \overline{b}.A'); \hat{\delta}'' \vdash C$. We have $\Delta' \vdash \forall \overline{b}.A'$ **ok** and $(\Delta, \mathrm{rc}(F)); \Xi'; (\mathrm{tc}(F), x : \bot) \vdash C$ **ok** which means we can weaken it to $\Delta'; \Xi'; (\Gamma', x : \forall \overline{b}.A'); \hat{\delta} \vdash C$.

Next, we define $\hat{\delta}' \coloneqq \hat{\delta}[a \mapsto A']$ as in the $\Rightarrow$ case and show that $(\Delta', \tilde{b}); (\Xi', a); \Gamma'; \hat{\delta}' \vdash \exists \tilde{a}.\mathfrak{U}(\Theta_0, \theta_0)$ **(13)** holds. To this end, we wish to apply Lemma 14 to $\Delta'; (\Xi', \tilde{a}''); \Gamma'; \hat{\delta}'' \vdash \mathfrak{U}(\Theta_1, \theta_1)$ (see derivation of (11)), which gives us the existence of θ_1' such that $\Delta' \vdash \theta_1' : \Theta_1 \Rightarrow \cdot$ and $\hat{\delta}'' = \theta_1' \circ \theta_1$ **(14)**. We now show that the necessary preconditions of the lemma are satisfied.

Recall that $\mathrm{ftv}(\Theta_1) = \mathrm{ftv}(\Theta_0) - \tilde{a}'$ and $\tilde{c} \subseteq \tilde{a}'$. We now define θ_0' as an extension of θ_1' by setting $\theta_0'(c) = \delta_{\mathrm{r}}(c)$ for all $c \in \tilde{c}$, and $\theta_0'(c) = \mathsf{unit}$ for all $c \in \tilde{a}' - \tilde{c}$. This implies $\Delta', \tilde{b} \vdash \theta_0' : \Theta_0 \Rightarrow \cdot$. We now show that $\hat{\delta}' = (\theta_0' \circ \theta_0)\restriction_{(\Xi', a)}$ **(15)**. For all $b \in \Xi'$ we immediately get $\theta_0'(\theta_0(b)) = \hat{\delta}''(b)$ by $\Xi' \subseteq \mathrm{ftv}(\Theta_1)$ and (14).

It remains to show that $\theta_0'(\theta_0(a)) = \hat{\delta}'(a) \overset{\text{def.}}{=} A'$. By definition of A' and δ_{m} we have $A' = \delta'(\delta_{\mathrm{m}}(a)) = \delta'(\delta_{\mathrm{r}}(\theta_0(a))$. Therefore, it sufficies to show that for all $b \in \mathrm{ftv}(\theta_0(a)) - \Delta'$ we have $\theta_0'(b) = \delta'(\delta_{\mathrm{r}}(b))$. If $b \in (\Xi', \tilde{a}'')$ we have $\theta_0'(b) = \theta_1'(b)$. By $b \in \mathrm{ftv}(\theta_0)$ we have $\theta_0(b) = b$, which means that (14) imposes $\theta_1'(b) = \hat{\delta}''(b)$. By definition of δ' we then have $\theta_0'(b) = \theta_1'(b) = \hat{\delta}''(b) = \hat{\delta}''(\delta_{\mathrm{r}}^{-1}(\delta_{\mathrm{r}}(b))) = \delta'(\delta_{\mathrm{r}}(b))$. Otherwise, if $b \in \tilde{a}'$, we have $\delta_{\mathrm{r}}(b) \notin \Delta_{\mathrm{o}}$ and therefore $\theta_0'(b) = \delta_{\mathrm{r}}(b) = \delta'(\delta_{\mathrm{r}}(b))$. Finally, having shown (15) we may apply Lemma 14 to obtain (13).

