FreezeML

Complete and Easy Type Inference for First-Class Polymorphism

Frank Emrich
The University of Edinburgh
frank.emrich@ed.ac.uk

Sam Lindley
The University of Edinburgh
Imperial College London
sam.lindley@ed.ac.uk

Jan Stolarek
The University of Edinburgh
Lodz University of Technology
jan.stolarek@ed.ac.uk

James Cheney
The University of Edinburgh
The Alan Turing Institute
jcheney@inf.ed.ac.uk

Jonathan Coates
The University of Edinburgh
s1627856@sms.ed.ac.uk

Abstract

ML is remarkable in providing statically typed polymorphism without the programmer ever having to write any type annotations. The cost of this parsimony is that the programmer is limited to a form of polymorphism in which quantifiers can occur only at the outermost level of a type and type variables can be instantiated only with monomorphic types.

Type inference for unrestricted System F-style polymorphism is undecidable in general. Nevertheless, the literature abounds with a range of proposals to bridge the gap between ML and System F.

We put forth a new proposal, FreezeML, a conservative extension of ML with two new features. First, let- and lambda-binders may be annotated with arbitrary System F types. Second, variable occurrences may be *frozen*, explicitly disabling instantiation. FreezeML is equipped with type-preserving translations back and forth between System F and admits a type inference algorithm, an extension of algorithm W, that is sound and complete and which yields principal types.

Keywords first-class polymorphism, type inference, impredicative types

1 Introduction

The design of ML [18] hits a sweet spot in providing statically typed polymorphism without the programmer ever having to write any type annotations. The Hindley-Milner type inference algorithm on which ML relies is *sound* (it only yields correct types) and *complete* (if a program has a type then it will be inferred). Moreover, inferred types are always *principal*, that is, most general. Alas, this sweet spot is rather narrow, depending on a delicate balance of features; it still appears to be an open question how best to extend ML type inference to support first-class polymorphism as found in System F.

Nevertheless, ML has unquestionable strengths as the basis for high-level programming languages. Its implicit polymorphism is extremely convenient for writing concise programs. Functional programming languages such Haskell and OCaml employ algorithms based on Hindley-Milner type inference and go to great efforts to reduce the need to write type annotations on programs. Whereas the plain Hindley-Milner algorithm supports a limited form of polymorphism in which quantifiers must be top-level and may only be instantiated with monomorphic types, advanced programming techniques often rely on first-class polymorphism, where quantifiers may appear anywhere and may be instantiated with arbitrary polymorphic types, as in System F. However, working directly in System F is painful due to the need for explicit type abstractions and applications. Alas, type inference, and indeed type checking, is undecidable for System F [29] without type annotations. Moreover, even in System F with type annotations but no explicit instantiation, type inference remains undecidable [21].

The primary difficulty in extending ML to support first-class polymorphism is with the implicit instantiation of polymorphic type schemes: whenever a variable occurrence is typechecked, any quantified type variables are immediately instantiated with (monomorphic) types. Whereas in plain ML there is nothing to be lost by greedily instantiating type variables, with first-class polymorphism there is sometimes a non-trivial choice to be made over whether to instantiate or not. The basic Hindley-Milner algorithm [3] restricts the use of polymorphism in types to *type schemes* of the form $\forall \vec{a}.A$ where A does not contain any further polymorphism. This means that, for example, given a function $\mathsf{single} : \forall a.a \to \mathsf{List}\ a$, that constructs a list of one element, and a polymorphic function choosing its first argument $\mathsf{choose} : \forall a.a \to a \to a$, the expression single choose is assigned the type $\forall a.\mathsf{List}\ (a \to a \to a)$, which is a polymorphic list of functions of type $a \to a \to a$. The type $\forall a.\mathsf{List}\ (a \to a \to a)$ arises from instantiating the quantifier of single with $a \to a \to a$. But what if instead of constructing a polymorphic list of choice functions a programmer wishes to construct a list of polymorphic choice functions, i.e. a list of type $\mathsf{List}\ (\forall a.a \to a \to a)$? This requires instantiating the quantifier of single with a polymorphic type $\forall a.a \to a \to a$. In ML, it is forbidden to instantiate type variables with polymorphic types, thus it is impossible to construct a term of type List $(\forall a.a \to a \to a)$. (Indeed, this System F type is not even an ML type scheme.) However, in a richer language such as System F, the expression single choose could be annotated as appropriate in order to obtain either the type $\forall a.\mathsf{List}\ (a \to a \to a)$ or the type $\mathsf{List}\ (\forall a.a \to a \to a)$.

All is not lost. By adding a sprinkling of explicit type annotations, in combination with other extensions, it is possible to retain much of the convenience of ML alongside the expressiveness of System F. Indeed, there is a plethora of techniques bridging the expressiveness gap between ML and System F without sacrificing desirable type inference properties of ML [6, 10–13, 23, 24, 26, 27].

However, there is still not a wide consensus on what constitutes a good design for a language combining ML-style type inference with System F-style first-class polymorphism, beyond the typical criteria of decidability, soundness, completeness, and principal typing. As Serrano et al. [24] put it in their PLDI 2018 paper, type inference in the presence of first-class polymorphism is still "a deep, deep swamp" and "no solution (...) with a good benefit-to-weight ratio has been presented to date". While previous proposals offer considerable expressive power, we nevertheless consider the following combination of design goals to be both compelling and not yet achieved by any prior work:

- **Familiar System F types** Our ideal solution would use exactly the type language of System F. Several approaches, such as HML [12], MLF [10], Poly-ML[1] [6], and QML [23], capture (or exceed) the power of System F, but employ a strict superset of System F's type language. While in some cases this difference is superficial, we consider that it does increase the burden on the programmer to understand and use these features effectively, and may also contribute to increasing the syntactic overhead and decreasing the clarity of programs.
- **Close to ML type inference** Our ideal solution would conservatively extend ML and standard Hindley-Milner type inference, including the (now-standard) *value restriction* [30]. Approaches such as MLF and Boxy Types have relied on much more sophisticated type inference techniques than needed in classical Hindley-Milner type inference, and proven difficult to implement or extend further because of their complexity. Other approaches, such as GI, are relatively straightforward to implement atop an OutsideIn(X)-style constraint-based type inference algorithm, but would be much more work to add to a standard Hindley-Milner implementation.
- **Low syntactic overhead** Our ideal solution would provide first-class polymorphism without significant departures from ordinary ML-style programming. Early approaches [8, 9, 19, 22] showed how to accommodate System F-style polymorphism by associating it with nominal datatype constructors, but this imposes a significant syntactic overhead to make use of these capabilities, which can also affect the readability and maintainability of programs. All previous approaches necessarily involve some type annotations as well, which we also desire to minimise as much as possible.
- **Predictable behaviour** Our ideal solution would avoid guessing polymorphism and be specified so that programmers can anticipate where type annotations will be needed. More recent approaches, such as HMF [11] and GI [24], use System F types, and are relatively easy to implement, but employ heuristics to guess one of several different polymorphic types, and require programmer annotations if the default heuristic behaviour is not what is needed.

In short, we consider that the problem of extending ML-style type inference with the power of System F is solved as a *technical* problem by several existing approaches, but there remains a significant *design* challenge to develop an approach that uses *familiar System F types*, is *close to ML type inference*, has *low syntactic overhead*, and has *predictable behaviour*. Of course, these desiderata represent our (considered, but subjective) views as language designers, and others may (and likely will) disagree. We welcome such debate.

Our contribution: FreezeML In this paper, we introduce FreezeML, a core language extending ML with two System F-like features:

- "frozen" variable occurrences for which polymorphic instantiation is inhibited (written $\lceil x \rceil$ to distinguish them from ordinary variables x whose polymorphic types are implicitly instantiated); and
- type-annotated lambda abstractions $\lambda(x : A).M$.

FreezeML also refines the typing rule for let by:

- restricting let-bindings to have principal types; and
- allowing type annotations on let-bindings.

As we shall see in Section 2, the introduction of type-annotated let-bindings and frozen variables allows us to macro-express explicit versions of generalisation and instantiation (the two features that are implicit in plain ML). Thus, unlike ML, although FreezeML still has ML-like variables and let-binding it *also* enjoys explicit encodings of all of the underlying System F features. Correspondingly, frozen variables and type-annotated let-bindings are also central to encoding type abstraction and type application of System F (Section 4.1). Although, as we explain later, our approach is similar in expressiveness to existing proposals such as Poly-ML, we believe its close alignment with System F types and ML type inference are important benefits, and we argue via examples that its syntactic overhead and predictability compare favourably with the state of the art. Nevertheless, further work would need to be done to systematically compare the syntactic overhead and predictability of our approach with

[1]The name Poly-ML does not appear in the original [6] paper, but was introduced retrospectively [10].

existing approaches — this criticism, however, also applies to most previous work on new language design ideas.

Contributions This paper is a programming language design paper. Though we have an implementation on top of the Links programming language [2][2] implementation is not the primary focus. The paper makes the following main contributions:

- A high-level introduction to FreezeML (Section 2).
- A type system for FreezeML as a conservative extension of ML with the expressive power of System F (Section 3).
- Local type-preserving translations back and forth between System F and FreezeML, and a discussion of the equational theory of FreezeML (Section 4).
- A type inference algorithm for FreezeML as an extension of algorithm W [3], which is sound, complete, and yields principal types (Section 5).

Section 6 discusses implementation, Section 7 presents related work and Section 8 concludes.

2 An Overview of FreezeML

We begin with an informal overview of FreezeML. Recall that the types of FreezeML are exactly those of System F. Appendix A contains an additional collection of code examples that showcase how our system works in practice.

Implicit Instantiation In FreezeML (as in standard Hindley-Milner type inference), when variable occurrences are type-checked, the outer universally quantified type variables in the variable's type are instantiated implicitly. Suppose a programmer writes choose id, where variable choose has polymorphic type $\forall a.a \to a \to a$. The identity function id has type $\forall a.a \to a$, and (as in ML) the quantifier in the type of id is implicitly instantiated with an as yet unknown type a, yielding the type $a \to a$. This type is then used to instantiate the quantifier in the type of choose, resulting in the final type of the expression $(a \to a) \to (a \to a)$.

Explicit Freezing **($\lceil x \rceil$)** The programmer may explicitly prevent a variable from having its quantifiers instantiated by using the freeze operator $\lceil - \rceil$. Whereas each ordinary occurrence of choose has type $a \to a \to a$ for an unknown type a, a frozen occurrence $\lceil \mathsf{choose} \rceil$ has type $\forall a.a \to a \to a$. More interestingly, whereas the term single choose has type List $(a \to a \to a)$, the term single $\lceil \mathsf{choose} \rceil$ has type List $(\forall a.a \to a \to a)$. This makes it possible to pass polymorphic arguments to functions that expect them. Consider the term auto id, where auto : $(\forall a.a \to a) \to (\forall a.a \to a)$. This will not typecheck because id will be implicitly instantiated to type $a \to a$ which does not match the argument type $\forall a.a \to a$ of auto. We can however apply auto to id by freezing the identity function: auto $\lceil \mathsf{id} \rceil$, which does typecheck.

Explicit Generalisation **($\$V$)** We can generalise an expression to its principal polymorphic type by binding it to a variable and then freezing it, for instance: **let** $id = \lambda x.x$ **in** poly $\lceil id \rceil$, where poly : $(\forall a.a \to a) \to \mathsf{Int} \times \mathsf{Bool}$. The explicit generalisation operator \$ generalises the type of any value. Whereas the term $\lambda x.x$ has type $a \to a$, the term $\$(\lambda x.x)$ has type $\forall a.a \to a$, allowing us to write poly $\$(\lambda x.x)$. Explicit generalisation is macro-expressible [5] in FreezeML.

$$\$V \equiv \textbf{let}\ x = V\ \textbf{in}\ \lceil x \rceil$$

We can also define a type-annotated variant:

$$\$^A V \equiv \textbf{let}\ (x : A) = V\ \textbf{in}\ \lceil x \rceil$$

Note that FreezeML adopts the ML value restriction [30]; hence let generalisation only applies to syntactic values.

Explicit Instantiation **($@M$)** As in ML, the polymorphic types of variables are implicitly instantiated when typechecking each variable occurrence. Unlike in ML, other terms can have polymorphic types, which are *not* implicitly instantiated. Nevertheless, we can instantiate a term by binding it to a variable: **let** x = head ids **in** $x\,42$, where head : $\forall a.\mathsf{List}\ (a) \to a$ returns the first element in a list and ids : List $(\forall a.a \to a)$ is a list of polymorphic identity functions. The explicit instantiation operator @ supports instantiation of a term without having to explicitly bind it to a variable. For instance, whereas the term head ids has type $\forall a.a \to a$ the term (head ids)@ has type $\mathsf{Int} \to \mathsf{Int}$, so (head ids)@ 42 is well-formed. Explicit instantiation is macro-expressible in FreezeML:

$$M@ \equiv \textbf{let}\ x = M\ \textbf{in}\ x$$

Ordered Quantifiers Like in System F, but unlike in ML, the order of quantifiers matters. Generalised quantifiers are ordered according to the order in which they first appear in a type. Note that when we generalise a variable it will first be implicitly instantiated and only then generalised. In some cases instantiating a variable and then generalising it leads to reordering of quantifiers, while in other cases it does not. For example, if we have functions $f : (\forall a\ b.a \to b \to a \times b) \to \mathsf{Int}$, pair : $\forall a\ b.a \to b \to a \times b$, and pair$'$: $\forall b\ a.a \to b \to a \times b$, then $f\ \lceil \mathsf{pair} \rceil$, $f\ \$\mathsf{pair}$, and $f\ \$\mathsf{pair}'$ have type Int and behave identically. Notice how \$pair$'$ gets instantiated first and then generalised to have the quantifiers ordered in a way the f function expects. Note also how expression $f\ \lceil \mathsf{pair}' \rceil$ is ill-typed as the order of quantifiers on $\lceil \mathsf{pair}' \rceil$ does not match the requirements of the f function.

Monomorphic parameter inference As in ML, function arguments need not have annotations, but their inferred types must be monomorphic, i.e. we cannot typecheck bad:

$$\mathsf{bad} = \lambda f.(f\ 42, f\ \mathsf{True})$$

[2] https://github.com/links-lang/links

Unlike in ML we can annotate arguments with polymorphic types and use them at different types:

$$\mathsf{poly} = \lambda(f : \forall a.a \to a).(f\ 42, f\ \mathsf{True})$$

One might hope that it is safe to infer polymorphism by local, compositional reasoning, but that is not the case. Consider the following two functions.

$$\begin{aligned}\mathsf{bad1} &= \lambda f.(\mathsf{poly}\ \lceil f \rceil, (f\ 42) + 1)\\ \mathsf{bad2} &= \lambda f.((f\ 42) + 1, \mathsf{poly}\ \lceil f \rceil)\end{aligned}$$

We might reasonably expect both to be typeable with a declarative type system that assigns the type $\forall a.a \to a$ to f. Now, assume type inference proceeds from left to right. In bad1 we first infer that f has type $\forall a.a \to a$ (as $\lceil f \rceil$ is the argument to poly); then we may instantiate a to Int when applying f to 42. In bad2 we eagerly infer that f has type $\mathsf{Int} \to \mathsf{Int}$; now when we pass $\lceil f \rceil$ to poly, type inference fails. To rule out this kind of sensitivity to the order of type inference, and the resulting incompleteness of our type inference algorithm, we insist that unannotated λ-bound variables be monomorphic. This in turn entails checking monomorphism constraints on type variables and maintaining other invariants (Section 3.2). (One can build more sophisticated systems that defer determining whether a term is polymorphic or not until more information becomes available — both Poly-ML and MLF do, for instance — but we prefer to keep things simple.)

3 FreezeML via System F and ML

In this section we give a syntax-directed presentation of FreezeML and discuss various design choices that we have made. We wish for FreezeML to be an ML-like call-by-value language with the expressive power of System F. To this end we rely on a standard call-by-value definition of System F, which additionally obeys the value restriction (i.e. only values are allowed under type abstractions). We take mini-ML [1] as a core representation of a call-by-value ML language. Unlike System F, ML separates monotypes from (polymorphic) type schemes and has no explicit type abstraction and application. Polymorphism in ML is introduced by generalising the body of a let-binding, and eliminated implicitly when using a variable. Another crucial difference between System F and ML is that in the former the order of quantifiers in a polymorphic type matters, whereas in the latter it does not. Full definitions of System F and ML, including the syntax, kinding and typing rules, as well as translation from ML to System F, are given in Appendix C.

Notations. We write $\mathsf{ftv}(A)$ for the sequence of distinct free type variables of a type in the order in which they first appear in A. For example, $\mathsf{ftv}((a \to b) \to (a \to c)) = a, b, c$. We write $\Delta - \Delta'$ for the restriction of Δ to those type variables that do not appear in Δ'. We write $\Delta \# \Delta'$ to mean that the type variables in Δ and Δ' are disjoint. Disjointedness is also implicitly required when concatenating Δ and Δ' to Δ, Δ'.

Type Variables	$\mathsf{TVar} \ni a, b, c$	
Type Constructors	$\mathsf{Con} \ni D$	$::= \mathsf{Int} \mid \mathsf{List} \mid \to \mid \times \mid \ldots$
Types	$\mathsf{Type} \ni A, B$	$::= a \mid D\,\overline{A} \mid \forall a.A$
Monotypes	$\mathsf{MType} \ni S, T$	$::= a \mid D\,\overline{S}$
Guarded Types	$\mathsf{GType} \ni H$	$::= a \mid D\,\overline{A}$
Type Instantiation	$\mathsf{Subst} \ni \delta$	$::= \emptyset \mid \delta[a \mapsto S]$
Term Variables	$\mathsf{Var} \ni x, y, z$	
Terms	$\mathsf{Term} \ni M, N$	$::= x \mid \lceil x \rceil \mid \lambda x.M$
		$\mid\ \lambda(x : A).M \mid M\,N$
		$\mid\ \mathbf{let}\ x = M\ \mathbf{in}\ N$
		$\mid\ \mathbf{let}\ (x : A) = M\ \mathbf{in}\ N$
Values	$\mathsf{Val} \ni V, W$	$::= x \mid \lceil x \rceil \mid \lambda x.M$
		$\mid\ \lambda(x : A).M$
		$\mid\ \mathbf{let}\ x = V\ \mathbf{in}\ W$
		$\mid\ \mathbf{let}\ (x : A) = V\ \mathbf{in}\ W$
Guarded Values	$\mathsf{GVal} \ni U$	$::= x \mid \lambda x.M \mid \lambda(x{:}A).M$
		$\mid\ \mathbf{let}\ x = V\ \mathbf{in}\ U$
		$\mid\ \mathbf{let}\ (x : A) = V\ \mathbf{in}\ U$
Kinds	$\mathsf{Kind} \ni K$	$::= \bullet \mid \star$
Kind Environments	$\mathsf{PEnv} \ni \Delta$	$::= \cdot \mid \Delta, a$
Type Environments	$\mathsf{TEnv} \ni \Gamma$	$::= \cdot \mid \Gamma, x : A$

Figure 1. FreezeML Syntax

3.1 FreezeML

FreezeML is an extension of ML with two new features. First, let-bindings and lambda-bindings may be annotated with arbitrary System F types. Second, FreezeML adds a new form $\lceil x \rceil$, called *frozen variables*, for preventing variables from being instantiated.

The syntax of FreezeML is given in Figure 1. (We name the syntactic categories for later use in Section 5.) The types are the same as in System F. We explicitly distinguish two kinds of type: a monotype (S), is as in ML a type entirely free of polymorphism, and a guarded type (H) is a type with no top-level quantifier (in which any polymorphism is guarded by a type constructor). The terms include all ML terms plus frozen variables ($\lceil x \rceil$) and lambda- and let-bindings with type ascriptions. Values are those terms that may be generalised. They are slightly more general than the value forms of Standard ML in that they are closed under let binding (as in OCaml). Guarded values are those values that can only have guarded types (that is, all values except those that have a frozen variable in tail position).

The FreezeML kinding judgement $\Delta \vdash A : K$ states that type A has kind K in kind environment Δ. The kinding rules are given in Figure 2. As in ML we distinguish monomorphic types ($\bullet$) from polymorphic types ($\star$). Unlike in ML polymorphic types can appear inside data type constructors.

Type instantiation is adjusted to account for polymorphism by either restricting it to instantiate type variables with monomorphic kinds only ($\Rightarrow_\bullet$) or permit polymorphic

$$\boxed{\Delta \vdash A : K}$$

$$\frac{a \in \Delta}{\Delta \vdash a : \bullet} \qquad \frac{\begin{array}{c}\mathsf{arity}(D) = n \\ \Delta \vdash A_1 : K \\ \cdots \\ \Delta \vdash A_n : K\end{array}}{\Delta \vdash D\,\overline{A} : K} \qquad \frac{\Delta, a \vdash A : \star}{\Delta \vdash \forall a.A : \star} \qquad \frac{\Delta \vdash A : \bullet}{\Delta \vdash A : \star}$$

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

$$\frac{}{\Delta \vdash \emptyset : \cdot \Rightarrow_K \Delta'} \qquad \frac{\Delta \vdash \delta : \Delta' \Rightarrow_K \Delta'' \qquad \Delta, \Delta'' \vdash A : K}{\Delta \vdash \delta[a \mapsto A] : (\Delta', a) \Rightarrow_K \Delta''}$$

Figure 2. FreezeML Kinding and Instantiation Rules

$$\begin{array}{cc} \emptyset(A) = A & \delta[a \mapsto A](a) = A \\ \delta(D\,\overline{A}) = D\,(\overline{\delta(A)}) & \delta[a \mapsto A](b) = \delta(b) \\ \multicolumn{2}{c}{\delta(\forall a.A) = \forall c.\delta[a \mapsto c](A), \text{where } c \notin \mathsf{ftv}(\delta(b)) \text{ for all } b \neq c} \end{array}$$

Figure 3. Application of a Type Instantiation in FreezeML

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

$$\textsc{Freeze}\ \frac{x : A \in \Gamma}{\Delta; \Gamma \vdash \lceil x \rceil : A} \qquad \textsc{Var}\ \frac{\begin{array}{c}x : \forall \Delta'.H \in \Gamma \\ \Delta \vdash \delta : \Delta' \Rightarrow_\star \cdot\end{array}}{\Delta; \Gamma \vdash x : \delta(H)} \qquad \textsc{App}\ \frac{\begin{array}{c}\Delta; \Gamma \vdash M : A \to B \\ \Delta; \Gamma \vdash N : A\end{array}}{\Delta; \Gamma \vdash M\,N : B}$$

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

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

$$\textsc{Let-Ascribe}\ \frac{\begin{array}{c}(\Delta', A') = \mathsf{split}(A, M) \\ \Delta, \Delta'; \Gamma \vdash M : A' \qquad \Delta; \Gamma, x : A \vdash N : B\end{array}}{\Delta; \Gamma \vdash \mathbf{let}\ (x : A) = M\ \mathbf{in}\ N : B}$$

Figure 4. FreezeML typing rules

instantiations ($\Rightarrow_\star$). The following rule is admissible

$$\frac{\Delta, \Delta' \vdash A : K \qquad \Delta \vdash \delta : \Delta' \Rightarrow_{K'} \Delta''}{\Delta, \Delta'' \vdash \delta(A) : K \sqcup K'}$$

where $\bullet \sqcup \bullet = \bullet$ and $\bullet \sqcup \star = \star \sqcup \bullet = \star \sqcup \star = \star$. When applying a type instantiation we take care to account for shadowing of type variables (Figure 3).

The FreezeML judgement $\Delta; \Gamma \vdash M : A$ states that term M has type A in kind environment Δ and type environment Γ; its rules are shown in Figure 4. These rules are adjusted with respect to ML to allow full System F types everywhere except

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

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

$$\mathsf{gen}(\Delta, A, M) = \begin{cases} (\Delta', \Delta') & \text{if } M \in \mathsf{GVal} \\ (\cdot, \quad \Delta') & \text{otherwise} \end{cases}$$
$$\text{where } \Delta' = \mathsf{ftv}(A) - \Delta$$

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

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

Figure 5. FreezeML auxiliary definitions

in the types of variables bound by unannotated lambdas, where only monotypes are permitted.

As in ML, the Var rule implicitly instantiates variables. The $\star$ in the judgement $\Delta \vdash \delta : \Delta' \Rightarrow_\star \cdot$ indicates that the type variables in Δ' may be instantiated with polymorphic types. The Freeze rule differs from the Var rule only in that it suppresses instantiation. In the Lam rule, the restriction to a syntactically monomorphic argument type ensures that an argument cannot be used at different types inside the body of a lambda abstraction. However, the type of an unannotated lambda abstraction may subsequently be generalised. For example, consider the expression poly $\$(\lambda x.x)$. The parameter x cannot be typed with a polymorphic type; giving the syntactic monotype a to x yields type $a \to a$ for the lambda-abstraction. The \$ operator then generalises this to $\forall a.a \to a$ as the type of argument passed to poly. The Lam-Ascribe rule allows an argument to be used polymorphically inside the body of a lambda abstraction. The App rule is standard.

Let Bindings Because we adopt the value restriction, the Let rule behaves differently depending on whether or not M is a guarded value (cf. GVal syntactic category in Figure 1). The choice of whether to generalise the type of M is delegated to the judgement $(\Delta, \Delta'', M, A') \Updownarrow A$, where A' is the type of M and Δ'' are the generalisable type variables of M, i.e. $\Delta'' = \mathsf{ftv}(A') - \Delta$. The $\Updownarrow$ judgement determines A, the type given to x while type-checking N. If M is a guarded value, we generalise and have $A = \forall \Delta''.A'$. If M is not a guarded value, we have $A = \delta(A')$, where δ is an instantiation with $\Delta \vdash \delta : \Delta'' \Rightarrow_\bullet \cdot$. This means that instead of abstracting over the unbound type variables Δ'' of A', we instantiate them *monomorphically*. We further discuss the need for this behaviour in Section 3.2.

The gen judgement used in the Let rule may seem surprising — its first component is unused whilst the second component is identical in both cases and corresponds to the generalisable type variables of A'. Indeed, the first component of gen is irrelevant for typing but it is convenient for writing the translation from FreezeML to System F (Figure 8 in Section 4.2), where it is used to form a type abstraction, and in the type inference algorithm (Figure 13 in Section 5.4), where it allows us to collapse two cases into one.

The Let rule requires that A' is the principal type for M. This constraint is necessary to ensure completeness of our type inference algorithm; we discuss it further in Section 3.2. The relation principal is defined in Figure 5.

The Let-Ascribe rule is similar to the Let rule, but instead of generalising the type of M, it uses the type A supplied via an annotation. As in Let, A' denotes the type of M. However, the annotated case admits non-principal types for M. The split operator enforces the value restriction. If M is a guarded value, A' must be a guarded type, i.e. we have $A' = H$ for some H. We then have $A = \forall\Delta'.H$. If M is not a guarded value split requires $A' = A$ and $\Delta' = \cdot$. This means that *all* toplevel quantifiers in A must originate from M itself, rather than from generalising it.

Every valid typing judgement in ML is also a valid typing judgement in FreezeML.

Theorem 1. *If* $\Delta;\Gamma \vdash M : S$ *in ML then* $\Delta;\Gamma \vdash M : S$ *in FreezeML.*

(The exact derivation can differ due to differences in the kinding rules and the principality constraint on the Let rule.)

3.2 Design Considerations

Monomorphic instantiation in the Let rule Recall that the Let rule enforces the value restriction by instantiating those type variables that would otherwise be quantified over. Requiring these type variables to be instantiated with monotypes allows us to avoid problems similar to the ones outlined in Section 2. Consider the following two functions.

$$\mathsf{bad3} = \lambda(bot : \forall a.a).\mathbf{let}\ f = bot\ bot\ \mathbf{in}\ (\mathsf{poly}\ \lceil f \rceil, (f\ 42) + 1)$$
$$\mathsf{bad4} = \lambda(bot : \forall a.a).\mathbf{let}\ f = bot\ bot\ \mathbf{in}\ ((f\ 42) + 1, \mathsf{poly}\ \lceil f \rceil)$$

Since we do not generalise non-values in let-bindings due to the value restriction, in both of these examples f is initially assigned the type a rather than the most general type $\forall a.a$ (because *bot bot* is a non-value). Assuming type inference proceeds from left to right then type inference will succeed on bad3 and fail on bad4 for the same reasons as in Section 2. In order to rule out this class of examples, we insist that non-values are first generalised and then instantiated with monomorphic types. Thus we constrain a to only unify with monomorphic types, which leads to type inference failing on both bad3 and bad4.

Our guiding principle is "never guess polymorphism". The high-level invariant that FreezeML uses to ensure that this principle is not violated is that any (as yet) unknown types appearing in the type environment (which maps term variables to their currently inferred types) during type inference must be explicitly marked as monomorphic. The only means by which inference can introduce unknown types into the type environment are through unannotated lambda-binders or through not-generalising let-bound variables. By restricting these cases to be monomorphic we ensure in turn that any unknown type appearing in the type environment must be explicitly marked as monomorphic.

Principal Type Restriction The Let rule requires that when typing $\mathbf{let}\ x = M\ \mathbf{in}\ N$, the type A' given to M must be principal. Consider the program

$$\mathsf{bad5} = \mathbf{let}\ f = \lambda x.x\ \mathbf{in}\ \lceil f \rceil\ 42$$

On the one hand, if we infer the type $\forall a.a \to a$ for f, then bad5 will fail to type check as we cannot apply a term of polymorphic type (instantiation is only automatic for variables). However, given a traditional declarative type system one might reasonably propose $\mathsf{Int} \to \mathsf{Int}$ as a type for f, in which case bad5 would be typeable — albeit a conventional type inference algorithm would have difficulty inferring a type for it. In order to ensure completeness of our type inference algorithm in the presence of generalisation and freeze, we bake principality into the typing rule for let, similarly to [6, 12, 14, 26]. This means that the only legitimate type that f may be assigned is the most general one, that is $\forall a.a \to a$.

One may think of side-stepping the problem with bad5 by always instantiating terms that appear in application position (after all, it is always a type error for an uninstantiated term of polymorphic type to appear in application position). But then we can exhibit the same problem with a slightly more intricate example.

$$\mathsf{bad6} = \mathbf{let}\ f = \lambda x.x\ \mathbf{in}\ \mathsf{id}\ \lceil f \rceil\ 42$$

The principality condition is also applied in the non-generalising case of the Let rule, meaning that we must instantiate the principal type for M rather than an arbitrary one. Otherwise, we could still type bad4 by assigning *bot bot* type $\forall a.a \to a$. In the Let rule Δ' would be empty, making instantiation a no-op.

Type Variable Scoping A type annotation in FreezeML may contain type variables that is not bound by the annotation. In contrast to many other systems, we do not interpret such variables existentially, but allow binding type variables across different annotations. In an expression $\mathbf{let}\ (x : A) = M\ \mathbf{in}\ N$, we therefore consider the toplevel quantifiers of A bound in M, meaning that they can be used freely in annotations inside M, rather like GHC's scoped type variables [20], However, this is only true for the generalising case, when M is a guarded value. In the absence of generalisation, any polymorphism in the type A originates from M directly (e.g., because M is a frozen variable). Hence, if M is not a guarded value no bound variables of A are bound in M.

Note that given the **let** binding above, where A has the shape $\forall\Delta.H$, there is no ambiguity regarding which of the type variables in Δ result from generalisation and which originate from M itself. If M is a guarded value, its type is guarded, too, and hence all variables in Δ result from generalisation. Conversely, if $M \notin$ GVal, then there is no generalisation at all.

Due to the unambiguity of the binding behaviour in our system with the value restriction, we can define a purely syntax-directed well-formedness judgement for verifying that types in annotations are well-kinded and respect the intended scoping of type-variables. We call this property well-scopedness, and it is a prerequisite for type inference. The corresponding judgement is $\Delta \Vdash M$, checking that in M, the type annotations are well-formed with respect to kind environment Δ (Figure 6). The main subtlety in this judgement is in how Δ grows when we encounter *annotated* let-bindings. For annotated lambdas, we just check that the type annotation is well-formed in Δ but do not add any type variables in Δ. For plain let, we just check well-scopedness recursively. However, for annotated let-bindings, we check that the type annotation A is well-formed, and we check that M is well-scoped *after extending* Δ *with the top-level type variables of* A. This is sensible because in the Let-Ascribe rule, these type variables (present in the type annotation) are introduced into the kind environment when type checking M. In an unannotated let, in contrast, the generalisable type variables are not mentioned in M, so it does not make sense to allow them to be used in other type annotations inside M.

As a concrete example of how this works, consider an explicitly annotated let-binding of the identity function: **let** $(f : \forall a.a \to a) = \lambda(x : a).x$ **in** N, where the a type annotation on x is bound by $\forall a$ in the type annotation on f. However, if we left off the $\forall a.a \to a$ annotation on f, then the a annotation on x would be unbound. This also means that in expressions, we cannot let type annotations α-vary freely; that is, the previous expression is α-equivalent to **let** $(f : \forall b.b \to b) = \lambda(x : b).x$ **in** N but not to **let** $(f : \forall b.b \to b) = \lambda(x : a).x$ **in** N. This behaviour is similar to other proposals for scoped type variables [20].

"Pure" FreezeML In a hypothetical pure version of FreezeML without the value restriction, a purely syntactic check on **let** $(x : A) = M$ **in** N is not sufficient to determine which top-level quantifiers of A are bound in M. In the expression

$$\begin{array}{l} \textbf{let}\ (f : \forall a\, b.a \to b \to b) = \\ \quad \textbf{let}\ (g : \forall b.a \to b \to b) = \lambda y\, z.z\ \textbf{in}\ \mathsf{id}\ \lceil g \rceil \\ \textbf{in}\ N \end{array}$$

the outer **let** generalises a, unlike the subsequent variable b, which arises from the inner **let** binding. The well-scopedness judgement would require typing information. Moreover, the Let-Asc rule would have to nondeterministically split the type annotation A into $\forall\Delta', \Delta''.H$, such that Δ' contains those variables to generalise (a in the example), and Δ'' contains those type variables originating from M directly (b in the example). Similarly, type inference would have to take this splitting into account.

$$\frac{}{\Delta \Vdash \lceil x \rceil} \qquad \frac{}{\Delta \Vdash x} \qquad \frac{\Delta \Vdash M}{\Delta \Vdash \lambda x.M}$$

$$\frac{\Delta \vdash A : \star \quad \Delta \Vdash M}{\Delta \Vdash \lambda(x : A).M} \qquad \frac{\Delta \Vdash M \quad \Delta \Vdash N}{\Delta \Vdash M\, N} \qquad \frac{\Delta \Vdash M \qquad \Delta \Vdash N}{\Delta \Vdash \textbf{let}\ x = M\ \textbf{in}\ N}$$

$$\frac{\Delta \vdash A : \star \quad (\Delta', A') = \mathsf{split}(A, M) \qquad \Delta, \Delta' \Vdash M \qquad \Delta \Vdash N}{\Delta \Vdash \textbf{let}\ (x : A) = M\ \textbf{in}\ N}$$

Figure 6. Well-Scopedness of FreezeML Terms

Instantiation strategies In FreezeML (and indeed ML) the only terms that are implicitly instantiated are variables. Thus (head ids) 42 is ill-typed and we must insert the instantiation operator @ to yield a type-correct expression: (head ids)@ 42. It is possible to extend our approach to perform *eliminator instantiation*, whereby we implicitly instantiate terms appearing in monomorphic elimination position (in particular application position), and thus, for instance, infer a type for bad5 without compromising completeness.

Another possibility is to instantiate all terms, except those that are explicitly frozen or generalised. Here, it also makes sense to extend the $\lceil - \rceil$ operator to act on arbitrary terms, rather than just variables. We call this strategy *pervasive instantiation*. Like eliminator instantiation, pervasive instantiation infers a type for (head ids) 42. However, pervasive instantiation requires inserting explicit generalisation where it was previously unnecessary. Moreover, pervasive instantiation complicates the meta-theory, requiring two mutually recursive typing judgements instead of just one.

The formalism developed in this paper uses variable instantiation alone, but our implementation also supports eliminator instantiation. We defer further theoretical investigation of alternative strategies to future work.

4 Relating System F and FreezeML

In this section we present type-preserving translations mapping System F terms to FreezeML terms and vice versa. We also briefly discuss the equational theory induced on FreezeML by these translations.

$$
\begin{array}{rcl}
\mathcal{E}[\![x]\!] &=& \lceil x \rceil \\
\mathcal{E}[\![\lambda x^A.M]\!] &=& \lambda(x:A).\mathcal{E}[\![M]\!] \\
\mathcal{E}[\![M\ N]\!] &=& \mathcal{E}[\![M]\!]\ \ \mathcal{E}[\![N]\!] \\
\mathcal{E}[\![\Lambda a.V^B]\!] &=& \mathbf{let}\ (x:\forall a.B) = (\mathcal{E}[\![V]\!])@\ \mathbf{in}\ \lceil x \rceil \\
\mathcal{E}[\![M^{\forall a.B}\ A]\!] &=& \mathbf{let}\ (x:B[A/a]) = (\mathcal{E}[\![M]\!])@\ \mathbf{in}\ \lceil x \rceil
\end{array}
$$

Figure 7. Translation from System F to FreezeML

4.1 From System F to FreezeML

Figure 7 defines a translation $\mathcal{E}[\![-]\!]$ of System F terms into FreezeML. The translation depends on types of subterms and is thus formally defined on derivations, but we use a shorthand notation in which subterms are annotated with their type (e.g., in $\Lambda a.V^B$, B indicates the type of V).

Variables are frozen to suppress instantiation. Term abstraction and application are translated homomorphically.

Type abstraction $\Lambda a.V$ is translated using an annotated let-binding to perform the necessary generalisation. However, we cannot bind x to the translation of V directly as only *guarded* values may be generalised but $\mathcal{E}[\![V]\!]$ may be an unguarded value (concretely, a frozen variable). Hence, we bind x to $(\mathcal{E}[\![V]\!])@$, which is syntactic sugar for **let** $y = \mathcal{E}[\![V]\!]$ **in** y. This expression is indeed a guarded value. We then freeze x to prevent immediate instantiation. Type application $M\ A$, where M has type $\forall a.B$, is translated similarly to type abstraction. We bind x to the result of translating M, but only after instantiating it. The variable x is annotated with the intended return type $B[A/a]$ and returned frozen.

Explicit instantiation is strictly necessary and the following, seemingly easier translation is incorrect.

$$\mathcal{E}[\![M^{\forall a.B}\ A]\!] \quad \neq \quad \mathbf{let}\ (x:B[A/a]) = \mathcal{E}[\![M]\!]\ \mathbf{in}\ \lceil x \rceil$$

The term $\mathcal{E}[\![M]\!]$ may be a frozen variable or an application, whose type cannot be implicitly instantiated to type $B[A/a]$.

For any System F value V (i.e., any term other than an application), $\mathcal{E}[\![V]\!]$ yields a FreezeML value (Figure 1).

Each translated term has the same type as the original.

Theorem 2 (Type preservation)**.** *If* $\Delta;\Gamma \vdash M : A$ *in System F then* $\Delta;\Gamma \vdash \mathcal{E}[\![M]\!] : A$ *in FreezeML.*

4.2 From FreezeML to System F

Figure 8 gives the translation of FreezeML to System F. Frozen variables in FreezeML are simply variables in System F. A plain (i.e., not frozen) variable x is translated to a type application $x\ \delta(\Delta')$, where $\delta(\Delta')$ stands for the pointwise application of δ to Δ'. Here, δ and Δ' are obtained from x's type derivation in FreezeML; Δ' contains all top-level quantifiers of $x's$ type. This makes FreezeML's implicit instantiation of non-frozen variables explicit. Lambda abstractions and applications translate directly. Let-bindings in FreezeML are translated as generalised let-bindings in System F where **let** $x^A = M$ **in** N is syntactic sugar for $(\lambda x^A.N)\ M$. Here, generalisation is repeated type abstraction.

Each translated term has the same type as the original.

Theorem 3 (Type preservation)**.** *If* $\Delta;\Gamma \vdash M : A$ *holds in FreezeML then* $\Delta;\Gamma \vdash \mathcal{C}[\![M]\!] : A$ *holds in System F.*

4.3 Equational reasoning

We can derive and verify equational reasoning principles for FreezeML by lifting from System F via the translations. We write $M \simeq N$ to mean M is observationally equivalent to N whenever $\Delta;\Gamma \vdash M : A$ and $\Delta;\Gamma \vdash N : A$. At a minimum we expect β-rules to hold, and indeed they do; the twist is that they involve substituting a different value depending on whether the variable being substituted for is frozen or not.

$$
\begin{array}{lcl}
\mathbf{let}\ x = V\ \mathbf{in}\ N & \simeq & N[\$V\ /\ \lceil x \rceil,\ (\$V)@\ /\ x] \\
\mathbf{let}\ (x:A) = V\ \mathbf{in}\ N & \simeq & N[\$^A V\ /\ \lceil x \rceil,\ (\$^A V)@\ /\ x] \\
(\lambda x.M)\ V & \simeq & M[V\ /\ \lceil x \rceil,\ V@\ /\ x] \\
(\lambda(x:A).M)\ V & \simeq & M[V\ /\ \lceil x \rceil,\ V@\ /\ x]
\end{array}
$$

If we perform type-erasure then these rules degenerate to the standard ones. We can also verify that η-rules hold.

$$
\begin{array}{lcl@{\qquad}lcl}
\mathbf{let}\ x = U\ \mathbf{in}\ x & \simeq & U & \mathbf{let}\ x = \lceil y \rceil\ \mathbf{in}\ x & \simeq & y \\
\mathbf{let}\ (x:A) = U\ \mathbf{in}\ x & \simeq & U & \mathbf{let}\ (x:A) = \lceil y \rceil\ \mathbf{in}\ x & \simeq & y \\
\lambda x.M\,x & \simeq & M & \lambda(x:A).M\,\lceil x \rceil & \simeq & M
\end{array}
$$

5 Type Inference

In this section we present a sound and complete type inference algorithm for FreezeML. The style of presentation is modelled on that of Leijen [12].

5.1 Type Variables and Kinds

When expressing type inference algorithms involving first-class polymorphism, it is crucial to distinguish between object language type variables, and meta language type variables that stand for unknown types required to solve the type inference problem. This distinction is the same as that between *eigenvariables* and *logic variables* in higher-order logic programming [17]. We refer to the former as *rigid* type variables and the latter as *flexible* type variables. For the purposes of the algorithm we will explicitly separate the two by placing them in different kind environments.