In conclusion, we can now derive $\Delta'; \Xi'; \Gamma'; \hat{\delta} \vdash C_1'$ as follows:

$$\frac{\begin{array}{c}\mathsf{mostgen}(\Delta', (\Xi', a), \Gamma', \exists \tilde{a}.\mathfrak{U}(\Theta_0, \theta_0), \Delta_{\mathrm{m}}, \delta_{\mathrm{m}}) \\ \Delta_{\mathrm{o}} = \mathrm{ftv}(\delta_{\mathrm{m}}(\Xi')) - \Delta' \qquad \overline{b} = \mathrm{ftv}(\delta_{\mathrm{m}}(a)) - \Delta', \Delta_{\mathrm{o}} \\ \Delta' \vdash \delta' : \Delta_{\mathrm{o}} \Rightarrow_{\bullet} \cdot \qquad A' = \delta'(\delta_{\mathrm{m}}(a)) \\ (\Delta', \tilde{b}); (\Xi', a); \Gamma'; \hat{\delta}[a \mapsto A'] \vdash \exists \tilde{a}.\mathfrak{U}(\Theta_0, \theta_0) \qquad \Delta'; \Xi'; (\Gamma', x : \forall \overline{b}.A'); \hat{\delta} \vdash C\end{array}}{\Delta'; \Xi'; \Gamma'; \hat{\delta} \vdash \mathbf{let}_{\star}\ x = \sqcap a.\exists \tilde{a}.\mathfrak{U}(\Theta_0, \theta_0)\ \mathbf{in}\ C}$$

□

B.3 Proof of Theorem 4

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* $(F, \Theta, \Gamma, \theta, C) \longrightarrow s_1$.

Proof. Let $s \coloneqq (F, \Theta, \theta, C)$. By assumption, we have $\cdot; \cdot; \cdot; \emptyset \vdash F[C \wedge \mathfrak{U}(\Theta, \theta)]$ **(1)**.
We first assume $C \neq \mathsf{true}$ and show that one of the rules in Figure 9 is applicable.

- Case $C_1 \wedge C_2$: Rule S-ConjPush is applicable.
- Case $A \sim B$: The left-hand-side of S-Eq matches. The derivation of (1) must contain a subderivation of the form
$$\frac{\delta'(A) = \delta'(B)}{\Delta'; \Xi'; \Gamma'; \delta' \vdash A \sim B}$$
for some $\Delta', \Xi', \Gamma', \delta'$.
Since we are using the same unification algorithm as in [Emrich et al. 2020], we can then use [Emrich et al. 2020, Theorem 5] to show that unification succeeds.
- Case $\lceil x : A \rceil$: The left-hand-side of S-Inst matches. By (1) we have $x \in \mathrm{tc}(F)$, meaning that the rule succeeds.

- Case $x \preceq A$: Rule S-Freeze is applicable (using the same argument to show $x \in \mathrm{tc}(F)$ as in the previous case).
- Case $\exists a.C'$: Rule S-ExistsPush is applicable.
- Case $\forall a.C'$: Rule S-ForallPush is applicable.
- Case $\mathbf{def}\ (x : A)\ \mathbf{in}\ C'$: The left-hand-side of S-DefPush matches.
  The derivation of (1) must contain a sub-derivation of the form

$$\frac{\begin{array}{c}(\text{for all } a \in \mathrm{ftv}(A) - (\mathrm{rc}(F), \Delta') \mid (\mathrm{rc}(F), \Delta'); \mathrm{fc}(F); \delta(\mathrm{tc}(F)); \delta \vdash \mathrm{mono}(a) \\ (\mathrm{rc}(F), \Delta'); \mathrm{fc}(F); (\delta(\mathrm{tc}(F)), x : \delta A); \delta \vdash C'\end{array}}{(\mathrm{rc}(F), \Delta'); \mathrm{fc}(F); \delta(\mathrm{tc}(F)); \delta \vdash \mathbf{def}\ (x : A)\ \mathbf{in}\ C'}$$