As in the rest of the paper, we let Δ range over fixed kind environments in which every type variable is monomorphic (kind $\bullet$). In order to support, for instance, applying a function to a polymorphic argument, we require flexible variables that may be unified with polymorphic types. For this purpose we introduce refined kind environments ranged over by Θ. Type variables in a refined kind environment may be polymorphic (kind $\star$) or monomorphic (kind $\bullet$). In our algorithms we place rigid type variables in a fixed environment Δ and flexible type variables in a refined environment Θ. Refined kind environments (Θ) are given by the following grammar.

$$\mathsf{KEnv} \ni \Theta ::= \cdot \mid \Theta, a : K$$

$$C\left\llbracket \frac{x : A \in \Gamma}{\Delta;\Gamma \vdash \lceil x \rceil : A} \right\rrbracket = x \qquad C\left\llbracket \frac{\Delta;\Gamma, x : S \vdash M : B}{\Delta;\Gamma \vdash \lambda x.M : S \to B} \right\rrbracket = \lambda x^{S}.C\llbracket M \rrbracket \qquad C\left\llbracket \frac{\Delta;\Gamma, x : A \vdash M : B}{\Delta;\Gamma \vdash \lambda(x : A).M : A \to B} \right\rrbracket = \lambda x^{A}.C\llbracket M \rrbracket$$

$$C\left\llbracket \frac{x : \forall\Delta'.H \in \Gamma \qquad \Delta \vdash \delta : \Delta' \Rightarrow_{\star} \cdot}{\Delta;\Gamma \vdash x : \delta(H)} \right\rrbracket = x\ \delta(\Delta') \qquad C\left\llbracket \frac{\Delta;\Gamma \vdash M : A \to B \qquad \Delta;\Gamma \vdash N : A}{\Delta;\Gamma \vdash M\,N : B} \right\rrbracket = C\llbracket M \rrbracket\ C\llbracket N \rrbracket$$

$$C\left\llbracket \frac{\begin{array}{c}(\Delta', \Delta'') = \mathsf{gen}(\Delta, A', M) \qquad (\Delta, \Delta'', M, A') \Updownarrow A \\ \Delta, \Delta''; \Gamma \vdash M : A' \qquad \Delta;\Gamma, x : A \vdash N : B \\ \mathsf{principal}(\Delta, \Gamma, M, \Delta'', A')\end{array}}{\Delta;\Gamma \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : B} \right\rrbracket = \begin{array}{l}\mathbf{let}\ x^{A} = \Lambda\,\Delta'.C\llbracket M \rrbracket \\ \mathbf{in}\ C\llbracket N \rrbracket\end{array} = C\left\llbracket \frac{\begin{array}{c}(\Delta', A') = \mathsf{split}(A, M) \\ \Delta, \Delta'; \Gamma \vdash M : A' \\ \Delta;\Gamma, x : A \vdash N : B\end{array}}{\Delta;\Gamma \vdash \mathbf{let}\ (x : A) = M\ \mathbf{in}\ N : B} \right\rrbracket$$

Figure 8. Translation from FreezeML to System F

$\boxed{\Theta \vdash A : K}$

$$\textsc{Cons}\ \frac{\begin{array}{c}\mathsf{arity}(D) = n \\ \Theta \vdash A_1 : K \\ \cdots \\ \Theta \vdash A_n : K\end{array}}{\Theta \vdash D\,\overline{A} : K}$$

$$\textsc{TyVar}\ \frac{a : K \in \Theta}{\Theta \vdash a : K} \qquad \textsc{ForAll}\ \frac{\Theta, a : \bullet \vdash A : \star}{\Theta \vdash \forall a.A : \star} \qquad \textsc{Upcast}\ \frac{\Theta \vdash A : \bullet}{\Theta \vdash A : \star}$$

$\boxed{\Theta \vdash \Gamma}$

$$\textsc{Empty}\ \frac{}{\Theta \vdash \cdot} \qquad \textsc{Extend}\ \frac{\begin{array}{c}\Theta \vdash \Gamma \qquad \Theta \vdash A : \star \\ (\text{for all } a \in \mathsf{ftv}(A) \mid a : \bullet \in \Theta)\end{array}}{\Theta \vdash \Gamma, x : A}$$

Figure 9. Refined Kinding Rules

We often implicitly treat fixed kind environments $\overline{a}$ as refined kind environments $\overline{a : \bullet}$. The refined kinding rules are given in Figure 9.

The key difference with respect to the object language kinding rules is that type variables can now be polymorphic. Rather than simply defining kinding of type environments point-wise the Extend rule additionally ensures that all type variables appearing in a type environment are monomorphic. This restriction is crucial for avoiding guessing of polymorphism. More importantly, it is also key to ensuring that typing judgements are stable under substitution. Without it it would be possible to substitute monomorphic type variables with types containing nested polymorphic variables, thus introducing polymorphism into a monomorphic type.

We generalise typing judgements $\Delta;\Gamma \vdash M : A$ to $\Theta;\Gamma \vdash M : A$, adopting the convention that $\Theta \vdash \Gamma$ and $\Theta \vdash A$ must hold as preconditions.

5.2 Type Substitutions

In order to define the type inference algorithm we will find it useful to define a judgement for type substitutions θ, which operate on flexible type variables, unlike type instantiations δ, which operate on rigid type variables. The type substitution rules are given in Figure 10. The rules are as in Figure 4,

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

$$\frac{}{\Delta \vdash \emptyset : \cdot \Rightarrow \Theta} \qquad \frac{\Delta \vdash \theta : \Theta' \Rightarrow \Theta \qquad \Delta, \Theta \vdash A : K}{\Delta \vdash \theta[a \mapsto A] : (\Theta', a : K) \Rightarrow \Theta}$$

Figure 10. Type Substitutions

$$\textsc{S-Identity}\ \frac{}{\Delta \vdash \iota_{\Theta} : \Theta \Rightarrow \Theta} \qquad \textsc{S-Weaken}\ \frac{\Delta \vdash \theta : \Theta \Rightarrow \Theta'}{\Delta, \Delta' \vdash \theta : \Theta \Rightarrow \Theta', \Theta''}$$

$$\textsc{S-Compose}\ \frac{\begin{array}{c}\Delta \vdash \theta : \Theta' \Rightarrow \Theta'' \\ \Delta \vdash \theta' : \Theta \Rightarrow \Theta'\end{array}}{\Delta \vdash \theta \circ \theta' : \Theta \Rightarrow \Theta''} \qquad \textsc{S-Strengthen}\ \frac{\begin{array}{c}\Delta \vdash \theta : \Theta \Rightarrow \Theta' \\ \mathsf{ftv}(\theta) \mathbin{\#} \Delta', \Theta''\end{array}}{\Delta - \Delta' \vdash \theta : \Theta \Rightarrow \Theta' - \Theta''}$$

Figure 11. Properties of Substitution

except that the kind environments on the right of the turnstile are refined kind environments and rather than the substitution having a fixed kind, the kind of each type variable must match up with the kind of the type it binds.

We write ι_{Θ} for the identity type substitution on Θ, omitting the subscript when clear from context.

$$\iota_{\cdot} = \emptyset \qquad \iota_{\Theta, a:K} = \iota_{\Theta}[a \mapsto a]$$

Composition of type substitutions is standard.

$$\theta \circ \emptyset = \emptyset \qquad \theta \circ \theta'[a \mapsto A] = (\theta \circ \theta')[a \mapsto \theta(A)]$$

The rules shown in Figure 11 are admissible and we make use of them freely in our algorithms and proofs.

5.3 Unification

A crucial ingredient for type inference is unification. The unification algorithm is defined in Figure 12. It is partial in that it either returns a result or fails. Following Leijen [12] we explicitly indicate the successful return of a result X by writing return X. Failure may be either explicit or implicit (in the case that an auxiliary function is undefined).

```
$\mathsf{unify} : (\mathsf{PEnv} \times \mathsf{KEnv} \times \mathsf{Type} \times \mathsf{Type}) \rightharpoonup (\mathsf{KEnv} \times \mathsf{Subst})$
$\mathsf{unify}(\Delta, \Theta, a, a) =$
  return $(\Theta, \iota)$
$\mathsf{unify}(\Delta, (\Theta, a : K), a, A) =$
  let $\Theta_1 = \mathsf{demote}(K, \Theta, \mathsf{ftv}(A) - \Delta)$
  assert $\Delta, \Theta_1 \vdash A : K$
  return $(\Theta_1, \iota[a \mapsto A])$
$\mathsf{unify}(\Delta, (\Theta, a : K), A, a) =$
  let $\Theta_1 = \mathsf{demote}(K, \Theta, \mathsf{ftv}(A) - \Delta)$
  assert $\Delta, \Theta_1 \vdash A : K$
  return $(\Theta_1, \iota[a \mapsto A])$
$\mathsf{unify}(\Delta, \Theta, D\,\overline{A}, D\,\overline{B}) =$
  let $(\Theta_1, \theta_1) = (\Theta, \iota)$
  let $n = \mathsf{arity}(D)$
  for $i \in 1 \ldots n$
    let $(\Theta_{i+1}, \theta_{i+1}) =$
      let $(\Theta', \theta') = \mathsf{unify}(\Delta, \Theta_i, \theta_i(A_i), \theta_i(B_i))$
      return $(\Theta', \theta' \circ \theta_i)$
  return $(\Theta_{n+1}, \theta_{n+1})$
$\mathsf{unify}(\Delta, \Theta, \forall a.A, \forall b.B) =$
  assume fresh $c$
  let $(\Theta_1, \theta') = \mathsf{unify}((\Delta, c), \Theta, A[c/a], B[c/b])$
  assert $c \notin \mathsf{ftv}(\theta')$
  return $(\Theta_1, \theta')$

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

Figure 12. Unification Algorithm

The algorithm takes a quadruple (Δ, Θ, A, B) of a fixed kind environment Δ, a refined kind environment Θ, and types A and B, such that $\Delta, \Theta \vdash A, B$. It returns a unifier, that is, a pair (Θ', θ) of a new refined kind environment Θ' and a type substitution θ, such that $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$.

A type variable unifies with itself, yielding the identity substitution. Due to the use of explicit kind environments, there is no need for an explicit occurs check to avoid unification of a type variable a with a type A including recursive occurrences of a. Unification of a flexible variable a with a type A implicitly performs an occurs check by checking that the type substituted for a is well-formed in an environment (Δ, Θ_1) that does not contain a. A polymorphic flexible variable unifies with any other type, as is standard. A monomorphic flexible variable only unifies with a type A if A may be *demoted* to a monomorphic type. The auxiliary demote function converts any polymorphic flexible variables in A to monomorphic flexible variables in the refined kind environment. This demotion is sufficient to ensure that further unification cannot subsequently make A polymorphic. Unification of data types is standard, checking that the data type constructors match, and recursing on the substructures. Following Leijen [12], unification of quantified types ensures that forall-bound type variables do not escape their scope by introducing a fresh rigid (skolem) variable and ensuring it does not appear in the free type variables of the substitution.

Theorem 4 (Unification is sound)**.** *If* $\Delta, \Theta \vdash A, B : K$ *and* $\mathsf{unify}(\Delta, \Theta, A, B) = (\Theta', \theta)$ *then* $\theta(A) = \theta(B)$ *and* $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$.

Theorem 5 (Unification is complete and most general)**.** *If* $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$ *and* $\Delta, \Theta \vdash A : K$ *and* $\Delta, \Theta \vdash B : K$ *and* $\theta(A) = \theta(B)$, *then* $\mathsf{unify}(\Delta, \Theta, A, B) = (\Theta'', \theta')$ *where there exists* θ'' *satisfying* $\Delta \vdash \theta'' : \Theta'' \Rightarrow \Theta'$ *such that* $\theta = \theta'' \circ \theta'$.

5.4 The Inference Algorithm

The type inference algorithm is defined in Figure 13. It is partial in that it either returns a result or fails. The algorithm takes a quadruple $(\Delta, \Theta, \Gamma, M)$ of a fixed kind environment Δ, a refined kind environment Θ, a type environment Γ, and a term M, such that $\Delta; \Theta \vdash \Gamma$. If successful, it returns a triple (Θ', θ, A) of a new refined kind environment Θ', a type substitution θ, such that $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$, and a type A such that $\Delta, \Theta' \vdash A : \star$.

The algorithm is an extension of algorithm W [3] adapted to use explicit kind environments Δ, Θ. Inferring the type of a frozen variable is just a matter of looking up its type in the type environment. As usual, the type of a plain (unfrozen) variable is inferred by instantiating any polymorphism with fresh type variables. The returned identity type substitution is weakened accordingly. Crucially, the argument type inferred for an unannotated lambda abstraction is monomorphic. If on the other hand the argument type is annotated with a type, then we just use that type directly. For applications we use the unification algorithm to check that the function and argument match up. Generalisation is performed for unannotated let-bindings in which the let-binding is a guarded value. For unannotated let-bindings in which the let-binding is not a guarded value, generalisation is suppressed and any ungeneralised flexible type variables are demoted to be monomorphic. When a let-binding is annotated with a type then rather than performing generalisation we use the annotation, taking care to account for any polymorphism that is already present in the inferred type for M using split, and checking that none of the quantifiers escape by inspecting the codomain of θ_2.

Theorem 6 (Type inference is sound)**.** *If* $\Delta, \Theta \vdash \Gamma$ *and* $\Delta \Vdash M$ *and* $\mathsf{infer}(\Delta, \Theta, \Gamma, M) = (\Theta', \theta, A)$ *then* $\Delta, \Theta'; \theta(\Gamma) \vdash M : A$ *and* $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$.