  for some δ, where $C' = C \wedge \mathfrak{U}(\Theta, \theta)$ and $\mathrm{ftv}(\Theta) = \Xi$. Note that by s **ok** we have $\mathrm{ftv}(A) \subseteq (\mathrm{rc}(F), \mathrm{fc}(F))$.
  According to Lemma 14, we have $\delta = \theta' \circ \theta$ for some θ' with $(\mathrm{rc}(F), \Delta') \vdash \theta' : \Theta \Rightarrow \cdot$. Therefore, the monomorphism conditions imposed by S-DefPush are satisfied.
- Case $\mathrm{mono}(a)$: Analogous to def case; the sub-derivation for $\mathrm{mono}(a)$ implies that the monomorphism conditions imposed by S-Mono are satisifed, making the rule applicable.
- Case $\mathbf{let}_R\ x = \sqcap a.C_1\ \mathbf{in}\ C_2$: Rule S-LetPush is applicable.

We now conside the case that C is true. Due the assumption about the shape of $F[C]$, we know that F is neither emtpy nor of the shape $\forall\Delta :: \exists\overline{a}$ for any $\overline{a}$.

We perform a case analysis on the topmost stack frame of F:

- Case $\square \wedge C_2$: Rule S-ConjPop is applicable.
- Case $\mathbf{def}\ (x : A)$: Rule S-DefPop is applicable.
- Case $\mathbf{let}_R\ x = \sqcap a.C_1\ \mathbf{in}\ C_2$: Rule S-LetPolyPop or S-LetMonoPop is applicable, where the sequence $\overline{a}$ mentioned in the rule's definition is empty. None of the respective rule's side conditions can fail.
- Case $\forall a$: Let F' be defined such that $F = F' :: \forall a$. By assumption s **ok** we have that the variables in Θ are exactly the variables bound by $\exists$ or **let** frames in F'. Suppose there exists $b \in \mathrm{ftv}(\Theta)$ such that $a \in \mathrm{ftv}(\theta(b))$, which would cause the rule to fail. Then there exists a frame f in F' that binds b. Let F_p be the (possibly empty) prefix of F' up to, but not including, frame f and let F_s be the (possibly empty) suffix from there (i.e., $F' = F_p :: f :: F_s$).
  We distinguish two sub-cases further:
  (1) If $f = \exists b$ then the derivation of (1) must contain a sub-derivation of the following form:

$$\frac{(\mathrm{rc}(F_p), \Delta'); (\mathrm{fc}(F_p), b); \delta(\mathrm{tc}(F_p)); \delta[b \mapsto A] \vdash C'}{(\mathrm{rc}(F_p), \Delta'); \mathrm{fc}(F_p); \delta(\mathrm{tc}(F_p)); \delta \vdash \exists b.C'}$$

  for some A, δ, and Δ', where $C' = F_s[C \wedge \mathfrak{U}(\Theta, \theta)]$.
  Note that $(\mathrm{rc}(F_p), \Delta'); (\mathrm{fc}(F_p), b); \delta(\mathrm{tc}(F_p)); \delta[b \mapsto A] \vdash C'$ implies $(\mathrm{rc}(F_p), \Delta') \vdash A$ **ok (2)**. Further, as F_s contains the frame $\forall a$, we have $a \notin \mathrm{rc}(F_p), \Delta'$ **(3)**.
  Likewise, there existis a subderivation showing $(\mathrm{rc}(F), \Delta''); \mathrm{fc}(F); \delta''(\mathrm{tc}(F)); \delta'' \vdash \mathfrak{U}(\Theta, \theta)$ for some Δ'' and δ'', where $\Delta'' \supseteq \Delta'$ and δ'' is an extension of $\delta[b \mapsto A]$. By Lemma 14 we have that there exists θ' such that $\mathrm{rc}(F), \Delta'' \vdash \theta' : \Theta \Rightarrow \cdot$ and $\delta'' = (\theta' \circ \theta)$. Because δ'' is an extension of $\delta[b \mapsto A]$, this implies $A = \theta'(\theta(b))$.
  By $a \in \mathrm{rc}(F)$ we have $\theta'(a) = a$. Due to assumption $a \in \mathrm{ftv}(\theta(b))$ we then have $a \in \mathrm{ftv}(A)$ However, we have $a \notin \mathrm{btv}(F_p), \Delta'$ (3) and $\mathrm{rc}(F_p), \Delta' \vdash A$ **ok** (2), yielding the contradiction $a \notin \mathrm{ftv}(A)$.