```
infer : (PEnv × KEnv × TEnv × Term) ⇀ (KEnv × Subst × Type)
infer($\Delta, \Theta, \Gamma, \lceil x \rceil$) =
  return $(\Theta, \iota, \Gamma(x))$
infer($\Delta, \Theta, \Gamma, x$) =
  let $\forall \overline{a}.H = \Gamma(x)$
  assume fresh $\overline{b}$
  return $((\Theta, \overline{b : \star}), \iota, H[\overline{b}/\overline{a}])$
infer($\Delta, \Theta, \Gamma, \lambda x.M$) =
  assume fresh $a$
  let $(\Theta_1, \theta[a \mapsto S], B)$ = infer($\Delta, (\Theta, a : \bullet), (\Gamma, x : a), M$)
  return $(\Theta_1, \theta, S \to B)$
infer($\Delta, \Theta, \Gamma, \lambda(x : A).M$) =
  let $(\Theta_1, \theta, B)$ = infer($\Delta, \Theta, (\Gamma, x : A), M$)
  return $(\Theta_1, \theta, A \to B)$
infer($\Delta, \Theta, \Gamma, M\ N$) =
  let $(\Theta_1, \theta_1, A')$ = infer($\Delta, \Theta, \Gamma, M$)
  let $(\Theta_2, \theta_2, A)$ = infer($\Delta, \Theta_1, \theta_1(\Gamma), N$)
  assume fresh $b$
  let $(\Theta_3, \theta_3[b \mapsto B])$ = unify($\Delta, (\Theta_2, b : \star), \theta_2(A'), A \to b$)
  return $(\Theta_3, \theta_3 \circ \theta_2 \circ \theta_1, B)$
infer($\Delta, \Theta, \Gamma, \mathbf{let}\ x = M\ \mathbf{in}\ N$) =
  let $(\Theta_1, \theta_1, A)$ = infer($\Delta, \Theta, \Gamma, M$)
  let $\Delta'$ = ftv($\theta_1$) $- \Delta$
  let $(\Delta'', \Delta''')$ = gen($(\Delta, \Delta'), A, M$)
  let $\Theta'_1$ = demote($\bullet, \Theta_1, \Delta'''$)
  let $(\Theta_2, \theta_2, B)$ = infer($\Delta, \Theta'_1 - \Delta'', \theta_1(\Gamma), x : \forall \Delta''.A, N$)
  return $(\Theta_2, \theta_2 \circ \theta_1, B)$
infer($\Delta, \Theta, \Gamma, \mathbf{let}\ (x : A) = M\ \mathbf{in}\ N$) =
  let $(\Delta', A')$ = split($A, M$)
  let $(\Theta_1, \theta_1, A_1)$ = infer($(\Delta, \Delta'), \Theta, \Gamma, M$)
  let $(\Theta_2, \theta'_2)$ = unify($(\Delta, \Delta'), \Theta_1, A', A_1$)
  let $\theta_2 = (\theta'_2 \circ \theta_1)$
  assert ftv($\theta_2$) # $\Delta'$
  let $(\Theta_3, \theta_3, B)$ = infer($\Delta, \Theta_2, (\theta_2(\Gamma), x : A), N$)
  return $(\Theta_3, \theta_3 \circ \theta_2, B)$
```

Figure 13. Type Inference Algorithm

Theorem 7 (Type inference is complete and principal). *Let* $\Delta \Vdash M$ *and* $\Delta, \Theta \vdash \Gamma$. *If* $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$ *and* $\Delta, \Theta'; \theta(\Gamma) \vdash M : A$, *then* infer$(\Delta, \Theta, \Gamma, M) = (\Theta'', \theta', A')$ *where there exists* θ'' *satisfying* $\Delta \vdash \theta'' : \Theta'' \Rightarrow \Theta'$ *such that* $\theta = \theta'' \circ \theta'$ *and* $\theta''(A') = A$.

6 Implementation

We have implemented FreezeML as an extension of Links. This exercise was mostly routine. In the process we addressed several practical concerns and encountered some non-trivial interactions with other features of Links. In order to keep this paper self-contained we avoid concrete Links syntax, but instead illustrate the ideas of the implementation in terms of extensions to the core syntax used in the paper.

In ASCII we render $\lceil x \rceil$ as ~x. For convenience, Links builds in the generalisation $ and instantiation operators @.

In practice (in Links and other functional languages), it is often convenient to include a type signature for a function definition rather than annotations on arguments. Thus

$$\begin{array}{l} f : \forall a.A \to B \to C \\ f\ x\ y = M \\ N \end{array}$$

is treated as:

$$\mathbf{let}\ (f : \forall a.A \to B \to C) = \lambda(x : A).\lambda(y : B).M\ \mathbf{in}\ N$$

Though x and y are not themselves annotated, A and B may be polymorphic, and may mention a.

Given that FreezeML is explicit about the order of quantifiers, adding support for explicit type application [4] is straightforward. We have implemented this feature in Links.

Links has an implicit subkinding system used for various purposes including classifying base types in order to support language-integrated query [15] and distinguishing between linear and non-linear types in order to support session typing [16]. In plain FreezeML, if we have poly : $(\forall a.a \to a) \to$ Int × Bool and id : $\forall a.a \to a$, then we may write poly $\lceil$id$\rceil$. The equivalent in Links also works. However, the type inferred for the identity function in Links is not $\forall a.a \to a$, but rather $\forall (a : \circ).a \to a$, where the subkinding constraint $\circ$ captures the property that the argument is used linearly. Given this more refined type for id the term poly $\lceil$id$\rceil$ no longer type-checks. In this particular case one might imagine generating an implicit coercion (a function that promises to uses its argument linearly may be soundly treated as a function that may or may not use its argument linearly). In general one has to be careful to be explicit about the kinds of type variables when working with first-class polymorphism. Similar issues arise from the interaction between first-class polymorphism and Links's effect type system [15].

Existing infrastructure for subkinding in the implementation of Links was helpful for adding support for FreezeML as we exploit it for tracking the monomorphism / polymorphism distinction. However, there is a further subtlety: in FreezeML type variables of monomorphic kind may be instantiated with (though not unified with) polymorphic types; this behaviour differs from that of other kinds in Links.

The Links source language allows the programmer to explicitly distinguish between rigid and flexible type variables. Flexible type variables can be convenient to use as wildcards during type inference. As a result, type annotations in Links are slightly richer than those admitted by the well-scopedness judgement of Figure 6. It remains to verify the formal properties of the richer system.

7 Related Work

There are many previous attempts to bridge the gap between ML and System F. Some systems employ more expressive types than those of System F; others implement heuristics in the type system to achieve a balance between increased complexity of the system and reducing the number of necessary type annotations; finally, there are systems like ours that eschew such heuristics for the sake of simplifying the type system further. Users then have to state their intentions explicitly, potentially resulting in more verbose programs.

Expressive Types MLF [10] (sometimes stylised as ML$^\mathrm{F}$) is considered to be the most expressive of the conservative ML extensions so far. MLF achieves its expressiveness by going beyond regular System F types and introducing polymorphically bounded types, though translation from MLF to System F and vice versa remains possible [10, 11]. MLF also extends ML with type annotations on lambda binders. Annotations on binders that are *used* polymorphically are mandatory, since type inference will not guess second-order types. This is required to maintain principal types.

HML [13] is a simplification of MLF. In HML all polymorphic function arguments require annotations. It significantly simplifies the type inference algorithm compared to MLF, though polymorphically bounded types are still used.

Heuristics HMF [12] contrasts with the above systems in that it only uses regular System F types (disregarding order of quantifiers). Like FreezeML, it only allows principal types for let-bound variables, and type annotations are needed on all polymorphic function parameters. HMF allows both instantiation and generalisation in argument positions, taking n-ary applications into account. The system uses weights to select between less and more polymorphic types. Whole lambda abstractions require an annotation to have a polymorphic return type. Such term annotations are *rigid*, meaning they suppress instantiation and generalisation. As instantiation is implicit in HMF, rigid annotations can be seen as a means to freeze arbitrary expressions.

Several systems for first-class polymorphism were proposed in the context of the Haskell programming language. These systems include boxy types [26], FPH [27], and GI [24]. The Boxy Types system, used to implement GHC's `ImpredicativeTypes` extension, was very fragile and thus difficult to use in practice. Similarly, the FPH system – based on MLF – was simpler but still difficult to implement in practice. GI is the latest development in this line of research. Its key ingredient is a heuristic that restricts polymorphic instantiation, based on whether a variable occurs under a type constructor and argument types in an application. Like HMF, it uses System F types, considers n-ary applications for typing, and requires annotations both for polymorphic parameter and return types. However, only top-level type variables may be re-ordered. The authors show how to combine their system with the OutsideIn(X) [25] constraint-solving type inference algorithm used by the Glasgow Haskell Compiler. They also report a prototype implementation of GI as an extension to GHC with encouraging experience porting existing Hackage packages that use rank-n polymorphism.

Explicitness Some early work on first-class polymorphism was based on the observation that polymorphism can be encapsulated inside nominal types [8, 9, 19, 22].

The QML [23] system explicitly distinguishes between polymorphic schemes and quantified types and hence does not use plain System F types. Type schemes are used for ML let-polymorphism and introduced and eliminated implicitly. Quantified types are used for first-class polymorphism, in particular for polymorphic function arguments. Such types must always be introduced and eliminated explicitly, which requires stating the full type and not just instantiating the type variables. All polymorphic instantiations must therefore be made explicitly by annotating terms at call sites. Neither **let**- nor λ-bound variables can be annotated with a type.

Poly-ML [6] is similar to QML in that it distinguishes two incompatible sorts of polymorphic types. Type schemes arise from standard ML generalisation; (boxed) polymorphic types are introduced using a dedicated syntactic form which requires a type annotation. Boxed polymorphic types are considered to be simple types, meaning that a type variable can be instantiated with a boxed polymorphic type, but not with a type scheme. Terms of a boxed type are not instantiated implicitly, but must be opened explicitly, resulting in instantiation. Unlike QML, the instantiated type is deduced from the context, rather than requiring an annotation.

Unlike FreezeML, Poly-ML supports inferring polymorphic parameter types for unannotated lambdas, but this is limited to situations where the type is unambiguously determined by the context. This is achieved by using *labels*, which track whether polymorphism was guessed or confirmed by a type annotation. Whereas FreezeML has type annotations on binders, Poly-ML has type annotations on terms and propagates them using the label system.

In Poly-ML, the example $\lambda x.\mathrm{auto}\ x$ typechecks, guessing a polymorphic type for x; FreezeML requires a type annotation on x. In FreezeML the program $\mathbf{let}\ id = \lambda x.x\ \mathbf{in}\ \mathbf{let}\ c = id\ 3\ \mathbf{in}\ \mathrm{auto}\ \lceil id \rceil$ typechecks, whereas in Poly-ML a type annotation is required (in order to convert between $\forall a.a \to a$ and $[\forall a.a \to a]$). However, Poly-ML could be extended with a new construct for introducing boxed polymorphism without a type annotation, using the principal type instead. With such a change it is possible to translate from FreezeML into this modified version of Poly-ML without inserting any new type annotations (see Appendix D).

Appendix B contains an example-based comparison of FreezeML, GI, MLF, HMF, FPH, and HML.

8 Conclusions

In this paper, we have introduced FreezeML as an exercise in language design for reconciling ML type inference with System F-style first-class polymorphism. We have also implemented FreezeML as part of the Links programming language [2], which uses a variant of Hindley-Milner type inference extended with row types, and has a kind system readily adapted to check that inferred function arguments are monotypes.

Directions for future work include extending FreezeML to accommodate features such as higher-kinds, GADTs, and dependent types, as well as exploring different implicit instantiation strategies. It would also be instructive to rework our formal account using the methodology of Gundry et al. [7] and use that as the basis for mechanised soundness and completeness proofs.

References

[1] Dominique Clément, Joëlle Despeyroux, Th. Despeyroux, and Gilles Kahn. 1986. A Simple Applicative Language: Mini-ML. In *LISP and Functional Programming*. 13–27.

[2] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2006. Links: Web Programming Without Tiers. In *FMCO (Lecture Notes in Computer Science)*, Vol. 4709. Springer, 266–296. http://links-lang.org/

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

[4] Richard A. Eisenberg, Stephanie Weirich, and Hamidhasan G. Ahmed. 2016. Visible Type Application. In *ESOP (Lecture Notes in Computer Science)*, Vol. 9632. Springer, 229–254.

[5] Matthias Felleisen. 1991. On the Expressive Power of Programming Languages. *Sci. Comput. Program.* 17, 1-3 (1991), 35–75.

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

[7] Adam Gundry, Conor McBride, and James McKinna. 2010. Type Inference in Context. In *MSFP@ICFP*. ACM, 43–54.

[8] Mark P. Jones. 1997. First-class Polymorphism with Type Inference. In *POPL*. ACM Press, 483–496.

[9] Konstantin Läufer and Martin Odersky. 1994. Polymorphic Type Inference and Abstract Data Types. *ACM Trans. Program. Lang. Syst.* 16, 5 (1994), 1411–1430.

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

[11] Daan Leijen. 2007. A type directed translation of MLF to system F. In *ICFP*. ACM, 111–122.

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

[13] Daan Leijen. 2009. Flexible types: robust type inference for first-class polymorphism. In *POPL*. ACM, 66–77.

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

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

[16] Sam Lindley and J Garrett Morris. 2017. Lightweight Functional Session Types. In *Behavioural Types: from Theory to Tools*. River Publishers, 265–286.

[17] Dale Miller. 1992. Unification Under a Mixed Prefix. *J. Symb. Comput.* 14, 4 (1992), 321–358. https://doi.org/10.1016/0747-7171(92)90011-R

[18] Robin Milner, Mads Tofte, and David Macqueen. 1997. *The Definition of Standard ML (Revised)*. MIT Press.

[19] Martin Odersky and Konstantin Läufer. 1996. Putting Type Annotations to Work. In *POPL*. ACM Press, 54–67.

[20] Simon Peyton Jones and Mark Shields. 2002. Lexically scoped type variables. Unpublished. https://www.microsoft.com/en-us/research/publication/lexically-scoped-type-variables/.

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

[22] Didier Rémy. 1994. Programming Objects with ML-ART, an Extension to ML with Abstract and Record Types. In *TACS (Lecture Notes in Computer Science)*, Vol. 789. Springer, 321–346.

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

[24] Alejandro Serrano, Jurriaan Hage, Dimitrios Vytiniotis, and Simon Peyton Jones. 2018. Guarded impredicative polymorphism. In *PLDI*. ACM, 783–796.

[25] 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.

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

[27] Dimitrios Vytiniotis, Stephanie Weirich, and Simon L. Peyton Jones. 2008. FPH: first-class polymorphism for Haskell. In *ICFP*. ACM, 295–306.

[28] Philip Wadler. 1990. Recursive types for free! Unpublished. Revised 2008. https://homepages.inf.ed.ac.uk/wadler/papers/free-rectypes/free-rectypes.txt.

[29] 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.

[30] Andrew K. Wright. 1995. Simple Imperative Polymorphism. *Lisp and Symbolic Computation* 8, 4 (1995), 343–355.

A FreezeML by Example

A	POLYMORPHIC INSTANTIATION	
A1	$\lambda x\ y.y$	$a \to b \to b$
A1•	$\$(\lambda x\ y.y)$	$\forall a\ b.a \to b \to b$
A2	choose id	$(a \to a) \to (a \to a)$
A2•	choose $\lceil$id$\rceil$	$(\forall a.a \to a) \to (\forall a.a \to a)$
A3	choose [] ids	List $(\forall a.a \to a)$
A4	$\lambda(x : \forall a.a \to a).x\ x$	$(\forall a.a \to a) \to (b \to b)$
A4•	$\lambda(x : \forall a.a \to a).x\ \lceil x \rceil$	$(\forall a.a \to a) \to (\forall a.a \to a)$
A5	id auto	$(\forall a.a \to a) \to (\forall a.a \to a)$
A6	id auto$'$	$(\forall a.a \to a) \to (b \to b)$
A6•	id $\lceil$auto$'\rceil$	$\forall b.(\forall a.a \to a) \to (b \to b)$
A7	choose id auto	$(\forall a.a \to a) \to (\forall a.a \to a)$
A8	choose id auto$'$	$\times$
A9★	f (choose $\lceil$id$\rceil$) ids	$\forall a.a \to a$
	where $f : \forall a.(a \to a) \to$ List $a \to a$	
A10★	poly $\lceil$id$\rceil$	Int $\times$ Bool
A11★	poly $\$(\lambda x.x)$	Int $\times$ Bool
A12★	id poly $\$(\lambda x.x)$	Int $\times$ Bool
C	FUNCTIONS ON POLYMORPHIC LISTS	
C1	length ids	Int
C2	tail ids	List $(\forall a.a \to a)$
C3	head ids	$\forall a.a \to a$
C4	single id	List $(a \to a)$
C4•	single $\lceil$id$\rceil$	List $(\forall a.a \to a)$
C5★	$\lceil$id$\rceil$ :: ids	List $(\forall a.a \to a)$
C6★	$\$(\lambda x.x)$ :: ids	List $(\forall a.a \to a)$
C7	(single inc) ++ (single id)	List (Int $\to$ Int)
C8★	g (single $\lceil$id$\rceil$) ids	$\forall a.a \to a$
	where g : $\forall a.$List $a \to$ List $a \to a$	
C9★	map poly (single $\lceil$id$\rceil$)	List (Int $\times$ Bool)
C10	map head (single ids)	List $(\forall a.a \to a)$
B	INFERENCE WITH POLYMORPHIC ARGUMENTS	
B1★	$\lambda(f : \forall a.a \to a).$ $(f\ 1, f$ True$)$	$(\forall a.a \to a) \to$ Int $\times$ Bool
B2★	$\lambda(xs :$ List $(\forall a.a \to a))$. poly (head xs)	List $(\forall a.a \to a) \to$ Int $\times$ Bool
D	APPLICATION FUNCTIONS	
D1★	app poly $\lceil$id$\rceil$	Int $\times$ Bool
D2★	revapp $\lceil$id$\rceil$ poly	Int $\times$ Bool
D3★	runST $\lceil$argST$\rceil$	Int
D4★	app runST $\lceil$argST$\rceil$	Int
D5★	revapp $\lceil$argST$\rceil$ runST	Int
E	η-EXPANSION	
E1	$k\ h\ l$	$\times$
E2★	$k\ \$(\lambda x.(h\ x)@)\ l$	$\forall a.$Int $\to a \to a$
	where $k : \forall a.a \to$ List $a \to a$; $h :$ Int $\to \forall a.a \to a$; $l :$ List $(\forall a.$Int $\to a \to a)$	
E3	$r\ (\lambda x\ y.y)$	$\times$
E3•	$r\ \$(\lambda x.\$(\lambda y.y))$	Int
	where $r : (\forall a.a \to \forall b.b \to b) \to$ Int	
F	FreezeML PROGRAMS	
F1	id $x = x$	$\forall a.a \to a$
F2	ids = [$\lceil$id$\rceil$]	List $(\forall a.a \to a)$
F3	auto $x = x\ \lceil x \rceil$	$(\forall a.a \to a) \to (\forall a.a \to a)$
F4	auto$'$ $x = x\ x$	$\forall b.(\forall a.a \to a) \to b \to b$
F5★	auto $\lceil$id$\rceil$	$\forall a.a \to a$
F6	(head ids) :: ids	List $(\forall a.a \to a)$
F7★	(head ids)@ 3	Int
F8	choose (head ids)	$(\forall a.a \to a) \to (\forall a.a \to a)$
F8•	choose (head ids)@	$(a \to a) \to (a \to a)$
F9	**let** f = revapp $\lceil$id$\rceil$ **in** f poly	Int $\times$ Bool
F10	choose id $(\lambda(x : \forall a.a \to a).\$($auto$'\ x))$	$(\forall a.a \to a) \to (\forall a.a \to a)$

Figure 14. Example FreezeML Terms and Types

head : $\forall a.$List $a \to a$	id : $\forall a.a \to a$	map : $\forall a\ b.(a \to b) \to$ List $a \to$ List b
tail : $\forall a.$List $a \to$ List a	ids : $[\forall a.a \to a]$	app : $\forall a\ b.(a \to b) \to a \to b$
[ ] : $\forall a.$List a	inc : Int $\to$ Int	revapp : $\forall a\ b.a \to (a \to b) \to b$
(::) : $\forall a.a \to$ List $a \to$ List a	choose : $\forall a.a \to a \to a$	runST : $\forall a.(\forall s.ST\ s\ a) \to a$
single : $\forall a.a \to$ List a	poly : $(\forall a.a \to a) \to$ Int $\times$ Bool	argST : $\forall s.ST\ s$ Int
(++) : $\forall a.$List $a \to$ List $a \to$ List a	auto : $(\forall a.a \to a) \to (\forall a.a \to a)$	pair : $\forall a\ b.a \to b \to a \times b$