(2) If f is of the the form $\mathbf{let}_\star\ x = \sqcap b.C_1\ \mathbf{in}\ C_2$ then the derivation of (1) must contain a sub-derivation of the following form:

$$\frac{\begin{array}{c}\vdots\\ (\mathrm{rc}(F_p), \Delta', \tilde{a}); (\mathrm{fc}(F_p), b); \delta(\mathrm{tc}(F_p)); \delta[b \mapsto A] \vdash C_1\end{array}}{(\mathrm{rc}(F_p), \Delta'); \mathrm{fc}(F_p); \delta(\mathrm{tc}(F_p)); \delta \vdash \mathbf{let}_\star\ x = \sqcap b.C_1\ \mathbf{in}\ C_2}$$

for some $A, \Delta', \tilde{a}$ and δ, where $C_1 = F_s[C \wedge \mathfrak{U}(\Theta, \theta)]$. We have $(\mathrm{rc}(F_p), \Delta', \tilde{a}) \vdash A\ \mathbf{ok}$ and $a \notin \mathrm{rc}(F_p), \Delta', \tilde{a}$ and may therefore obtain the same contradiction as in the previous case $f = \exists b$.

(3) The case $\mathbf{let}_\bullet\ x = \sqcap b.C_1\ \mathbf{in}\ C_2$ is analogous.

- Case $\exists a$: If the topmost stack frames of F have the shape $\mathbf{let}_R x = \sqcap a.\square\ \mathbf{in}\ C' :: \exists \overline{b}$, then S-LetPolyPop or S-LetMonoPop is applicable, as discussed before. Otherwise, due to our assumption about the shape of $F[C]$, there exists a frame f in F that isn't an $\exists$ frame and we can apply S-ExistsLower, which always succeeds.

□

B.4 Proof of Theorem 5

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

Proof. Follows immediately from Lemma 13, which guarantees the absence of infinite sequences of steps. □

B.5 Proof of Theorem 6

The following lemma is a slight variation of Theorem 6; we use it in the proof of the latter.

Lemma 21. *Let* $(\forall \Delta :: \exists \tilde{a} :: F, \Theta, \theta, C)\ \mathbf{ok}$. *Then we have*

$$\begin{array}{l}\Delta; \tilde{a}; \cdot; \delta \vdash F[C \wedge \mathfrak{U}(\Theta, \theta)]\\ \textit{iff}\\ \textit{there exist } \Theta', \theta'', \theta', \tilde{b} \textit{ s.t.}\\ \quad (\forall \Delta :: \exists \tilde{a} :: F, \Theta, \theta, C) \rightarrow^* (\forall \Delta :: \exists\ (\tilde{a}, \tilde{b}), \Theta', \theta', \mathsf{true}) \textit{ and}\\ \quad \Delta \vdash \theta'' : \Theta' \Rightarrow \cdot \textit{ and}\\ \quad (\theta'' \circ \theta')\restriction_{\tilde{a}} = \delta\end{array}$$

Proof. We show each direction individually:

$\Longrightarrow$ By transfinite induction on the well-ordering $<$ on stack machine states s whose existence is shown in Lemma 13. Hence, we assume that the left-to-right direction of the lemma holds for all s' on the left of the $\rightarrow^*$ s.t. $s' < s$ and show that the left-to-right direction holds for s on the left of $\rightarrow^*$, too.
To this end, let $s = (\forall \Delta :: \exists \tilde{a} :: F, \Theta, \theta, F, C)$ and we assume s **ok (1)** and $\Delta; \tilde{a}; \cdot; \delta \vdash F[C \wedge \mathfrak{U}(\Theta, \theta)]$ **(2)**.
We first consider the case that s is already a final state in the senses of this lemma, meaning that F is of the shape $\exists \tilde{b}$ for some $\tilde{b}$ and C is true. Further, we have $\theta = \theta'$ and $\Theta = \Theta'$, where $\mathrm{ftv}(\Theta) = \tilde{a}, \tilde{b}$.
This makes (2) equivalent to $\Delta; \tilde{a}; \cdot; \delta \vdash \exists \tilde{b}.\mathfrak{U}(\Theta, \theta)$. Applying Lemma 14 then gives us the existence of an appropriate θ''.

We now consider the case where s is not a final state, i.e., we don't have $F[C] = \exists \tilde{b}.\mathsf{true}$ for any $\tilde{b}$. We observe that (2) implies $; \cdot; \cdot; \emptyset \vdash \forall \Delta :: \exists \tilde{a} :: F[C \wedge \mathfrak{U}(\Theta, \theta)]$. This allows us to

apply Theorem 4, showing that the machine can take a step from s to a new state s_1. We now show that s_1 is of the form $(\forall\Delta :: \exists\tilde{a} :: F_1, \Theta_1, \theta_1, C_1)$ for some F_1, Θ_1, θ_1, and C_1:
If F is empty, then C must not be true. All stack machine rules applicable in this case preserve all existing stack frames. Otherwise, if F is not empty, we observe that the only rules of the stack machine that may replace more than the topmost stack frame are S-ExistsLower, S-LetMonoPop, and S-LetPolyPop.
If S-ExistsLower was applied, we observe that the only way for the variables $\tilde{a}$ in the definition of the rule S-ExistsLower not to be disjoint from the variables $\tilde{a}$ in the statement of this lemma is if the stack of s is of the form $\forall\Delta :: \exists\tilde{a} :: \exists\tilde{b}$ for some $\tilde{b}$, which violates the assumption about the shape of $F[C]$ above. Therefore, if S-ExistsLower was applied, the bottom-most frames $\forall\Delta :: \exists\tilde{a}$ of s remained unchanged. If S-LetPolyPop or S-LetMonoPop was applied, then F must contain a **let** frame and any stack frames below that in s (in particular, the frames $\forall\Delta :: \exists\tilde{a}$) remain unchanged.
Therefore, the $\forall\Delta :: \exists\tilde{a}$ frames at the bottom of s's stack are preserved by any rule possibly turning s into s_1. Using (1) and the fact that the lower stack frames of s_1 are $\forall\Delta :: \exists\tilde{a}$, we may apply Theorem 3 to the step $s \to s_1$, which gives us
(3) $\Delta; \tilde{a}; \cdot; \delta \vdash F[C \wedge \mathfrak{U}(\Theta, \theta)]$ iff $\Delta; \tilde{a}; \cdot; \delta \vdash F_1[C_1 \wedge \mathfrak{U}(\Theta_1, \theta_1)]$
By Lemma 13, we further have $s_1 < s$ **(4)** and by Lemma 17 s_1 **ok (5)**.
Combining (2) with (3) gives us $\Delta; \tilde{a}; \cdot; \delta \vdash F_1[C_1 \wedge \mathfrak{U}(\Theta_1, \theta_1)]$. This, together with (5) and (4) allows us to apply the induction hypothesis to s_1. This gives us the existence of $\Theta', \theta'', \theta', \tilde{a}$ s.t.
(6) $(\forall\Delta :: \exists\tilde{a} :: F_1, \Theta_1, \theta_1, C_1) \to^* (\forall\Delta :: \exists\ (\tilde{a}, \tilde{b}), \Theta', \theta', \mathsf{true})$
(7) $\Delta \vdash \theta'' : \Theta' \Rightarrow \cdot$
(8) $(\theta'' \circ \theta')\restriction_{\tilde{a}} = \delta$.
The step $s \to s_1$ extends (6) to $(\forall\Delta :: \exists\tilde{a} :: F, \Theta, \theta, C) \to^* (\forall\Delta :: \exists\ (\tilde{a}, \tilde{b}), \Theta', \theta', \mathsf{true})$ and (7) as well as (8) show us that θ'' has the desired properties.