length : $\forall a.$List $a \to$ Int	auto$'$: $\forall b.(\forall a.a \to a) \to (b \to b)$	pair$'$: $\forall b\ a.a \to b \to a \times b$

Figure 15. Type signatures for functions used in the text; adapted from [24].

Figure 14 presents a collection of FreezeML examples that showcase how our system works in practice. We use functions with type signatures shown in Figure 15 (adapted from Serrano et al. [24]). In Figure 14 well-typed expressions are annotated with a type inferred in FreezeML, whilst ill-typed expressions are annotated with $\times$. Sections A-E of the table are taken from [24]. Section F of the table contains additional examples which further highlight the behaviour of our system. In FreezeML it is sometimes possible to infer a different type depending on the presence of freeze, generalisation, and instantiation operators. In such cases we provide two copies of an example in Figure 14, the one with extra FreezeML annotations being marked with •. Sometimes annotations are mandatory to make an expression well-typed in FreezeML. In such cases there is only one, well-typed copy of an example marked with a ★, e.g. A9★.

B FreezeML vs. Other Systems

In this appendix we present an example-based comparison of FreezeML with other systems for first-class polymorphism: GI [24], MLF [10], HMF [12], FPH [27], and HML [13]. Sections A-E of Figure 14 have been presented in [24], together with analysis of how the five systems behave for these examples. We now use these examples to compare FreezeML with other systems.

Firstly, we focus on which examples can be typechecked without explicit type annotations. (We do not count FreezeML freezes, generalisations, and instantiations as annotations, since these are mandatory in our system by design and they do not require spelling out a type explicitly, allowing the programmer to rely on type inference.) Out of 32 examples presented in Sections A-E of the Figure 14, MLF typechecks all but B1 and E1, placing it first in terms of expressiveness. HML ranks second, being unable to typecheck B1, B2 and E1[3]. FreezeML handles all examples except for A8, B1, B2, and E1, ranking third. FPH, GI, and HMF fail to typecheck 6 examples, 8 examples, and 11 examples respectively. If we permit annotations on binders only, the number of failures for most systems decreases by 2, because the systems can now typecheck Examples B1 and B2. MLF was already able to typecheck B2 without an annotation, so now it handles all but E1. If we permit type annotations on arbitrary terms the number of examples that cannot be typechecked becomes: MLF – 1 (E1), FreezeML– 2 (A8, E1) – GI and HML – 2 (E1, E3), FPH – 4, and HMF – 6. These observations are summarised in Table 1 below.

Table 1. Summary of the number of examples not handled by each system

Annotate?	MLF	HML	FreezeML	FPH	GI	HMF
Nothing	2	3	4	6	8	11
Binders	1	2	2	4	6	6
Terms	1	2	2	4	2	6

Due to FreezeML's approach of explicitly annotating polymorphic instantiations, we might require $\lceil - \rceil$, \$, and @ annotations where other systems need no annotations whatsoever. This is especially the case for Examples A10-12, which all other five systems can handle without annotations. We are being more verbose here, but the additional ink required is minimal and we see this as a fair price for the benefits our system provides. Also, being explicit about generalisations allows us to be precise about the location of quantifiers in a type. This allows us to typecheck Example E3, which no other system except MLF can do.

FreezeML is incapable of typechecking A8, under the assumption that the only allowed modifications are insertions of freeze, generalisation, and instantiation. We can however η-expand and rewrite A8 to F10.

When dealing with n-ary function applications, FreezeML is insensitive to the order of arguments. Therefore, if an application $M\ N$ is well-typed then so are app $M\ N$ and revapp $N\ M$, as shown in section D of the table. Many systems in the literature also enjoy this property, but there are exceptions such as Boxy Types [26].

C Specifications of Core Calculi

In this appendix we provide full specification of two core calculi on which we base FreezeML— call-by-value System F and ML — as well as translation from ML to System F.

C.1 Call-by-value System F

We begin with a standard call-by-value variant of System F. The syntax of System F types, environments, and terms is given in Figure 16.

We let a, b, c range over type variables. We assume a collection of type constructors D each of which has a fixed arity $\mathsf{arity}(D)$. Types formed by type constructor application include base types (Int and Bool), lists of elements of type A ($\mathsf{List}\,A$), and functions from A to B ($A \to B$). Data types may be Church-encoded using polymorphic functions [28], but for the purposes of our examples we treat them specially. Types comprise type variables (a), fully-applied type constructors ($D\,\overline{A}$), and polymorphic types ($\forall a.A$). Type environments track the types of term variables in a term. Kind environments track the type

[3]Table presented in [24] claims that HML cannot typecheck E3 but Didier Rémy pointed out to us in private correspondence that this is not the case and HML can indeed typecheck E3.

$$
\begin{array}{ll}
\text{Type Variables} & a, b, c \\
\text{Type Constructors} & D ::= \mathsf{Int} \mid \mathsf{Bool} \mid \mathsf{List} \mid \to \mid \times \mid \ldots \\
\text{Types} & A, B ::= a \mid D\,\overline{A} \mid \forall a.A \\
\text{Term Variables} & x, y, z \\
\text{Terms} & M, N ::= x \mid \lambda x^A.M \mid M\,N \mid \Lambda a.V \mid M\,A \\
\text{Values} & V, W ::= I \mid \lambda x^A.M \mid \Lambda a.V \\
\text{Instantiations} & I ::= x \mid I\,A \\
\text{Kind Environments} & \Delta ::= \cdot \mid \Delta, a \\
\text{Type Environments} & \Gamma ::= \cdot \mid \Gamma, x : A
\end{array}
$$

Figure 16. System F Syntax

$$
\boxed{\Delta \vdash A : \star}
\qquad
\frac{a \in \Delta}{\Delta \vdash a : \star}
\qquad
\frac{\mathsf{arity}(D) = n \qquad \Delta \vdash A_1 : \star \ \cdots \ \Delta \vdash A_n : \star}{\Delta \vdash D\,\overline{A} : \star}
\qquad
\frac{\Delta, a \vdash A : \star}{\Delta \vdash \forall a.A : \star}
$$

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

$$
\textsc{F-Var}\ \frac{x : A \in \Gamma}{\Delta; \Gamma \vdash x : A}
\qquad
\textsc{F-App}\ \frac{\Delta; \Gamma \vdash M : A \to B \qquad \Delta; \Gamma \vdash N : A}{\Delta; \Gamma \vdash M\,N : B}
\qquad
\textsc{F-PolyLam}\ \frac{\Delta, a; \Gamma \vdash V : A}{\Delta; \Gamma \vdash \Lambda a.V : \forall a.A}
$$

$$
\textsc{F-Lam}\ \frac{\Delta; \Gamma, x : A \vdash M : B}{\Delta; \Gamma \vdash \lambda x^A.M : A \to B}
\qquad
\textsc{F-PolyApp}\ \frac{\Delta; \Gamma \vdash M : \forall a.B \qquad \Delta \vdash A : \star}{\Delta; \Gamma \vdash M\,A : B[A/a]}
$$

Figure 17. System F Kinding and Typing Rules

β-rules

$$
\begin{array}{rcl}
(\lambda x^A.V)\,W & \simeq & V[W/x] \\
(\Lambda a.V)\,A & \simeq & V[A/a]
\end{array}
$$

η-rules

$$
\begin{array}{rcl}
\lambda x^A.M\,x & \simeq & M \\
\Lambda a.V\,a & \simeq & V
\end{array}
$$

Figure 18. System F Equational Rules

variables in a term. For the calculi we present in this section, we only have a single kind, $\star$, the kind of all types, which we omit. Nevertheless, kind environments are still useful for explicitly tracking which type variables are in scope, and when we consider type inference (Section 5) we will need a refined kind system in order to distinguish between monomorphic and polymorphic types.

We let x, y, z range over term variables. Terms comprise variables (x), term abstractions ($\lambda x^A.M$), term applications ($M\,N$), type abstractions ($\Lambda a.V$), and type applications ($M\,A$). We write **let** $x^A = M$ **in** N as syntactic sugar for $(\lambda x^A.N)\,M$, we write $M\,\overline{A}$ as syntactic sugar for repeated type application $M\,A_1 \cdots A_n$, and $\Lambda\overline{a}.V$ as syntactic sugar for repeated type abstraction $\Lambda a_1. \cdots \Lambda a_n.V$. We also may write $\Lambda\Delta.A$ when $\Delta = \overline{a}$. We restrict the body of type abstractions to be syntactic values in accordance with the ML value restriction [30].

Well-formedness of types and the typing rules for System F are given in Figure 17. Standard equational rules (β) and (η) for System F are given in Figure 18.

C.2 ML

We now outline a core fragment of ML. The syntax is given in Figure 19, well-formedness of types and the typing rules in Figure 20. Unlike in System F we here separate monomorphic types (S, T) from type schemes (P, Q) and there is no explicit provision for type abstraction or type application. Instead, only variables may be polymorphic and polymorphism is introduced by generalising the body of a let-binding (ML-Let), and eliminated implicitly when using a variable (ML-Var).

Instantiation applies a type instantiation to the monomorphic body of a polymorphic type. The rules for type instantiations are given in Figure 20. The judgement $\Delta \vdash \delta : \Delta' \Rightarrow \Delta''$ defines a well-formed finite map from type variables in Δ, Δ' into type variables in Δ, Δ'', such that $\delta(a) = a$ for every $a \in \Delta$. As such, it is only well-defined if Δ and Δ' are disjoint and Δ and Δ'' are disjoint. We may apply type instantiations to types and type schemes in the standard way:

Type Variables	a, b, c
Type Constructors	$D ::= \mathsf{Int} \mid \mathsf{Bool} \mid \mathsf{List} \mid \to \mid \times \mid \ldots$
Monotypes	$S, T ::= a \mid D\,\overline{S}$
Type Schemes	$P, Q ::= \forall \overline{a}.S$
Type Instantiations	$\delta ::= \emptyset \mid \delta[a \mapsto S]$
Term Variables	x, y, z
Terms	$M, N ::= x \mid \lambda x.M \mid M\,N \mid \mathbf{let}\ x = M\ \mathbf{in}\ N$
Values	$V, W ::= x \mid \lambda x.M \mid \mathbf{let}\ x = V\ \mathbf{in}\ W$
Kinds	$K ::= \bullet \mid \star$
Kind Environments	$\Delta ::= \cdot \mid \Delta, a$
Type Environments	$\Gamma ::= \cdot \mid \Gamma, x : P$

Figure 19. ML Syntax

$\boxed{\Delta \vdash S : \bullet}$ $\boxed{\Delta \vdash P : \star}$

$$\frac{a \in \Delta}{\Delta \vdash a : \bullet} \qquad \frac{\Delta, \Delta' \vdash S : \bullet}{\Delta \vdash \forall \Delta'.S : \star} \qquad \frac{\mathsf{arity}(D) = n \quad \Delta \vdash S_1 : \bullet \quad \cdots \quad \Delta \vdash S_n : \bullet}{\Delta \vdash D\,\overline{S} : \bullet}$$

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

$$\textsc{ML-Var}\ \frac{x : \forall \Delta'.S \in \Gamma \qquad \Delta \vdash \delta : \Delta' \Rightarrow \cdot}{\Delta; \Gamma \vdash x : \delta(S)} \qquad \textsc{ML-Lam}\ \frac{\Delta; \Gamma, x : S \vdash M : T}{\Delta; \Gamma \vdash \lambda x.M : S \to T}$$

$$\textsc{ML-App}\ \frac{\Delta; \Gamma \vdash M : S \to T \qquad \Delta; \Gamma \vdash N : S}{\Delta; \Gamma \vdash M\,N : T} \qquad \textsc{ML-Let}\ \frac{\Delta' = \mathsf{gen}(\Delta, S, M) \quad \Delta, \Delta'; \Gamma \vdash M : S \quad P = \forall \Delta'.S \quad \Delta; \Gamma, x : P \vdash N : T}{\Delta; \Gamma \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : T}$$

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

$$\frac{}{\Delta \vdash \emptyset : \cdot \Rightarrow \Delta'} \qquad \frac{\Delta \vdash \delta : \Delta' \Rightarrow \Delta'' \qquad \Delta, \Delta'' \vdash S : \bullet}{\Delta \vdash \delta[a \mapsto S] : (\Delta', a) \Rightarrow \Delta''}$$

$$\mathsf{gen}(\Delta, S, M) = \begin{cases} \mathsf{ftv}(S) - \Delta & \text{if } M \text{ is a value} \\ \cdot & \text{otherwise} \end{cases}$$

Figure 20. ML Kinding and Typing Rules

$$\begin{array}{ll} \emptyset(S) = S & \delta[a \mapsto S](a) = S \\ \delta(D\,\overline{S}) = D\,(\overline{\delta(S)}) & \delta[a \mapsto S](b) = \delta(b) \end{array}$$

Generalisation is defined at the bottom of Figure 20. If M is a value, the generalisation operation $\mathsf{gen}(\Delta, S, M)$ returns the list of type variables in S that do not occur in the kind environment Δ, in the order in which they occur, with no duplicates. To satisfy the value restriction, $\mathsf{gen}(\Delta, S, M)$ is empty if M is not a value.

A crucial difference between System F and ML is that in System F the order in which quantifiers appear is important ($\forall a\,b.A$ and $\forall b\,a.A$ are different types), whereas in ML, because instantiation is implicit, the order does not matter. As we are concerned with bridging the gap between the two we will develop an extension of ML in which the order of quantifiers is important. However, this change will not affect the behaviour of type inference for ML terms since the order of quantifiers is lost when polymorphic variable types are instantiated, as in rule ML-Var.

$$\mathcal{C}\left[\!\!\left[\frac{\begin{array}{c}x : \forall\Delta'.S \in \Gamma\\ \Delta \vdash \delta : \Delta' \Rightarrow \cdot\end{array}}{\Delta;\Gamma \vdash x : \delta(S)}\right]\!\!\right] = x\ \delta(\Delta')$$

$$\mathcal{C}\left[\!\!\left[\frac{\begin{array}{c}\Delta' = \mathrm{gen}(\Delta, S, M)\\ \Delta,\Delta';\Gamma \vdash M : S\\ P = \forall\Delta'.S\\ \Delta;\Gamma, x : P \vdash N : T\end{array}}{\Delta;\Gamma \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : T}\right]\!\!\right] = \begin{array}{l}\mathbf{let}\ x^{\forall\Delta'.S} = \Lambda\,\Delta'.\mathcal{C}[\![M]\!]\\ \mathbf{in}\ \mathcal{C}[\![N]\!]\end{array}$$

$$\mathcal{C}\left[\!\!\left[\frac{\Delta;\Gamma, x : S \vdash M : T}{\Delta;\Gamma \vdash \lambda x.M : S \to T}\right]\!\!\right] = \lambda x^{S}.\mathcal{C}[\![M]\!]$$

$$\mathcal{C}\left[\!\!\left[\frac{\begin{array}{c}\Delta;\Gamma \vdash M : S \to T\\ \Delta;\Gamma \vdash N : S\end{array}}{\Delta;\Gamma \vdash M\,N : T}\right]\!\!\right] = \mathcal{C}[\![M]\!]\ \mathcal{C}[\![N]\!]$$

Figure 21. Translation from ML to System F

C.3 ML as System F

ML is remarkable in providing statically typed polymorphism without the programmer having to write any type annotations. In order to achieve this coincidence of features the type system is carefully constructed, and crucial operations (instantiation and generalisation) are left implicit (i.e., not written as explicit constructs in the program). This is convenient for programmers, but less so for metatheoretical study.

In order to explicate ML's polymorphic type system, let us consider a translation of ML into System F. Such a translation is given in Figure 21. As the translation depends on type information not available in terms, formally it is defined as a translation from derivations to terms (rather than terms to terms). But we abuse notation in the standard way to avoid explicitly writing derivation trees everywhere. Each recursive invocation on a subterm is syntactic sugar for invoking the translation on the corresponding part of the derivation.

The translation of variables introduces repeated type applications. Recall that we use $\mathbf{let}\ x^A\ =\ M\ \mathbf{in}\ N$ as syntactic sugar for $(\lambda x^A.N)\ M$ in System F. Translating the let binding of a value then yields repeated type abstractions. For non-values M, Δ' is empty.

Theorem 8. *If* $\Delta;\Gamma \vdash M : S$ *then* $\Delta;\Gamma \vdash \mathcal{C}[\![M]\!] : S$.

The fragment of System F in the image of the translation is quite restricted in that type abstractions are always immediately bound to variables and type applications are only performed on variables. Furthermore, all quantification must be top-level. Next we will extend ML in such a way that the translation can also be extended to cover the whole of System F.

D Example Translation from FreezeML to System F

Below is an example translation from FreezeML to System F, where app, auto, and id have the types given in Figure 15.

$$\begin{array}{l}
\quad\mathcal{C}[\![\mathbf{let}\ \mathsf{app} = \lambda f.\lambda z.f\ z\ \mathbf{in}\ \mathsf{app}\ \lceil\mathsf{auto}\rceil\ \lceil\mathsf{id}\rceil]\!]\\
= \mathbf{let}\ \mathsf{app}^{\forall a\,b.(a\to b)\to a\to b} =\\
\qquad\qquad \Lambda a\,b.\mathcal{C}[\![\lambda f.\lambda z.f\ z]\!]\ \mathbf{in}\ \mathcal{C}[\![\mathsf{app}\ \lceil\mathsf{auto}\rceil\ \lceil\mathsf{id}\rceil]\!]\\
= (\lambda\mathsf{app}^{\forall a\,b.(a\to b)\to a\to b}.\\
\qquad\qquad \mathcal{C}[\![\mathsf{app}\ \lceil\mathsf{auto}\rceil\ \lceil\mathsf{id}\rceil]\!])\ (\Lambda a\,b.\mathcal{C}[\![\lambda f.\lambda z.f\ z]\!])\\
= (\lambda\mathsf{app}^{\forall a\,b.(a\to b)\to a\to b}.\\
\qquad\qquad \mathcal{C}[\![\mathsf{app}\ \lceil\mathsf{auto}\rceil\ \lceil\mathsf{id}\rceil]\!])\ (\Lambda a\,b.\lambda f^{a\to b}.\lambda z^{a}.f\ z)
\end{array}$$

where subterm $C[\![\text{app}\ \lceil\text{auto}\rceil\ \lceil\text{id}\rceil]\!]$ further translates as:

$$\begin{aligned} C[\![\text{app}\ \lceil\text{auto}\rceil\ \lceil\text{id}\rceil]\!] = \text{app}\,&((\forall a.a \to a) \to (\forall a.a \to a)) \\ &(\forall a.a \to a) \\ &\text{auto} \\ &\text{id} \end{aligned}$$

The type of the whole translated term is $\forall a.a \to a$. The translation enjoys a type preservation property.

E Translation from FreezeML to Poly-ML

Types Let ϵ be a fixed label. Then $[\![]\!]_\tau$ is defined as follows:

$$\begin{aligned} [\![a]\!]_\tau &= a \\ [\![A_1 \to A_2]\!]_\tau &= [\![A_1]\!]_\tau \to [\![A_2]\!]_\tau \\ [\![\forall\Delta.H]\!]_\tau &= [\forall\Delta.[\![H]\!]_\tau]^\epsilon \qquad \text{if } \Delta \neq \cdot \end{aligned}$$

Further, $[\![]\!]_\sigma$ is defined as follows, meaning that $[\![]\!]_\sigma$ behaves like $[\![]\!]_\tau$ but leaves quantifiers at the toplevel unboxed.

$$\begin{aligned} [\![a]\!]_\sigma &= [\![a]\!]_\tau \\ [\![A_1 \to A_2]\!]_\sigma &= [\![A_1 \to A_2]\!]_\tau \\ [\![\forall\Delta.H]\!]_\sigma &= \forall\Delta.[\![H]\!]_\tau \qquad \text{if } \Delta \neq \cdot \end{aligned}$$

Finally, $[\![A]\!]_\varsigma$ is defined as $\forall\epsilon.[\![A]\!]_\tau$ and is applied to typing environments by applying $[\![]\!]_\varsigma$ to the types therein.

Terms (Core)

$$\left[\!\!\left[\frac{x : A \in \Gamma}{\Delta;\Gamma \vdash \lceil x\rceil : A} \right]\!\!\right] = x$$

$$\left[\!\!\left[\frac{\begin{array}{c} x : \forall\Delta'.H \in \Gamma \\ \Delta \vdash \delta : \Delta' \Rightarrow_\star \cdot \end{array}}{\Delta;\Gamma \vdash x : \delta(H)} \right]\!\!\right] = \begin{cases} x & \text{if } \Delta' = \cdot \\ \langle x\rangle & \text{otherwise} \end{cases}$$

$$\left[\!\!\left[\frac{\begin{array}{c} \Delta;\Gamma \vdash M : A \to B \\ \Delta;\Gamma \vdash N : A \end{array}}{\Delta;\Gamma \vdash M\,N : B} \right]\!\!\right] = [\![M]\!]\ [\![N]\!]$$

$$\left[\!\!\left[\frac{\Delta;\Gamma, x : S \vdash M : B}{\Delta;\Gamma \vdash \lambda x.M : S \to B} \right]\!\!\right] = \lambda x.\,[\![M]\!]$$

$$\begin{aligned} \left[\!\!\left[\frac{\Delta;\Gamma, x : A \vdash M : B}{\Delta;\Gamma \vdash \lambda(x : A).M : A \to B} \right]\!\!\right] &= \lambda(x : [\![A]\!]_\tau).\,[\![M]\!] \\ &= \lambda x.\mathbf{let}\ x = (x : [\![A]\!]_\tau)\ \mathbf{in}\ [\![M]\!] \end{aligned}$$

Terms (Let, value-restricted)

$$\left[\!\!\left[\frac{\begin{array}{c} (\Delta',\Delta'') = \text{gen}(\Delta, A', M) \\ \Delta,\Delta'';\Gamma \vdash M : A' \\ (\Delta,\Delta'',M,A') \Updownarrow A \\ \Delta;\Gamma, x : A \vdash N : B \\ \text{principal}(\Delta,\Gamma,M,\Delta'',A') \end{array}}{\Delta;\Gamma \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : B} \right]\!\!\right] = \begin{cases} \mathbf{let}\ x = [[\![M]\!] : [\![A]\!]_\sigma]\ \mathbf{in}\ [\![N]\!] & \text{if } \Delta' \neq \cdot \\ \mathbf{let}\ x = [\![M]\!]\ \ \mathbf{in}\ \ [\![N]\!] & \text{otherwise} \end{cases}$$

Note that in the first case above, the type annotation $[\![A]\!]_\sigma$ would not be needed if Poly-ML was extended with a boxing operator that does not require a type annotation but uses the principal type instead.

$$\left[\!\!\left[\begin{array}{c}(\Delta', A') = \mathsf{split}(A, M) \\ \Delta, \Delta'; \Gamma \vdash M : A' \\ A = \forall \Delta'.A' \\ \Delta; \Gamma, x : A \vdash N : B \\ \hline \Delta; \Gamma \vdash \mathbf{let}\ (x : A) = M\ \mathbf{in}\ N : B\end{array}\right]\!\!\right] = \begin{cases}\mathbf{let}\ x = [[\![M]\!] : [\![A]\!]_\sigma]\ \mathbf{in}\ [\![N]\!] & \text{if } \Delta' \neq \cdot \\ \mathbf{let}\ x = [\![M]\!]\ \mathbf{in}\ [\![N]\!] & \text{otherwise}\end{cases}$$

Lemma E.1. *If $\Delta; \Gamma \vdash M : A$ in FreezeML, then $[\![\Gamma]\!]_\varsigma \vdash [\![M]\!] : [\![A]\!]_\tau$ in Poly-ML.*

F Proofs from Section 4

For convenience, we use the following (derivable) System F typing rules, allowing n-ary type applications and abstractions:

$$\frac{\Delta; \Gamma \vdash M : \forall \Delta'.B \qquad \Delta' = a_1, \ldots, a_n \qquad \overline{A} = A_1, \ldots, A_n}{\Delta; \Gamma \vdash M\,\overline{A} : B[A_1/a_1] \cdots [A_n/a_n]}\ \textsc{F-PolyApp}^*$$

where $\overline{A}$ may be empty

$$\frac{\Delta, \Delta'; \Gamma \vdash V : A}{\Delta; \Gamma \vdash \Lambda\,\Delta'.V : \forall \Delta'.A}\ \textsc{F-PolyLam}^*$$

Recall that we have defined $\mathbf{let}\ x^A = M\ \mathbf{in}\ N$ as syntactic sugar for $(\lambda x^A.N)\ M$ in System F .

For readability, we preserve the syntactic sugar in the proofs and use the following typing rule:

$$\frac{\Delta; \Gamma \vdash M : A \qquad \Delta; \Gamma, x : A \vdash N : B}{\Delta; \Gamma \vdash \mathbf{let}\ x^A = M\ \mathbf{in}\ N : B}\ \textsc{F-Let}$$

Lemma F.1. *For each System F value V, $\mathcal{E}[\![V]\!]$ is a FreezeML value.*

Proof. By induction on structure of V. □

Theorem 2 (Type preservation). *If $\Delta; \Gamma \vdash M : A$ in System F then $\Delta; \Gamma \vdash \mathcal{E}[\![M]\!] : A$ in FreezeML.*

Proof. The proof is by well-founded induction on derivations of $\Delta; \Gamma \vdash M : A$. This means that we may apply the induction hypothesis to any judgement appearing in a subderivation, not just to those appearing in the immediate ancestors of the conclusion. We slightly strengthen the induction hypothesis so that the A is the *unique* type of $\mathcal{E}[\![M]\!]$. Formally, we show that if $\Delta; \Gamma \vdash M : A$ holds in System F, then $\Delta; \Gamma \vdash \mathcal{E}[\![M]\!] : A$ holds in FreezeML and for all B with $\Delta; \Gamma \vdash \mathcal{E}[\![M]\!] : B$ we have $A = B$. We show how to extend $\mathcal{E}[\![-]\!]$ to a function that translates System F type derivations to FreezeML type derivations.

- Case F-Var, $\mathcal{J} = \Delta; \Gamma \vdash x : A$:

$$\mathcal{E}\left[\!\!\left[\frac{x : A \in \Gamma}{\Delta; \Gamma \vdash x : A}\right]\!\!\right] \Longrightarrow \frac{x : A \in \Gamma}{\Delta; \Gamma \vdash \lceil x \rceil : A}$$

- Case F-Lam, $\mathcal{J} = \Delta; \Gamma \vdash \lambda x^A.M : A \to B$:

$$\mathcal{E}\left[\!\!\left[\frac{\Delta; \Gamma, x : A \vdash M : B}{\Delta; \Gamma \vdash \lambda x^A.M : A \to B}\right]\!\!\right] \Longrightarrow \frac{\Delta; \Gamma, x : A \vdash \mathcal{E}[\![M]\!] : B}{\Delta; \Gamma \vdash \lambda(x : A).\mathcal{E}[\![M]\!] : A \to B}$$

- Case F-App, $\mathcal{J} = \Delta;\Gamma \vdash M\ N : B$:

$$\mathcal{E}\left[\!\!\left[\dfrac{\begin{array}{c}\Delta;\Gamma \vdash M : A \to B \\ \Delta;\Gamma \vdash N : A\end{array}}{\Delta;\Gamma \vdash M\ N : B} \right]\!\!\right] \Longrightarrow \dfrac{\begin{array}{c}\Delta;\Gamma \vdash \mathcal{E}[\![M]\!] : A \to B \\ \Delta;\Gamma \vdash \mathcal{E}[\![N]\!] : A\end{array}}{\Delta;\Gamma \vdash \mathcal{E}[\![M]\!]\ \mathcal{E}[\![N]\!] : B}$$

- Case F-TAbs, , $\mathcal{J} = \Delta;\Gamma \vdash \Delta a.V : \forall a.B$
Let $B = \forall \Delta_B.H_B$. By Lemma F.1, $\mathcal{E}[\![V]\!]$ is a value, and **let** $y = \mathcal{E}[\![V]\!]$ **in** y is a guarded value which we refer to as $U_@$.

$$\mathcal{E}\left[\!\!\left[\dfrac{\Delta, a;\Gamma \vdash V : B}{\Delta;\Gamma \vdash \Lambda a.V : \forall a.B} \right]\!\!\right] \quad \Longrightarrow$$

$$\dfrac{\mathcal{D} \qquad \dfrac{x : \forall a.B \in \Gamma, (x : \forall a.B)}{\Delta;\Gamma,(x : \forall a.B) \vdash \lceil x \rceil : \forall a.B} \qquad ((a,\Delta_B), H_B) = \mathsf{split}(\forall a.B, U_@)}{\Delta;\Gamma \vdash \mathbf{let}\ (x : \forall a.B) = U_@\ \mathbf{in}\ \lceil x \rceil : \forall a.B}$$

The sub-derivation $\mathcal{D}$ for $\Delta, a, \Delta_B;\Gamma \vdash U_@ : H_B$ differs based on whether $\mathcal{E}[\![V]\!]$ is a guarded value or not:

If $\mathcal{E}[\![V]\!] \in \mathsf{GVal}$: $\mathcal{E}[\![V]\!]$ must have a guarded type and hence we have $B = H_B$ and $\Delta_B = \cdot$. By induction we have $\Delta, a;\Gamma \vdash \mathcal{E}[\![V]\!] : B$ and hence $\mathsf{ftv}(B) \subseteq \Delta, a$. This further implies $\mathsf{gen}((\Delta, a, \Delta_B), H_B, \mathcal{E}[\![V]\!]) = (\cdot,\cdot)$. Let δ be the empty substitution.

$$\dfrac{\begin{array}{c} \Delta, a, \Delta_B;\Gamma \vdash \mathcal{E}[\![V]\!] : H_B \qquad \dfrac{\begin{array}{c} y : H_B \in \Gamma \\ \Delta, a, \Delta_B \vdash \delta : \cdot \Rightarrow_\star \cdot \end{array}}{\Delta;\Gamma, y : H_B \vdash y : \delta(H_B)} \\ ((\Delta, a, \Delta_B), \cdot, \mathcal{E}[\![V]\!], H_B) \Updownarrow H_B \qquad (\cdot,\cdot) = \mathsf{gen}((\Delta, a, \Delta_B), H_B, \mathcal{E}[\![V]\!]) \qquad \mathsf{principal}((\Delta, a, \Delta_B), \Gamma, \mathcal{E}[\![V]\!], \cdot, B') \end{array}}{\Delta, a, \Delta_B;\Gamma \vdash \mathbf{let}\ y = \mathcal{E}[\![V]\!]\ \mathbf{in}\ y : H_B}$$

If $\mathcal{E}[\![V]\!] \notin \mathsf{GVal}$: Let $B' = \forall \Delta'.H'$ be alpha-equivalent to B such that all Δ' are fresh. We then have $\Delta, a;\Gamma \vdash \mathcal{E}[\![V]\!] : B'$ by induction. This implies $\mathsf{gen}((\Delta, a, \Delta_B), B', \mathcal{E}[\![V]\!]) = (\cdot,\cdot)$. Let δ be defined such that $\delta(\Delta') = \Delta_B$, which implies $\delta(H') = H_B$.

$$\dfrac{\begin{array}{c} \Delta, a, \Delta_B;\Gamma \vdash \mathcal{E}[\![V]\!] : B' \qquad \dfrac{\begin{array}{c} y : \Delta'.H' \in \Gamma \\ \Delta, a, \Delta_B \vdash \delta : \Delta' \Rightarrow_\star \cdot \end{array}}{\Delta, a, \Delta_B;\Gamma, y : B' \vdash y : \delta(H')} \\ ((\Delta, a, \Delta_B), \cdot, \mathcal{E}[\![V]\!], B') \Updownarrow B' \qquad (\cdot,\cdot) = \mathsf{gen}((\Delta, a, \Delta_B), B', \mathcal{E}[\![V]\!]) \qquad \mathsf{principal}((\Delta, a, \Delta_B), \Gamma, \mathcal{E}[\![V]\!], \cdot, B') \end{array}}{\Delta, a, \Delta_B;\Gamma \vdash \mathbf{let}\ y = \mathcal{E}[\![V]\!]\ \mathbf{in}\ y : H_B}$$

In both cases, satisfaction of $\mathsf{principal}((\Delta, a, \Delta_B), \Gamma, \mathcal{E}[\![V]\!], \cdot, B')$ follows from the fact that by induction, B' is the unique type of $\mathcal{E}[\![V]\!]$.

- Case F-TApp, , $\mathcal{J} = \Delta;\Gamma \vdash M\ A : B[A/a]$
Let $B = \forall \Delta_B.H_B$ and w.l.o.g. $a \mathrel{\#} \Delta_B$ and $\mathsf{ftv}(A) \mathrel{\#} a, \Delta_B$. Let $U_@$ be defined as in the previous case. We then have

$$\mathcal{E}\left[\!\!\left[\dfrac{\Delta;\Gamma \vdash M : \forall a.B}{\Delta;\Gamma \vdash M\,A : B[A/a]} \right]\!\!\right] \Longrightarrow$$

$$\dfrac{\mathcal{D} \qquad \dfrac{x : B[A/a] \in \Gamma, (x : B[A/a])}{\Delta;\Gamma,(x : B[A/a]) \vdash \lceil x \rceil : B[A/a]} \qquad (\Delta_B, H_B[A/a]) = \mathsf{split}(B[A/a], U_@)}{\Delta;\Gamma \vdash \mathbf{let}\,(x : B[A/a]) = U_@\ \mathbf{in}\ \lceil x \rceil : B[A/a]}$$

We consider the sub-derivation $\mathcal{D}$ for $\Delta, \Delta_B;\Gamma \vdash U_@ : H_B[A/a]$

By induction, we have $\Delta;\Gamma \vdash \mathcal{E}[\![V]\!] : \forall a.B$, which implies that $\mathcal{E}[\![V]\!]$ is not a guarded value.

Let $B' = \forall \Delta'.H'$ be alpha-equivalent to B such that all Δ' are fresh. We then have $\Delta; \Gamma \vdash \mathcal{E}[\![V]\!] : \forall a.B'$ by induction. This implies $(\cdot, \cdot) = \mathrm{gen}((\Delta, \Delta_B), \forall a.B', \mathcal{E}[\![V]\!])$. Let δ be defined such that $\delta(\Delta') = \Delta_B$ and $\delta(a) = A$, which implies $\delta(H') = H_B[A/a]$.

$$\frac{\Delta; \Gamma \vdash \mathcal{E}[\![V]\!] : \forall a.B' \qquad \dfrac{\begin{array}{c} y : a, \Delta'.H' \in \Gamma \\ \Delta, \Delta_B \vdash \delta : a, \Delta' \Rightarrow_\star \cdot \end{array}}{\Delta, \Delta_B; \Gamma, y : \forall a.B' \vdash y : \delta(H')} \\ ((\Delta, \Delta_B), \cdot, \mathcal{E}[\![V]\!], \forall a.B') \Updownarrow \forall a.B' \qquad (\cdot, \cdot) = \mathrm{gen}((\Delta, \Delta_B), \forall a.B', \mathcal{E}[\![V]\!]) \qquad \mathrm{principal}((\Delta, \Delta_B), \Gamma, \mathcal{E}[\![V]\!], \cdot, \forall a.B')}{\Delta, \Delta_B; \Gamma \vdash \mathbf{let}\ y = \mathcal{E}[\![V]\!]\ \mathbf{in}\ y : H_B[A/a]}$$

As in the previous case, satisfaction of $\mathrm{principal}((\Delta, a, \Delta_B), \Gamma, \mathcal{E}[\![V]\!], \cdot, \forall a.B')$ follows from the fact that by induction, $\forall a.B'$ is the unique type of $\mathcal{E}[\![V]\!]$.

Finally, we observe that the translated terms indeed have unique types: For variables, the type is uniquely determined from the context. Functions are translated to annotated lambdas, without any choice for the parameter type. For term applications, uniqueness follows by induction. For term applications an abstractions, the result type of the expression is the type of freezing x. In both cases, this variable is annotated with a type.

This completes the proof, since any derivation is in one of the forms used in the above cases. □

Theorem 3 (Type preservation). *If $\Delta; \Gamma \vdash M : A$ holds in FreezeML then $\Delta; \Gamma \vdash \mathcal{C}[\![M]\!] : A$ holds in System F.*

Proof. We perform induction on the derivation of $\Delta, \Gamma \vdash M : A$. In each case we show how the definition of $\mathcal{C}[\![-]\!]$ can be extended to a function returning the desired derivation.

- Case Freeze:

$$\mathcal{C}\left[\!\!\left[\frac{x : A \in \Gamma}{\Delta; \Gamma \vdash \lceil x \rceil : A}\right]\!\!\right] \Longrightarrow \frac{x : A \in \Gamma}{\Delta; \Gamma \vdash x : A}\ \text{F-Var}$$

- Case Var: Let $\Delta' = (a_1, \ldots, a_n)$.

$$\mathcal{C}\left[\!\!\left[\frac{x : \forall \Delta'.H \in \Gamma \qquad \Delta \vdash \delta : \Delta' \Rightarrow \cdot}{\Delta; \Gamma \vdash x : \delta(H)}\right]\!\!\right] \Longrightarrow \frac{\dfrac{x : \forall \Delta'.H \in \Gamma}{\Delta; \Gamma \vdash x : \forall a_1, \ldots, a_n.H}\ \text{F-Var}}{\Delta; \Gamma \vdash x\, \delta a_1\ \cdots\ \delta a_n : H[\delta a_1/a_1] \cdots [\delta a_n/a_n]}\ \text{F-PolyApp}^*$$

- Case Lam:

$$\mathcal{C}\left[\!\!\left[\frac{\Delta; \Gamma, x : S \vdash M : B}{\Delta; \Gamma \vdash \lambda x.M : S \to B}\right]\!\!\right] \Longrightarrow \frac{\Delta; \Gamma, x : S \vdash \mathcal{C}[\![M]\!] : B}{\Delta; \Gamma \vdash \lambda x^S.\mathcal{C}[\![M]\!] : S \to B}\ \text{F-Lam}$$

- Case Lam-Ascribe

$$\mathcal{C}\left[\!\!\left[\frac{\Delta; \Gamma, x : A \vdash M : B}{\Delta; \Gamma \vdash \lambda(x : A).M : A \to B}\right]\!\!\right] \Longrightarrow \frac{\Delta; \Gamma, x : A \vdash \mathcal{C}[\![M]\!] : B}{\Delta; \Gamma \vdash \lambda x^A.\mathcal{C}[\![M]\!] : A \to B}\ \text{F-Lam}$$

- Case App:

$$\mathcal{C}\left[\!\!\left[\frac{\Delta; \Gamma \vdash M : A \to B \qquad \Delta; \Gamma \vdash N : A}{\Delta; \Gamma \vdash M\,N : B}\right]\!\!\right] \Longrightarrow \frac{\Delta; \Gamma \vdash \mathcal{C}[\![M]\!] : A \to B \qquad \Delta; \Gamma \vdash \mathcal{C}[\![N]\!] : A}{\Delta; \Gamma \vdash \mathcal{C}[\![M]\!]\,\mathcal{C}[\![N]\!] : B}\ \text{F-App}$$

- Case Let: In this case there are two subcases, depending on whether M is a guarded value or not.

– $M = V \in$ GVal: In this case, we have $\mathrm{gen}(\Delta, A', M) = (\Delta', \Delta')$ for some possibly nonempty Δ', and $(\Delta, \Delta', M, A') \Updownarrow \forall \Delta'.A'$. We proceed as follows:

$$C\left[\!\!\left[\begin{array}{c}\Delta, \Delta'; \Gamma \vdash V : A' \\ \Delta; \Gamma, x : \forall \Delta'.A' \vdash N : B \\ \cdots \\ \hline \Delta; \Gamma \vdash \mathbf{let}\ x = V\ \mathbf{in}\ N : B\end{array}\right]\!\!\right] \Longrightarrow$$

$$\dfrac{\dfrac{\Delta, \Delta'; \Gamma \vdash C[\![V]\!] : A'}{\Delta; \Gamma \vdash \Lambda\, \Delta'.C[\![V]\!] : \forall \Delta'.A'}\ \textsc{F-PolyLam}^* \qquad \Delta; \Gamma, x : \forall \Delta'.A' \vdash C[\![N]\!] : B}{\Delta; \Gamma \vdash \mathbf{let}\, x^A = \Lambda\, \Delta'.C[\![V]\!]\ \mathbf{in}\ C[\![N]\!] : B}\ \textsc{F-Let}$$

where we rely on the fact that $C[\![V]\!]$ is a value in System F as well, and appeal to the derivable rule F-PolyLam*.

– $M \notin$ GVal. In this case, we know that $\mathrm{gen}(\Delta, A, M) = (\cdot, \Delta')$ and $(\Delta, \Delta', M, A') = \delta(A') = A$ for some δ satisfying $\Delta \vdash \delta : \Delta' \Rightarrow_{\bullet} \cdot$. We proceed as follows:

$$C\left[\!\!\left[\begin{array}{c}\Delta, \Delta'; \Gamma \vdash M : A' \\ \Delta; \Gamma, x : A \vdash N : B \\ \cdots \\ \hline \Delta; \Gamma \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : B\end{array}\right]\!\!\right] \Longrightarrow$$

$$\dfrac{\Delta; \Gamma \vdash \delta(C[\![M]\!]) : \delta(A') \qquad \Delta; \Gamma, x : A \vdash C[\![N]\!] : B}{\Delta; \Gamma \vdash \mathbf{let}\, x^A = C[\![M]\!]\ \mathbf{in}\ C[\![N]\!] : B}$$

where we make use of a standard substitution lemma for System F to instantiate type variables from Δ' in $C[\![M]\!]$ and A to obtain a derivation of $\Delta; \Gamma \vdash \delta(C[\![M]\!]) : \delta(A')$, which suffices since $A = \delta(A')$. Note that $C[\![M]\!]$ could contain free type variables from Δ' since all inferred types are translated to explicit annotations.

- Case Let-Ascribe: This case is analogous to the case for Let.

□

G Type Substitutions, Environments and Well-Scoped Terms

This section collects, and sketches (mostly straightforward) proofs of properties about type substitutions, kind and type environments, and the well-scoped term judgement. We may then use the properties from this section without explicitly referencing them in subsequent sections.

Note that when types appear on their own or in contexts Γ, we identify α-equivalent types.

We use the following notations in this and subsequent sections, where $\Theta = (a_1 : K_1, \ldots, a_n : K_n)$. Recall that this implies all a_i being pairwise different.

- Let $(b : K) \in \Theta$ hold iff $b = a_i$ and $K = K_i$ for some $1 \le i \le n$ and let $b \in \Theta$ hold iff $(b : K) \in \Theta$ holds for some K.
- For all $1 \le i \le n$, we define $\Theta(a_i) = K_i$.
- We define $\mathrm{ftv}(\Theta)$ as $(a_1, \ldots, a_n)$.
- Given θ such that $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$, then $\mathrm{ftv}(\theta)$ is defined as $\mathrm{ftv}(\theta(a_1) \to \ldots \to \theta(a_n))$.
- Given $\Theta' = (b_1 : K'_1, \ldots, b_m : K'_m)$, $\Theta' \subseteq \Theta$ holds iff there exists a function f from $\{1, \ldots, m\}$ to $\{1, \ldots, n\}$ such that for all $1 \le i \le m$, we have $b_i = a_{f(i)}$ and $K'_i = K_{f(i)}$.
- We have $\Theta \approx \Theta'$ iff $\Theta \subseteq \Theta$ and $\Theta' \subseteq \Theta$.
- Given $\Delta = (a_1, \ldots, a_n)$, all of the above notations are defined on Δ by applying them to $\Theta = (a_1 : \bullet, \ldots, a_n : \bullet)$.
- Given kinds K, K', we write $K \le K'$ iff $K \sqcup K' = K'$.

Lemma G.1. *If $A = B$ then $\theta(A) = \theta(B)$ for any θ.*

Proof. The point of this property is that alpha-equivalence is preserved by substitution application, because substitution application is capture-avoiding. Concretely, the proof is by induction on the (equal) structure of A and B. In the case of a binder $A = \forall a.A' = \forall b.B' = B$, where one or both of a, b are affected by θ, alpha-equivalence implies that we may rename a and b respectively to a sufficiently fresh c, such that $A'[c/a] = B'[c/a]$ and $\theta(c) = c$. Therefore, by induction $\theta(A) = \theta(\forall a.A') = \theta(\forall c.A'[c/a]) = \forall c.\theta(A'[c/a]) = \forall c.\theta(B'[c/b]) = \theta(\forall c.B'[c/b]) = \theta(\forall b.B') = \theta(B)$. □

Lemma G.2. *$\theta(\forall a.A) = \theta(\forall c.A[c/a])$, where $c \notin \mathrm{ftv}(\theta) \cup \mathrm{ftv}(A)$ is fresh.*

Proof. This is a special case of the previous property, observing that $\forall a.A = \forall c.A[c/a]$ if c is sufficiently fresh. □

Lemma G.3. *If $\Delta \vdash \theta[a \to A] : \Theta, (a : K) \Rightarrow \Theta'$, then $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$ and $\Delta, \Theta' \vdash A : K$.*

Proof. This follows by inversion on the substitution well-formedness judgement. □

Lemma G.4. *If $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$ and $\Delta, \Theta \vdash a : K$ then $\Delta, \Theta' \vdash \theta(a) : K$.*

Proof. By induction on the structure of the derivation of $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$. The base case is straightforward: if θ is empty then Θ is also empty so $a \in \Delta$. Moreover, $\theta(a) = a$ so we can conclude $\Delta, \Theta' \vdash \theta(a) : K$. For the inductive case, we have a derivation of the form:

$$\frac{\Delta \vdash \theta : \Theta \Rightarrow \Theta' \qquad \Delta, \Theta \vdash A' : K'}{\Delta \vdash \theta[a' \mapsto A'] : (\Theta, a' : K') \Rightarrow \Theta'}$$

There are two cases. If $a = a'$ then the subderivation of $\Delta, \Theta \vdash A' : K'$ proves the desired conclusion since $\theta[a' \mapsto A'](a) = A'$ and $K = K'$. Otherwise, $a \neq a'$ so from $\Delta, \Theta, a' : K' \vdash a : K$ we can infer that $\Delta, \Theta \vdash a : K$ as well. So, by induction we have that $\Delta, \Theta' \vdash \theta(a) : K$. Since $a \neq a'$ we can also conclude that $\Delta, \Theta' \vdash \theta[a' \mapsto A'](a) : K$, as desired. □

Lemma G.5. *If $\Delta, \Theta \vdash A : K$ and $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$, then $\Delta, \Theta' \vdash \theta A : K$.*

Proof. By induction on the structure of the derivation of $\Delta, \Theta \vdash A : K$. The case for TyVar is G.4. The cases for Cons and Upcast are immediate by induction. For the ForAll case, assume the derivation is of the form:

$$\frac{\Delta, \Theta, a : \star \vdash A : \star}{\Delta, \Theta \vdash \forall a.A : \star}$$

Without loss of generality, assume a is fresh and in particular not mentioned in Θ, Θ', Δ. Then we can derive $\Delta \vdash \theta[a \mapsto a] : \Theta, a : \star \Rightarrow \Theta', a : \star$, and we may apply the induction hypothesis to conclude that $\Delta, \Theta', a : \star \vdash \theta[a \mapsto a](A) : \star$. Moreover, since a was sufficiently fresh, and is unchanged by $\theta[a \mapsto a]$, we can conclude $\Delta, \Theta' \vdash \forall a.A : \star$. □

Lemma G.6. *If $\Delta, \Theta \vdash \Gamma$ and $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$. then $\Delta, \Theta' \vdash \theta\Gamma$.*

Proof. By induction on the derivation of $\Delta, \Theta \vdash \Gamma$. The base case is:

$$\frac{}{\Delta, \Theta \vdash \cdot}$$

Moreover, it follows from $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$ that $\Delta \# \Theta'$, so the conclusion is immediate, since $\theta(\cdot) = \cdot$. In the inductive case, the derivation of $\Delta, \Theta \vdash \Gamma, x : A$ is of the form:

$$\frac{\Delta, \Theta \vdash \Gamma \qquad \Delta, \Theta \vdash A : \star \qquad \forall a \in \mathrm{ftv}(A).(\Delta, \Theta)(a) = \bullet}{\Delta, \Theta \vdash \Gamma, x : A}$$