$\Longleftarrow$ Let s be the state $(\forall\Delta :: \exists\tilde{a} :: F, \Theta, \theta, C)$. We prove this direction by induction on the length n of the sequence $s \to^n (\forall\Delta :: \exists\ (\tilde{a}, \tilde{b}), \Theta', \theta', \mathsf{true})$. By assumption, we also have $\Delta \vdash \theta'' : \Theta' \Rightarrow \cdot$ **(9)** and $(\theta'' \circ \theta')\restriction_{\tilde{a}} = \delta$ **(10)**.
If $n = 0$ we have $\Theta = \Theta'$, $\theta = \theta'$, $C = \mathsf{true}$, and $F = \exists\tilde{b}$. The property to prove simplifies to $\Delta; \tilde{a}; \cdot; \delta \vdash \exists\tilde{b}.\mathfrak{U}(\Theta, \theta)$. This follows directly from applying Lemma 14 to (9) and (10).

In the inductive step there exists some s_1 s.t.

$$s \to s_1 \to^\star (\forall\Delta :: \exists(\tilde{a}, \tilde{b}), \Theta', \theta', \mathsf{true})$$

We now assume that $F[C]$ is not of the form $\exists\tilde{c}.\mathsf{true}$ for any $\tilde{c}$ (otherwise, s would already be a final state in the sense of this lemma and we finish the proof directly using the $n = 0$ case above).
Therefore, using the same reasoning as in the $\Longrightarrow$ direction, we know that s_1 is of the form $(\forall\Delta :: \exists\tilde{a} :: F_1, \Theta_1, \theta_1, C_1)$ for some F_1, Θ_1, θ_1, and C_1. According to Lemma 17, we have s_1 **ok**. We can therefore apply Theorem 3 to this single step, yielding
(11) $\Delta; \tilde{a}; \cdot; \delta \vdash F[C \wedge \mathfrak{U}(\Theta, \theta)]$ iff $\Delta; \tilde{a}; \cdot; \delta \vdash F_1[C_1 \wedge \mathfrak{U}(\Theta_1, \theta_1)]$
We apply the induction hypothesis to the sequence $s_1 \to^\star (\forall\Delta :: \exists(\tilde{a}, \tilde{b}), \Theta', \theta', \mathsf{true})$, yielding $\Delta; \widetilde{a}; \cdot; \delta \vdash F_1[C_1 \wedge \mathfrak{U}(\Theta_1, \theta_1)]$ By (11), this gives us the desired property $\Delta; \widetilde{a}); \cdot; \delta \vdash F[C \wedge \mathfrak{U}(\Theta, \theta)]$.

□

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

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

Proof. Let $\overline{a}$ be an arbitrary ordering of the variables in Ξ. Further, let $\theta_a := [\overline{a} \mapsto \overline{a}]$ and $\Theta_a := (\widetilde{a : \star})$. We have

$$(\cdot, \cdot, \emptyset, \forall\Delta.\exists\tilde{a}.\mathbf{def}\ \Gamma\ \mathbf{in}\ C) \ \to^* \ (\forall\Delta :: \exists\tilde{a}, \Theta_a, \theta_a, \mathbf{def}\ \Gamma\ \mathbf{in}\ C)$$

after $|\Delta|$ applications of the rule S-ForallPush and $|\tilde{a}|$ applications of S-ExistsPush. Let the former state be defined as s, the latter one as s'. Here, due to $\Delta; \Theta; \Gamma \vdash C$ **ok**, we have s' **ok**.

Therefore, for all $\hat{\Theta}, \hat{\theta}, \tilde{b}$ we have

$$\begin{array}{rcl} s' & \to^* & (\forall\Delta :: \exists\ (\tilde{a}, \tilde{b}), \hat{\Theta}, \hat{\theta}, \mathsf{true}) \\ & & \text{iff} \\ s & \to^* & (\forall\Delta :: \exists\ (\tilde{a}, \tilde{b}), \hat{\Theta}, \hat{\theta}, \mathsf{true}) \end{array}$$