In this case, by induction we have $\Delta, \Theta' \vdash \theta\Gamma$ and using Lemma G.5 we have $\Delta, \Theta' \vdash \theta A : K$. We also need to show that $\forall a \in \mathrm{ftv}(\theta(A))$, we have $(\Delta, \Theta')(a) = \bullet$. There are two cases: if $a \in \Delta$ this is immediate. If $a \in \Theta'$, then since $a \in \mathrm{ftv}(\theta(A))$ we know that there must exist $b \in \Theta$ such that $a \in \mathrm{ftv}(\theta(b))$ and $b \in \mathrm{ftv}(A)$. By virtue of the assumption $\forall a \in \mathrm{ftv}(A).(\Delta, \Theta)(a) = \bullet$, we know that $(\Delta, \Theta)(b) = \bullet$, hence $\Theta(b) = \bullet$. This implies that $\Delta, \Theta' \vdash \theta(b) : \bullet$, which further implies that all the free type variables of $\theta(b)$, including a, must also have kind $\bullet$. Now the desired conclusion $\Delta, \Theta' \vdash \theta(\Gamma, x : A)$ follows. □

Lemma G.7. *1. If $\Delta \vdash \delta_1 : \Delta_1 \Rightarrow_K \Delta_2$ and $\Delta \vdash \delta_2 : \Delta_2 \Rightarrow_K \Delta_3$ then $\Delta \vdash \delta_2 \circ \delta_1 : \Delta_1 \Rightarrow_K \Delta_3$.*
2. If $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$ and $\Delta' \# \Theta' \# \Delta''$ and $\Delta, \Theta \vdash \delta_1 : \Delta' \Rightarrow_K \Delta''$ then $\Delta, \Theta' \vdash \theta \circ \delta_1 : \Delta' \Rightarrow_K \Delta''$.

Proof. In both cases, by straightforward induction on structure of δ_1. □

Lemma G.8. *If $\Theta \vdash A : K$ and $\Theta' \# \Theta$ then $\Theta, \Theta' \vdash A : K$.*

Proof. Straightforward by induction on the structure of derivations of $\Theta \vdash A : K$. The only subtlety is in the case for $\forall$-types, where we assume without loss of generality that the bound type variable a is renamed away from Θ and Θ', so that the induction hypothesis applies. □

Lemma G.9. *If $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$ and $\Delta' \# \Delta, \Theta'$ as well as $\Delta' \# \Theta$ then $\Delta, \Delta' \vdash \theta : \Theta \Rightarrow \Theta'$.*

Proof. By induction on the derivation of $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$. The base case is immediate given that Δ' is fresh for Δ and Θ'. For the inductive case, we have a derivation of the form:

$$\frac{\Delta \vdash \theta : \Theta \Rightarrow \Theta' \qquad \Delta, \Theta' \vdash A : K}{\Delta \vdash \theta[a \mapsto A] : (\Theta, a : K) \Rightarrow \Theta'}$$

By induction (since Δ' is clearly fresh for Δ, Θ, and Θ') we have $\Delta, \Delta' \vdash \theta : \Theta \Rightarrow \Theta'$. Moreover, by weakening (Lemma G.8) we also have $\Delta, \Delta', \Theta' \vdash A : K$. We can conclude, as required, that $\Delta, \Delta' \vdash \theta[a \mapsto A] : (\Theta, a : K) \Rightarrow \Theta'$. □

Lemma G.10. *If* $\Theta_D = \mathrm{demote}(K, \Theta, \Delta')$ *and* $\Delta \vdash \theta : \Theta_D \Rightarrow \Theta'$ *then* $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$.

Proof. If $K = \star$, demote yields $\Theta = \Theta_D$ and the statement holds immediately.

Otherwise, if $K = \bullet$, we perform induction on Θ_D. By definition of demote, we have $\mathrm{ftv}(\Theta) = \mathrm{ftv}(\Theta_D)$.

If $\Theta_D = \cdot$ we have $\Theta = \cdot$ and can derive the following:

$$\frac{}{\Delta \vdash \emptyset : \cdot \Rightarrow \Theta'}$$

Let $\Theta_D = (\Theta''_D, a : K')$. By inversion we then have

$$\frac{\Delta \vdash \theta : \Theta''_D \Rightarrow \Theta' \qquad \Delta, \Theta' \vdash A : K'}{\Delta \vdash \theta[a \mapsto A] : (\Theta''_D, a : K') \Rightarrow \Theta'}$$

By $\mathrm{ftv}(\Theta) = \mathrm{ftv}(\Theta_D)$ we have $\Theta = (\Theta'', a : K'')$. By induction this implies $\Delta \vdash \theta : \Theta'' \Rightarrow \Theta'$.

If $K' = \star$, then by definition of demote we have $a \notin \Delta'$ and $K'' = \star$. We can then derive the following:

$$\frac{\Delta \vdash \theta : \Theta'' \Rightarrow \Theta' \qquad \Delta, \Theta' \vdash A : \star}{\Delta \vdash \theta[a \mapsto A] : (\Theta'', a : \star) \Rightarrow \Theta'}$$

Otherwise, we have $K' = \bullet$ and show that $\Delta, \Theta' \vdash A : K''$ holds. If $K'' = \bullet$, this follows immediately from $\Delta, \Theta' \vdash A : K'$. If $K'' = \star$, we upcast $\Delta, \Theta' \vdash A : \bullet$ to $\Delta, \Theta' \vdash A : \star$.

In both cases for K'', we can then derive the following:

$$\frac{\Delta \vdash \theta : \Theta \Rightarrow \Theta' \qquad \Delta, \Theta' \vdash A : K''}{\Delta \vdash \theta[a \mapsto A] : (\Theta'', a : \bullet) \Rightarrow \Theta'}$$

□

Lemma G.11. *If* $\Theta' = \mathrm{demote}(K, \Theta, \Delta)$ *then* $\mathrm{ftv}(\Theta) = \mathrm{ftv}(\Theta')$ *and* $\Delta \vdash \iota : \Theta \Rightarrow \Theta'$.

Proof. Proof by case analysis on K and induction on Θ. There are three cases. If $K = \star$ then the result is immediate since $\Theta = \Theta'$. If $K = \bullet$ and $\Theta = \cdot$ then the result is also immediate. Otherwise, if $K = \bullet$ and $\Theta = \Theta_1, a : K$ then $\mathrm{demote}(K, \Theta, \Delta) = \mathrm{demote}(K, \Theta_1, \Delta), a : K'$, where $\Theta'_1 = \mathrm{demote}(K, \Theta_1, \Delta)$ and K' is $\bullet$ if $a \in \Delta$, otherwise $K = K'$. Then by induction we have $\mathrm{ftv}(\Theta_1) = \mathrm{ftv}(\Theta'_1)$ and $\Delta \vdash \iota : \Theta_1 \Rightarrow \Theta'_1$. Clearly, $\mathrm{ftv}(\Theta_1, a : K) = \mathrm{ftv}(\Theta'_1, a : K')$. To see that $\Delta \vdash \iota : \Theta \Rightarrow \Theta'$, consider two cases: if $a \in \Delta$ then $K' = \bullet$ and we can conclude $\Delta \vdash \iota : \Theta, a : K \Rightarrow \Theta'_1, a : \bullet$ since if $K = \star$ then we can use Upcast. Otherwise, $K = K'$ so the result is immediate. □

Lemma G.12. *Let* $\Delta : \Theta \Rightarrow \Theta'$ *and* $\Delta, \Theta \vdash A : K$ *such that* $\Delta, \Theta' \vdash \theta(A) : K'$ *for some* K' *with* $K' \leq K$. *Furthermore, let* $\mathrm{demote}(K', \Theta, \mathrm{ftv}(A) - \Delta) = \Theta_D$. *Then* $\Delta, \Theta_D \vdash A : K'$.

Proof. For $K' = K$, the statement follows immediately. Therefore, we consider only the case $K = \star, K' = \bullet$.

We perform induction on the derivation of $\Delta, \Theta' \vdash \theta(A) : \bullet$.

Case $\theta(A) = a$**:**

$$\frac{a : K' \in \Theta'}{\Delta, \Theta' \vdash a : \bullet}$$

We have $A = b$ for some $b \in \Delta, \Theta$. If $b \in \Delta$, then $\Delta \vdash b : \bullet$ follows immediately. Otherwise, we have $(b : K'') \in \Theta$ for some K''. By $b \in \mathrm{ftv}(A) - \Delta$, we then have $(b : \bullet)$ in Θ_D.

Case $\theta(A) = D\ \theta(A_1) \dots \theta(A_n)$:

$$\frac{\mathsf{arity}(D) = n \qquad \Delta, \Theta' \vdash \theta(A_1) : \bullet \quad \cdots \quad \Delta, \Theta' \vdash A_n : \bullet}{\Delta, \Theta' \vdash D\,\overline{\theta(A)} : \bullet}$$

By induction we have $\Delta, \Theta_D \vdash \theta(A_i) : \bullet$ for all $1 \le i \le n$. We can therefore derive $\Delta, \Theta_D \vdash D\,\overline{A} : \bullet$.

Note that we can disregard upcasts and $\theta(A) = \forall b.B$ as they would both yield $K' = \star$:

$$\frac{\Delta, \Theta', b : \bullet \vdash B : \star}{\Delta, \Theta' \vdash \forall b.B : \star} \qquad\qquad \frac{\Delta, \Theta' \vdash A : \bullet}{\Delta, \Theta' \vdash A : \star}$$

□

The following property states the well-formedness conditions needed in order for composition of substitutions to imply composition of the functions induced by them.

Lemma G.13. *Let the following conditions hold:*

$$\Delta \vdash \theta' : \Theta \Rightarrow \Theta' \qquad \textbf{(1)}$$

$$\Delta \vdash \theta'' : \Theta' \Rightarrow \Theta'' \qquad \textbf{(2)}$$

$$\theta = \theta'' \circ \theta' \qquad \textbf{(3)}$$

$$\Delta, \Theta \vdash A \qquad \textbf{(4)}$$

Then $\theta(A) = \theta''\theta'(A)$ holds.

Lemma G.14. *If $\Delta \Vdash M$, and $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$, then:*

1. *If $\mathsf{ftv}(A) - (\Delta, \Theta) \mathbin{\#} \Theta'$ then $\mathsf{gen}((\Delta, \Theta), A, M) = \mathsf{gen}((\Delta, \Theta'), \theta(A), M)$;*
2. *if $\Delta'' \mathbin{\#} \Delta, \Theta$ and $\Delta'' \mathbin{\#} \Theta'$ and $((\Delta, \Theta), \Delta'', M, A') \Updownarrow A$ then $((\Delta, \Theta'), \Delta'', M, \theta(A')) \Updownarrow \theta(A)$;*

Proof. 1. For part 1: Observe that

$$\mathsf{gen}((\Delta, \Theta), A, M) = \begin{cases} (\Delta', \Delta') & M \in \mathsf{GVal} \\ (\cdot, \Delta') & M \notin \mathsf{GVal} \end{cases}$$

$$\mathsf{gen}((\Delta, \Theta'), \theta(A), M) = \begin{cases} (\Delta'', \Delta'') & M \in \mathsf{GVal} \\ (\cdot, \Delta'') & M \notin \mathsf{GVal} \end{cases}$$

where $\Delta' = \mathsf{ftv}(A) - (\Delta, \Theta)$ and $\Delta'' = \mathsf{ftv}(\theta(A)) - (\Delta, \Theta')$. So, the equation $\mathsf{gen}((\Delta, \Theta), A, M) = \mathsf{gen}((\Delta, \Theta'), \theta(A), M)$ holds if and only if $\Delta' = \Delta''$. Suppose $a \in \Delta'$, that is, it is a free type variable of A and not among Δ, Θ. Since θ only affects type variables in Θ, we have $\theta(a) = a$ and it follows that $a \in \mathsf{ftv}(\theta(A))$. Moreover, by assumption $\Delta' \mathbin{\#} \Theta'$ so $a \in \mathsf{ftv}(\theta(A)) - (\Delta, \Theta') = \Delta''$. Conversely, suppose $a \in \Delta''$, that is, a is a free type variable of $\theta(A)$ and not among Δ, Θ'. Since $a \notin \Delta, \Theta'$, we must have $\theta(a) = a$ since θ was a well-formed substitution mentioning only type variables in Δ, Θ'. This implies that $a \in \mathsf{ftv}(A)$ since a cannot have been introduced by θ.
We has thus shown $\Delta' \approx \Delta''$. To show $\Delta' = \Delta''$, assume $a, b \in \Delta'$ such that a occurs before b in Δ'. This means that the first occurrence of a in A is before the first occurrence of b in A. For all $c \in \Theta$ we have $c \neq b$ and $\mathsf{ftv}(\theta(c)) \mathbin{\#} b$. Thus, the first occurrence of a in $\theta(A)$ remains before the first occurrence of b in $\theta(A)$.

2. For part 2: We consider two cases.
   - If the derivation is of the form

     $$\frac{M \in \mathsf{GVal}}{((\Delta, \Theta), \Delta'', M, A') \Updownarrow \forall \Delta''.A'}$$

     then we may derive

     $$\frac{M \in \mathsf{GVal}}{((\Delta, \Theta'), \Delta'', M, \theta(A')) \Updownarrow \forall \Delta''.\theta(A')}$$

     by observing that since $\Delta'' \mathbin{\#} \Theta$ and $\Delta'' \mathbin{\#} \Theta$, we know that $\theta(\forall \Delta''.A') = \forall \Delta''.\theta(A')$.

- If the derivation is of the form

$$\frac{\Delta, \Theta \vdash \delta : \Delta'' \Rightarrow_{\bullet} \cdot \qquad M \notin \mathsf{GVal}}{((\Delta, \Theta), \Delta'', M, A') \Updownarrow \delta(A')}$$

Then first we observe (by property G.7) that $\Delta, \Theta' \vdash \theta \circ \delta : \Delta'' \Rightarrow_{\bullet} \cdot$, so we can derive

$$\frac{\Delta, \Theta' \vdash \theta \circ \delta : \Delta'' \Rightarrow_{\bullet} \cdot \qquad M \notin \mathsf{GVal}}{((\Delta, \Theta'), \Delta'', M, \theta(A')) \Updownarrow \theta \circ \delta(\theta(A'))}$$

observing that $\theta(\delta(A')) = \theta \circ \delta(\theta(A'))$ since $\mathsf{ftv}(\theta) \# \Delta''$.

□

Lemma G.15. *Let $\Delta''; \Gamma \vdash M : A$ and $\Delta \Vdash M$. Further, let Δ' such that $\Delta \subseteq \Delta' \subseteq \Delta''$ and $\Delta' \vdash \Gamma$ and $\Delta' \subseteq \mathsf{ftv}(A)$, then $\Delta'; \Gamma \vdash M : A$ holds.*

Proof. Follows directly from observing that the variables in $\Delta'' - \Delta'$ have no influence on the typing derivation. □

Lemma G.16. *Let $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$ be a bijection between the type variables in Θ and Θ'. Furthermore, let $\Theta \# \Delta' \# \Theta'$. and $\Delta \Vdash M$ hold.*

Then the following holds:

1. *If $\Delta, \Theta; \Gamma \vdash M : A$ then $\Delta, \Theta'; \theta(\Gamma) \vdash M : \theta(A)$.*
2. *If $\mathsf{principal}((\Delta, \Theta), \Gamma, M, \Delta', A)$ then $\mathsf{principal}((\Delta, \Theta'), \theta(\Gamma), M, \Delta', \theta(A))$.*

Proof. Follows from the fact that θ merely swaps variables independent from Δ and Δ'. □

Lemma G.17. *If $\Delta \vdash \theta : \Theta \Rightarrow \Theta''$ and $\Delta \vdash \theta' : \Theta \Rightarrow \Theta'$ and $\Delta \vdash \theta'' : \Theta' \Rightarrow \Theta'', \Theta_E$ as well as $\theta = \theta'' \circ \theta'$ then for all $a \in \mathsf{ftv}(\theta') - \Delta$ we have $\Delta, \Theta'' \vdash \theta''(a)$.*

Proof. Via induction on θ', observing that $\Delta \vdash \theta : \Theta \Rightarrow \Theta''$ dictates the behaviour of θ'' on all variables in the intersection of Θ' and the codomain of θ'. □

H Correctness of unification proofs

H.1 Soundness of unification

Theorem 4 (Unification is sound). *If $\Delta, \Theta \vdash A, B : K$ and $\mathsf{unify}(\Delta, \Theta, A, B) = (\Theta', \theta)$ then $\theta(A) = \theta(B)$ and $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$.*

Proof. Via induction on the maximum of the sizes of A and B. We only consider the cases where unification succeeds.

1. $\mathsf{unify}(\Delta, \Theta, a, a)$: we have $\theta = \iota_{\Delta,\Theta}$ (identity substitution) and the result is immediate.
2. $\mathsf{unify}(\Delta, (\Theta, a : K'), a, A)$ or $\mathsf{unify}(\Delta, (\Theta, a : K'), A, a)$: We consider the first case; the second is symmetric. We have

$$\begin{array}{rcl} \mathsf{unify}(\Delta, (\Theta, a : K'), a, A) & = & (\Theta_1, \iota[a \mapsto A]) \\ \mathsf{demote}(K', \Theta, \mathsf{ftv}(A) - \Delta) & = & \Theta_1 \\ \Delta, \Theta_1 & \vdash & A : K' \end{array}$$

First, observe that $a \notin \mathsf{ftv}(A)$ since $a \notin \Delta, \Theta$ and $\mathsf{ftv}(\Theta_1) = \mathsf{ftv}(\Theta)$. Therefore

$$\iota[a \mapsto A](a) = A = \iota[a \mapsto A](A)$$

Next, by Lemma G.11 we know that $\Delta \vdash \iota : \Theta \Rightarrow \Theta_1$. Moreover, by $\Delta, \Theta_1 \vdash A : K'$ we can derive $\Delta \vdash \iota[a \mapsto A] : \Theta, a : K' \Rightarrow \Theta_1$.
3. $\mathsf{unify}(\Delta, \Theta, D\,A_1\,\ldots\,A_n, D\,B_1\,\ldots\,B_n)$: we need to show that types under the constructor D are pairwise identical after a substitution: $\theta(A_1) = \theta(B_1), \ldots, \theta(A_n) = \theta(B_n)$, where $n = \mathsf{arity}(D)$. We perform a nested induction, showing that for all $0 \le j \le n + 1$ the following holds: $\Delta \vdash \theta_j \vdash \Theta \Rightarrow \Theta_j$ and for all $1 \le i < j$ we have $\theta_j(A_i) = \theta_j(B_i)$.
For $j = 0$, this holds immediately.
In the inductive step, by definition of unify we have $\theta_{j+1} = \theta' \circ \theta_j$, and by the outer induction $\theta'(A_j) = \theta'(B_j)$ and $\Delta \vdash \theta' : \Theta_j \Rightarrow \Theta_{j+1}$. Together, we then have $\Delta \vdash \theta_{j+1} : \Theta \Rightarrow \Theta_{j+1}$. From Lemma G.1 we know that θ_{j+1} maintains equalities established by θ_j, and so we have $\theta_{j+1}(A_i) = \theta_{j+1}(B_i)$ for all $1 \le i < j + 1$.
From the definition of substitution we then have $\theta(D\,A_1\,\ldots\,A_n) = D\,\theta(A_1)\,\ldots\,\theta(A_n) = D\,\theta(B_1)\,\ldots\,\theta(B_n) = \theta(D\,B_1\,\ldots\,B_n)$, with $\Delta \vdash \theta : \Theta \Rightarrow \Theta_{n+1}$.

4. $\mathsf{unify}(\Delta, \Theta, \forall a.A, \forall b.B)$: In this case we must have

$$\begin{aligned} \mathsf{unify}((\Delta, c), \Theta, A[c/a], B[c/b]) &= (\Theta_1, \theta) \\ c &\# \Delta, \Theta && \textbf{(1)} \\ \mathsf{ftv}(B) \# c &\# \mathsf{ftv}(A) && \textbf{(2)} \\ c &\# \mathsf{ftv}(\theta) && \textbf{(3)} \end{aligned}$$

so from the inductive hypothesis we have $\theta(A[c/a]) = \theta(B[c/b])$ **(4)**, where c is fresh and $\Delta, c \vdash \theta : \Theta \Rightarrow \Theta_1$. We now derive:

$$\begin{array}{lll} & \theta(\forall a.A) & \\ = & \theta(\forall c.A[c/a]) & \text{(by (2) and (3), Lemma G.2)} \\ = & \forall c.\theta(A[c/a]) & \text{(by (1) and (3))} \end{array}$$

and by exactly the same reasoning, $\theta(\forall b.B) = \forall c.\theta(B[c/b])$. Then by (4) we can conclude $\theta(\forall a.A) = \forall c.\theta(A[c/a]) = \forall c.\theta(B[c/b]) = \theta(\forall b.B)$, which is the desired equality, and $\Delta \vdash \theta : \Theta_1 \Rightarrow \Theta$ because $c \notin \mathsf{ftv}(\theta)$ implies that we can remove it from Δ without damaging the well-formedness of θ.

□

H.2 Completeness of unification

Lemma H.1 (Unifiers are surjective)**.** *Let* $\mathsf{unify}(\Delta, \Theta, A, B) = (\Theta', \theta)$*. Then* $\mathsf{ftv}(\Theta') \subseteq \mathsf{ftv}(\Theta)$ *and for all* $b \in \Theta'$ *there exists* $a \in \Theta$ *such that* $b \in \mathsf{ftv}(\theta(a))$*.*

Proof. The first part follows immediately from the fact that in each case Θ', is always constructed from Θ by removing variables or demoting them.

For the second part, observe that θ' is constructed by manipulating appropriate identity functions. Mappings are only changed in the cases (a, A) and (A, a), such that $\theta(a) = A$. However, at the same time, a is removed from the output.

□

Theorem 5 (Unification is complete and most general)**.** *If* $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$ *and* $\Delta, \Theta \vdash A : K$ *and* $\Delta, \Theta \vdash B : K$ *and* $\theta(A) = \theta(B)$*, then* $\mathsf{unify}(\Delta, \Theta, A, B) = (\Theta'', \theta')$ *where there exists* θ'' *satisfying* $\Delta \vdash \theta'' : \Theta'' \Rightarrow \Theta'$ *such that* $\theta = \theta'' \circ \theta'$*.*

Proof. Via induction on the maximum of the sizes of A and B.

1. Case $A = a = B$: In this case $\mathsf{unify}(\Delta, \Theta, a, a)$ succeeds and returns $(\Theta, \iota_{\Delta,\Theta})$. Moreover, we may choose $\theta'' = \theta$ and conclude that $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$ and $\theta = \theta \circ \iota_{\Delta,\Theta}$, as desired.
2. Case $A = a \neq B$ or $B = b \neq A$. The two cases where one side is a variable are symmetric; we consider $A = a \neq B$.
Since $\theta(a) = \theta(B)$ for $B \neq a$, we must have that $a \in \Theta$. Thus, $\Theta = \Theta'_1, a : K'$ for some kind K' such that $K' \leq K$ (due to assumption $\Delta, \Theta \vdash A : K$). Also, since types are finite syntax trees we must have $a \neq \mathsf{ftv}(B)$ **(1)**. By assumption $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$, we have $\theta(a) : K'$ and by $\theta(a) = \theta(B)$ therefore also $\Delta, \Theta' \vdash \theta(B) : K'$ **(2)**.
We now define $\Theta_1 = \mathsf{demote}(K', \Theta'_1, \mathsf{ftv}(B) - \Delta)$ and choose θ'' to agree with θ on Θ_1, and undefined on a, yielding $\Delta \vdash \theta'' : \Theta'_1 \Rightarrow \Theta'$ **(3)**. By (1) we then have $\theta''(B) = \theta(B)$, making (2) equivalent to $\Delta, \Theta' \vdash \theta''(B) : K'$. We apply Lemma G.12, yielding $\Delta, \Theta_1 \vdash B : K'$
Hence unification succeeds in this case with $\mathsf{unify}(\Delta, \Theta, a, B) = (\Theta_1, \iota[a \mapsto B])$.
We strengthen (3) to $\Delta \vdash \theta'' : \Theta_1 \Rightarrow \Theta'$ by observing that for each $b \in \mathsf{ftv}(B) - \Delta$ (i.e., those variables potentially demoted to K' in Θ_1), we have $\Delta, \Theta' \vdash \theta(b) : K'$. If $K' = \star$ we have $K' = K$ by $K' \leq K$ and $\Delta, \Theta' \vdash \theta(b) : K'$ follows immediately. Otherwise, if $K' = \bullet$, then due to $b \in \mathsf{ftv}(B)$, $\theta(b)$ occurs in $\theta(B)$, and $\theta(b) : \star \geq K'$ would violate (2).
Clearly, $\theta'' \circ (\iota[a \mapsto B]) = (\theta'' \circ \iota)[a \mapsto \theta''(B)] = \theta$ since θ'' agrees with θ on all variables other than a, and $a \notin \mathsf{ftv}(B)$ as well as $\theta(a) = \theta(B)$.