Now, let F be the empty stack. We then have

	$\Delta; \Xi; \Gamma; \delta \vdash C$	
iff	$\Delta; \Xi; \cdot; \delta \vdash (\mathbf{def}\ \Gamma\ \mathbf{in}\ C)$	(by $\Delta \vdash \Gamma$: all mono. conditions satisfied)
iff	$\Delta; \Xi; \cdot; \delta \vdash (\mathbf{def}\ \Gamma\ \mathbf{in}\ C) \wedge \mathfrak{U}(\Theta_a, \theta_a)$	($\mathfrak{U}(\Theta_a, \theta_a)$ is equivalent to true)
iff	$\Delta; \Xi; \cdot; \delta \vdash F[(\mathbf{def}\ \Gamma\ \mathbf{in}\ C) \wedge \mathfrak{U}(\Theta_a, \theta_a)]$	(F is empty)

Using this, the equivalence to prove then follows directly from Lemma 21. □

B.6 Proof of Theorem 7

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

Proof. Suppose $(\cdot, \cdot, \emptyset, \forall\Delta.\exists a.\mathbf{def}\ \Gamma\ \mathbf{in}\ [\![M : a]\!]) \to^* (\forall\Delta :: \exists\ (a, \tilde{b}), \Theta, \theta, \mathsf{true})$ and $\Delta \vdash \theta' : \Theta \Rightarrow \cdot$. Let s refer to the first state of the sequence above and s' to its last state.

We apply Lemma 11, which gives us $\Delta; a; \Gamma \vdash [\![M : a]\!]$ **ok**. We therefore have $\vdash s$ **ok**. By Lemma 17 we then have $\vdash s'$ **ok**, too, which implies $\Delta \vdash \Theta \Rightarrow \cdot$ and $\mathrm{ftv}(\Theta) = \tilde{b}, a$.

We can therefore define δ as $\theta' \circ \theta\restriction_{\{a\}}$. This allows us to apply Theorem 6. We instantiate the right-to-left direction of the theorem such that we need to show that the following properties hold:

(1) $(\cdot, \cdot, \emptyset, \forall\Delta.\, \exists a.\, \mathbf{def}\ \Gamma\ \mathbf{in}\ C) \to^* (\forall\Delta :: \exists\ (a, \tilde{b}), \Theta, \theta, \mathsf{true})$
(2) $\Delta \vdash \theta' : \Theta \Rightarrow \cdot$
(3) $(\theta' \circ \theta)\restriction_a = \delta$

The first two properties follow immediately by assumption, the third one holds by definition of δ. Therefore, the right-to-left direction of Theorem 6 gives us $\Delta; a; \Gamma; \delta \vdash [\![M : a]\!]$. Theorem 2 then immediately yields $\Delta; \Gamma \vdash M : \delta(a)$. By definition of δ this is equivalent to the property to show.

□

B.7 Proof of Theorem 8

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

Proof. We assume $\Delta; \Gamma \vdash M : A$, which implies $\Delta \vdash \Gamma$ and $\Delta; \Gamma \vdash M$ **ok**. This means that Theorem 1 gives us $\Delta; a; \Gamma; [a \mapsto A] \vdash [\![M : a]\!]$.

We apply the left-to-right direction of Theorem 6 (using $[\![M : a]\!]$ for C and $[a \mapsto A]$ for δ in the theorem's statement) which gives us the existence of $\Theta, \theta, \theta', \tilde{c}$ s.t.

(1) $(\cdot, \cdot, \emptyset, \forall \Delta . \exists a . \mathbf{def}\ \Gamma\ \mathbf{in}\ [\![M : a]\!]) \rightarrow^* (\forall \Delta :: \exists (a, \tilde{c}), \Theta, \theta, \mathsf{true})$

(2) $\Delta \vdash \theta' : \Theta \Rightarrow \cdot$

(3) $(\theta' \circ \theta)\restriction_{\{a\}} = [a \mapsto A]$

Clearly, we have $(\theta' \circ \theta\restriction_{\{a\}})(a) = (\theta' \circ \theta)(a) = [a \mapsto A](a) = A$, which is the second property we need to show. By choosing $\Xi = (a, \tilde{c})$, property (1) becomes the first property that we needed to show.

□