3. $\theta(D\,A_1 \ldots A_n) = \theta(D\,B_1 \ldots B_n)$: by definition of substitution we have $\theta(A_i) = \theta(B_i)$, where $i \in 1, \ldots, n$ and $n \geq 0$. We perform a nested induction, showing that for all $0 \leq j \leq n+1$ the following holds: We have $\Delta \vdash \theta_j : \Theta \Rightarrow \Theta_j$ **(4)** and there exists θ''_j such that $\Delta \vdash \theta''_j : \Theta_n \Rightarrow \Theta'$ and $\theta''_j \circ \theta_j = \theta$ **(5)** as well as for all $1 \leq i < j$ unification of $\theta_i(A_i)$ and $\theta_i(B_i)$ succeeds.
    a. $j = 0$: unification succeeds with $\theta' = \theta_1 = \iota$ and the theorem holds for $\theta'' = \theta$ and $\Theta'' = \Theta$.
    b. $j \geq 1$: We use (5) to obtain $\theta''_j(\theta_j(A_j)) = \theta(A)$ **(6)** and $\theta''_j(\theta_j(B_j)) = \theta(B)$ **(7)**.
    We then have
    $$(\Theta_{j+1}, \theta'_{j+1}) = \mathsf{unify}(\Delta, \Theta_j, \theta_j(A_j), \theta_j(B_j))$$
    and $\theta_{j+1} = \theta'_{j+1} \circ \theta_j$ (by definition of unify).

By (4), (6) and (7) the outer induction shows that unification of $\theta_j(A_j)$ and $\theta_j(B_j)$ succeeds and there exists $\Delta \vdash \theta'' : \Theta_{j+1} \Rightarrow \Theta'$ such that $\theta'' \circ \theta'_{j+1} = \theta''_j$ **(8)**. By Theorem 4, we have $\Delta \vdash \theta'_{j+1} : \Theta_j \Rightarrow \Theta_{j+1}$ and hence by composition also $\Delta \vdash \theta_{j+1} : \Theta \Rightarrow \Theta_{j+1}$. Further, by (5) and (8), we have

$$(\theta'' \circ \theta'_{j+1}) \circ \theta_j = \theta''_j \circ \theta_j = \theta$$

Choosing $\theta''_{j+1} = \theta''$ then satisfies $\theta''_{j+1} \circ \theta_{j+1} = \theta$ and $\Delta \vdash \theta''_{j+1} : \Theta_{j+1} \Rightarrow \Theta'$.

4. $\theta(\forall a.A) = \theta(\forall b.B)$: we take fresh $c \notin \mathsf{ftv}(\theta, A, B)$. By Lemma G.2 and definition of substitution we have $\theta(A[c/a]) = \theta(B[c/b])$. By induction $\mathsf{unify}((\Delta, c), \Theta, A[c/a], B[c/b])$ succeeds with (Θ_1, θ') and there exist θ'' such that $\theta = \theta'' \circ \theta'$ **(9)** and $\Delta, c \vdash \theta'' : \Theta_1 \Rightarrow \Theta'$ **(10)**. The latter implies $c \notin \Delta, \Theta'$. By (9) and $c \notin \mathsf{ftv}(\theta)$ we have $c \notin \mathsf{ftv}(\theta')$.
This means that $\mathsf{unify}(\Delta, \Theta, \forall a.A, \forall b.B)$ succeeds with (Θ_1, θ')
We strengthen (10) to $\Delta \vdash \theta'' : \Theta_1 \Rightarrow \Theta'$ by showing that $c \notin \mathsf{ftv}(\theta'')$. Hence, assume $e \in \Theta$ such that $c \in \mathsf{ftv}(\theta''(e))$. By Lemma H.1, there exists $f \in \Theta$ such that $e \in \mathsf{ftv}(\theta'(f))$. This would imply $c \in \mathsf{ftv}(\theta''(\theta'(e))$, which by (9) contradicts $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$ and $c \notin \Delta, \Theta'$.

□

I Correctness of type inference proofs

This section contains proofs of correctness of the type inference algorithm. Observe that the proofs are mutually recursive in a well-founded way:

- The proofs in appendix I.1 are not mutually recursive among themselves. However, they use Theorems 6 and 7.
- The proof of Theorem 6 in appendix I.2 uses the lemmas from appendix I.1 only on subterms.
- Likewise, the proof of Theorem 7 in appendix I.3 uses Theorem 6 and the lemmas from appendix I.1 only on subterms.

I.1 Principality

In this subsection we collect together proofs of properties related to principality.

Lemma I.1 (Inferred types are principal)**.** *If* $\mathsf{infer}(\Delta, \Theta, \Gamma, M) = (\Theta', \theta, A)$ *and* $\Delta \Vdash M$ *and* $\Delta, \Theta \vdash \Gamma$ *then* $\mathsf{principal}((\Delta, \Theta' - \Delta'), \theta\Gamma, \Delta', A)$ *holds, where* $\Delta' = \mathsf{ftv}(A) - \Delta - \mathsf{ftv}(\theta)$.

Proof. By Theorem 6 we have $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$ **(1)** and $\Delta, \Theta'; \theta(\Gamma) \vdash M : A$ **(2)**. The latter implies $\Delta, \Theta' \vdash A$ and hence $\Delta' \subseteq \Theta'$. We can therefore rewrite (2) as $\Delta, (\Theta' - \Delta'), \Delta'; \theta(\Gamma) \vdash M : A$, satisfying the first condition of $\mathsf{principal}((\Delta, \Theta' - \Delta'), \theta\Gamma, \Delta', A)$.

By definition of Δ', we have $\Delta' \mathbin{\#} \mathsf{ftv}(\theta)$. We can therefore strengthen (1) to $\Delta \vdash \theta : \Theta \Rightarrow \Theta' - \Delta'$ **(3)**

Let Δ_p, A_p such that $\Delta_p = \mathsf{ftv}(A_p) - (\Delta, \Theta' - \Delta')$ and $\Delta, (\Theta' - \Delta'), \Delta_p \vdash M : A_p$ **(4)**. The latter implies $\Delta_p \mathbin{\#} \Delta, \Theta' - \Delta'$ and we can weaken (3) to $\Delta \vdash \theta : \Theta \Rightarrow (\Theta' - \Delta'), \Delta_p$ **(5)**.

Hence, we can apply Theorem 7, to (4) and (5), stating that there exists θ'' s.t. $\Delta \vdash \theta'' : \Theta' \Rightarrow (\Theta' - \Delta'), \Delta_p$ and $\theta''(A) = A_p$ and $\theta = \theta'' \circ \theta$.

The latter implies that for all $a \in \mathsf{ftv}(\theta)$, $\theta''(a) = a$ must hold. Hence, by defining δ as a restriction of θ'' such that $\delta(a) = \theta''(a)$ for all $a \in \mathsf{ftv}(A) - \Delta - \mathsf{ftv}(\theta)$ (i.e., Δ'), we get $\Delta \vdash \delta : \Delta' \Rightarrow_\star (\Theta' - \Delta'), \Delta_p$ and maintain $\delta(A) = A_p$. We rewrite the former to $\Delta, (\Theta' - \Delta') \vdash \delta : \Delta' \Rightarrow_\star \Delta_p$, obtaining an instantiation as required by the definition of principal. □

Lemma I.2 (Inferred types and principal types are isomorphic)**.** *Let the following conditions hold:*

$$\Delta, \Theta \vdash \Gamma \qquad \textbf{(1)}$$

$$\Delta \Vdash M \qquad \textbf{(2)}$$

$$\Delta' \mathbin{\#} \Theta' \qquad \textbf{(3)}$$

$$\mathsf{principal}((\Delta, \Theta), \Gamma, M, \Delta', A) \qquad \textbf{(4)}$$

$$\mathsf{infer}(\Delta, \Theta, \Gamma, M) = (\Theta', \theta, A') \qquad \textbf{(5)}$$

$$\Delta'' = \mathsf{ftv}(A') - \Delta - \mathsf{ftv}(\theta) \qquad \textbf{(6)}$$

Then there exists δ *such that* $\Delta, (\mathsf{ftv}(\theta) - \Delta) \vdash \delta : \Delta'' \Rightarrow_\bullet \Delta'$ *and* $\delta(\Delta'') = \Delta'$ *and* $\delta(A') = \theta(A)$.

Proof. By definition of principal we have $\Delta, \Theta, \Delta'; \Gamma \vdash M : A$ **(7)** and $\Delta' = \mathsf{ftv}(A) - \Delta, \Theta$.

Applying Theorem 6 to (5), we get $\Delta \vdash \Theta \Rightarrow \Theta'$ and $\Delta, \Theta'; \theta(\Gamma) \vdash M : A'$ **(8)**.

We have $\Delta \vdash \iota_{\Delta,\Theta} : \Theta \Rightarrow \Theta$ and therefore by $\Delta' \mathbin{\#} \Theta$ and weakening also $\Delta \vdash \iota_{\Delta,\Theta} : \Theta \Rightarrow \Theta, \Delta'$ **(9)**. Trivially, we can rewrite (7) and (4) as $\Delta, \Theta, \Delta'; \iota_{\Delta,\Theta}(\Gamma) \vdash M : A$ **(10)** and $\mathsf{principal}((\Delta, \Theta), \iota_{\Delta,\Theta}(\Gamma), M, \Delta', A)$ **(11)**, respectively. We can apply Theorem 7, using (2), (9) and (10), which yields existence of θ'' such that $\Delta \vdash \theta'' : \Theta' \Rightarrow \Theta, \Delta'$ and $\iota_{\Delta,\Theta} = \theta'' \circ \theta$ **(12)** and $\theta''(A') = A$ **(13)**. The latter implies that θ'' maps the type variables from Δ'' surjectively into Δ' **(14)**.

Let $\Theta_\theta = \mathsf{ftv}(\theta)$. By (12), we then have $\Delta' \# \Theta_\theta$ and θ is a bijection from Θ to Θ_θ. Conversely, the restriction of θ'' to Θ_θ is a bijection from Θ_θ to Θ.

We can therefore apply Lemma G.16(2) and obtain $\mathsf{principal}((\Delta, \Theta_\theta), \theta(\Gamma), M, \Delta', \theta(A))$ **(15)**.

By Lemma G.15 and $\Delta, \Theta_\theta \vdash \theta(\Gamma)$ as well as $\Delta, \Theta_\theta, \Delta'' \subseteq \mathsf{ftv}(A')$, we can strengthen (8) to $\Delta, \Theta_\theta, \Delta''; \theta(\Gamma) \vdash M : A'$ **(16)**.

We have $\mathsf{ftv}(\theta(A)) - (\Delta, \Theta_\theta) = \Delta' = \mathsf{ftv}(A) - (\Delta, \Theta)$. By definition of principal, (15) and (16) imposes that there exists δ_I such that $\Delta, \Theta_\theta \vdash \delta_I : \Delta' \Rightarrow_\bullet \Delta''$ and $\delta_I(\theta(A)) = A'$.

Using (13), we rewrite the latter to

$$\delta_I(\theta(\theta''(A')) = A' \qquad \textbf{(17)}$$

This implies that θ'' maps Δ'' not only surjectively (cf. (14)), but bijectively into Δ'. By (13) we further have $\theta''(\Delta'') = \Delta'$ (i.e., the order of variables is preserved).

Since θ is the identity on Δ', δ_I must be the inverse of θ'' on Δ'. Hence, we define δ such that $\delta(a) = \theta''(a)$ for all $a \in \Delta''$, yielding $\Delta, \Theta_\theta \vdash \delta : \Delta'' \Rightarrow_\bullet \Delta'$. As the inverse of δ_I, applying δ to both sides of (17) yields $\theta(\theta''(A')) = \theta(A) = \delta(A')$, which is the desired property.

□

Lemma I.3 (Stability of principality under substitution)**.** *Let the following conditions hold:*

$$\Delta, \Theta \vdash \Gamma \qquad \textbf{(1)}$$
$$\Delta' \# \Theta' \qquad \textbf{(2)}$$
$$\Delta \Vdash M \qquad \textbf{(3)}$$
$$\Delta \vdash \theta : \Theta \Rightarrow \Theta' \qquad \textbf{(4)}$$
$$\mathsf{principal}((\Delta, \Theta), \Gamma, M, \Delta', A) \qquad \textbf{(5)}$$

Then $\mathsf{principal}((\Delta, \Theta'), \theta\Gamma, M, \Delta', \theta A)$ *holds.*

Proof. By definition of principal, we have $\Delta, \Delta', \Theta; \Gamma \vdash M : A$ and $\Delta' = \mathsf{ftv}(A) - \Delta, \Theta$ **(6)**.

By (2), we can weaken (4) to $\Delta, \Delta' \vdash \Theta \Rightarrow \Theta'$. Together with the latter, we can then apply Lemma G.5 and obtain $\Delta, \Delta', \Theta' \vdash \theta A$.

Let $\Delta'' = \mathsf{ftv}(\theta A) - \Delta, \Theta'$. By (2), (4) and (6), Lemma G.14 yields $\Delta' = \Delta''$ **(7)**.

Let A_p and Δ_p such that $\Delta_p = \mathsf{ftv}(A_p) - \Delta, \Theta'$ and $\Delta, \Theta', \Delta_p; \theta\Gamma \vdash M : A_p$ **(8)**. Our goal is to show that there exists δ such that $\Delta, \Theta' \vdash \delta : \Delta'' \Rightarrow \Delta_p$ and $\delta(\theta A) = A_p$.

We weaken (4) to $\Delta \vdash \theta : \Theta \Rightarrow \Theta', \Delta_p$. We can then apply Theorem 7 to (8), which states that $\mathsf{infer}(\Delta, \Theta, \Gamma, M)$ returns (Θ'', θ', A') **(9)** and there exists θ'' such that

$$\Delta \vdash \theta'' : \Theta'' \Rightarrow \Theta', \Delta_p \qquad \textbf{(10)}$$
$$\theta = \theta'' \circ \theta' \qquad \textbf{(11)}$$
$$\theta''(A') = A_p \qquad \textbf{(12)}$$

By definition of infer, all type variables in Θ'' but not in Θ are fresh, which implies $\Delta' \# \Theta''$ **(13)**.

By Theorem 6, we have $\Delta \vdash \theta' : \Theta \Rightarrow \Theta''$ **(14)** and $\Delta, \Theta'' \vdash A'$ **(15)**.

We split Δ_p into a (possibly empty) part that is contained in Δ' and a remaining part that is not. Concretely, let Δ'_p, Δ''_p such that $\Delta_p \approx (\Delta'_p, \Delta''_p)$ and $\Delta'_p \subseteq \Delta'$ and $\Delta''_p \# \Delta'$. We weaken (14), (10), and (4), respectively:

$$\Delta, \Delta' \vdash \theta' : \Theta \Rightarrow \Theta'' \qquad \textbf{(16)}$$
$$\Delta, \Delta' \vdash \theta'' : \Theta'' \Rightarrow \Theta', \Delta''_p \qquad \textbf{(17)}$$
$$\Delta, \Delta' \vdash \theta : \Theta \Rightarrow \Theta', \Delta''_p \qquad \textbf{(18)}$$

Let $\Delta''' := \mathsf{ftv}(A') - \Delta - \mathsf{ftv}(\theta')$ **(19)**, which implies $\Delta''' \subseteq \Theta''$ **(20)** (using (14) and (15)). Further, let $\Theta_{\theta'} = \mathsf{ftv}(\theta') - \Delta$.

By (1), (3), (5), (9), (13) and (19), Lemma I.2 yields existence of δ_b such that $\Delta, \Theta_{\theta'} \vdash \delta_b : \Delta''' \Rightarrow_\bullet \Delta'$ **(21)** and $\delta_b(\Delta''') = \Delta'$ **(22)** and $\delta_b(A') = \theta' A$ **(23)**.

Let $\Delta' = (a_1, \ldots, a_n)$ and $\Delta''' = (b_1, \ldots, b_n)$. Let δ be defined such that for all $1 \leq i \leq n$, $\delta(a_i) = \theta''(b_i)$. By (7), (10) and (20) this yields $\Delta, \Theta' \vdash \delta : \Delta'' \Rightarrow_\star \Delta_p$.

Next, we show $\delta\theta''\delta_b(A') = \theta''(A')$ **(24)**: To this end, we show that for each $a \in \mathsf{ftv}(A')$ we have $\delta\theta''\delta_b(a) = \theta''(a)$. By the definition of Δ''' (cf. (19)) and (15), we have $\mathsf{ftv}(A') \subseteq \Delta, \Delta''', \Theta_{\theta'}$.

We consider three cases:

Case 1 $a = b_i \in \Delta'''$**:** We have $\delta_b(b_i) = a_i$ by (22). By $a_i \in \Delta'$ and (17) we have $\theta''(a_i) = \theta''(\delta_b(b_i)) = a_i$. By definition of δ, we have $\delta(a_i) = \delta(\theta''(\delta_b(b_i))) = \theta''(b_i)$.

Case 2 $a \in \Theta_{\theta'}$**:** We have $a \notin \Delta'''$ and therefore $\delta_b(a) = a$ by (21). By (14), we have $\Theta_{\theta'} \subseteq \Theta''$. Applying Lemma G.17 to (4) and (11) yields $\theta''(a) = \theta''(\delta_b(a)) = A$ for some A with $\Delta, \Theta' \vdash A$. By $\Delta'' = \Delta' \# \Delta, \Theta'$ we then have $\delta(A) = A$. In total, this yields $\delta\theta''\delta_b(a) = \theta''(\delta_b(a)) = \theta''(a)$.

Case 3 $a \in \Delta$**:** We have $\delta(a) = a$, $\delta_b(a) = a$ and $\theta''(a) = a$. This immediately yields $\delta\theta''\delta_b(a) = \theta''(a) = a$.

Finally, we show $\delta(\theta A) = A_p$:

$$\begin{array}{lll} & \delta\theta(A) & \\ = & \delta\theta''\theta'(A) & \text{(by (11) and (16) to (18))} \\ = & \delta\theta''\delta_b(A') & \text{(by (23))} \\ = & \theta''(A') & \text{(by (24))} \\ = & A_p & \text{(by (12))} \end{array}$$

□

Lemma I.4. *If* $\Delta \Vdash M$ *and* $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$ *and* $\Delta, \Theta; \Gamma \vdash M : A$*, then* $\Delta, \Theta'; \theta\Gamma \vdash M : \theta A$*.*

Proof. By induction on structure of M. In each case we apply inversion on derivations of $\Delta, \Theta; \Gamma \vdash M : A$ and $\Delta \Vdash M$ and start by showing the final steps in each derivation, then describe how to construct the needed conclusion.

- Case $M = \lceil x \rceil$. In this case we have derivations of the form:

$$\frac{x : A \in \Gamma}{\Delta, \Theta; \Gamma \vdash \lceil x \rceil : A} \qquad \frac{}{\Delta \Vdash \lceil x \rceil}$$

  Then we have $x : \theta(A) \in \theta(\Gamma)$, and may conclude

$$\frac{x : \theta(A) \in \theta(\Gamma)}{\Delta, \Theta'; \theta(\Gamma) \vdash \lceil x \rceil : \theta(A)}$$

- Case $M = x$. In this case, we have derivations of the form:

$$\frac{x : \forall \Delta'.H \in \Gamma \qquad \Delta, \Theta \vdash \delta : \Delta' \Rightarrow_{\star} \cdot}{\Delta, \Theta; \Gamma \vdash x : \delta(H)} \qquad \frac{}{\Delta \Vdash x}$$

  As before, we have $x : \theta(\forall \Delta'.H) \in \theta(\Gamma)$. Moreover, we can assume without loss of generality that the type variables in Δ' are fresh, so $\theta(\forall \Delta'.H) = \forall \Delta'.\theta(H)$. Since $\Delta, \Theta \vdash \Gamma$, we know that $\forall a \in \mathrm{ftv}(A).(\Delta, \Theta)(a) = \bullet$. Hence, for each such a, the substituted type $\theta(a)$ is a monotype, which implies that $\theta(H)$ is also a guarded type. Next, by Lemma G.7 we have $\Delta, \Theta' \vdash \theta \circ \delta : \Delta' \Rightarrow_{\star} \cdot$. We may conclude:

$$\frac{x : \forall \Delta'.\theta(H) \in \theta(\Gamma) \qquad \Delta, \Theta' \vdash \theta \circ \delta : \Delta' \Rightarrow_{\star} \cdot}{\Delta, \Theta'; \theta(\Gamma) \vdash x : \theta(\delta(H))}$$

  Note that in this case it is critical that we maintain the invariant (built into the context well-formedness judgement) that type variables in Γ are always of kind $\bullet$. This precludes substituting a type variable $a = H$ with a $\forall$-type, thereby changing the outer quantifier structure of $\forall \Delta'.H$.

- Case $M = \lambda x.M_0$. In this case we have derivations of the form:

$$\frac{\Delta, \Theta; \Gamma, x : S \vdash M_0 : B}{\Delta, \Theta; \Gamma \vdash \lambda x.M_0 : S \to B} \qquad \frac{\Delta \Vdash M_0}{\Delta \Vdash \lambda x.M_0}$$

  By induction, we have that $\Delta, \Theta'; \theta(\Gamma, a : S) \vdash M_0 : \theta B$. Moreover, clearly $\theta(\Gamma, a : S) = \theta(\Gamma), a : \theta(S)$. Since S is a monotype, and θ is a well-kinded substitution, and $\Delta, \Theta \Vdash \Gamma, x : S$, all of the free type variables in S are of kind $\bullet$ and are replaced with monotypes. Hence $\theta(S)$ is also a monotype, so we may derive:

$$\frac{\Delta, \Theta'; \theta(\Gamma), x : \theta(S) \vdash M_0 : \theta(B)}{\Delta, \Theta'; \theta(\Gamma) \vdash \lambda x.M_0 : \theta(S) \to \theta(B)}$$

  since $\theta(S \to B) = \theta(S) \to \theta(B)$.

- Case $M = \lambda(x : A_0).M_0$:

$$\frac{\Delta, \Theta; \Gamma, x : A_0 \vdash M_0 : B_0}{\Delta, \Theta; \Gamma \vdash \lambda(x : A_0).M_0 : A_0 \to B_0} \qquad \frac{\Delta \vdash A_0 : \star \qquad \Delta \Vdash M_0}{\Delta \Vdash \lambda(x : A_0).M_0}$$

By induction, we have that $\Delta, \Theta'; \theta(\Gamma, x : A_0) \vdash M_0 : \theta(B_0)$, and again $\theta(\Gamma, x : A_0) = \theta(\Gamma), x : \theta(A_0)$. Moreover, since $\Delta \vdash A_0 : \star$, we know that $\mathrm{ftv}(A_0) \subseteq \Delta$. Since the only variables substituted by θ are those in Θ, which is disjoint from Δ, we know that $\theta(A_0) = A_0$. Thus, we can proceed as follows:

$$\frac{\Delta, \Theta'; \theta(\Gamma), x : A_0 \vdash M_0 : \theta(B_0)}{\Delta, \Theta'; \theta(\Gamma) \vdash \lambda(x : A_0).M_0 : A_0 \to \theta(B_0)}$$

observing that $\theta(A_0 \to B_0) = \theta(A_0) \to \theta(B_0) = A_0 \to \theta(B_0)$, as required. This case illustrates part of the need for the $\Delta \Vdash M_0$ judgement: to ensure that the free type variables in terms are always treated rigidly and never “captured” by substitutions during unification or type inference.

- Case $M = M_0\ N_0$. In this case we proceed (refreshingly straightforwardly) by induction as follows.

$$\frac{\Delta, \Theta; \Gamma \vdash M_0 : A_0 \to B_0 \qquad \Delta, \Theta; \Gamma \vdash N_0 : A_0}{\Delta, \Theta; \Gamma \vdash M_0\ N_0 : B_0} \qquad \frac{\Delta \Vdash M_0 \qquad \Delta \Vdash N_0}{\Delta \Vdash M_0\ N_0}$$

By induction, we obtain the necessary hypotheses for the desired derivation:

$$\frac{\Delta, \Theta'; \theta(\Gamma) \vdash M_0 : \theta(A_0) \to \theta(B_0) \qquad \Delta, \Theta'; \theta(\Gamma) \vdash N_0 : \theta(A_0)}{\Delta, \Theta; \theta(\Gamma) \vdash M_0\ N_0 : \theta(B_0)}$$

again observing that $\theta(A_0 \to B_0) = \theta(A_0) \to \theta(B_0)$.

- Case $M = \mathbf{let}\ x = M_0\ \mathbf{in}\ N_0$. In this case we have derivations of the form:

$$\frac{\begin{array}{c}(\Delta', \Delta'') = \mathsf{gen}((\Delta, \Theta), A', M_0) \\ \Delta, \Theta, \Delta''; \Gamma \vdash M_0 : A' \qquad ((\Delta, \Theta), \Delta'', M_0, A') \Updownarrow A_0 \qquad \Delta, \Theta; \Gamma, x : A_0 \vdash N_0 : B \qquad \mathsf{principal}((\Delta, \Theta), \Gamma, M_0, \Delta'', A')\end{array}}{\Delta, \Theta; \Gamma \vdash \mathbf{let}\ x = M_0\ \mathbf{in}\ N_0 : B}$$

$$\frac{\Delta \Vdash M_0 \qquad \Delta \Vdash N_0}{\Delta \Vdash \mathbf{let}\ x = M_0\ \mathbf{in}\ N_0}$$

We assume without loss of generality that Δ'' is fresh with respect to Δ, Θ, and Θ'. This is justified as we may otherwise apply a substitution θ_F to $\Delta, \Theta, \Delta''; \Gamma \vdash M_0 : A'$ that replaces all variables in Δ'' by pairwise fresh ones. By induction, this would yield a corresponding typing judgement for M_0 using those fresh variables.
To apply the induction hypothesis to M_0, we need to extend θ to a substitution θ' satisfying $\Delta \vdash \theta' : \Theta, \Delta'' \Rightarrow \Theta', \Delta''$, which is the identity on all variables in Δ''. Then by induction we have $\Delta, \Theta', \Delta''; \theta'(\Gamma) \vdash M_0 : \theta'(A')$. Since θ' acts as the identity on Δ'' its behaviour is the same as θ weakened to $\Delta, \Delta'' \vdash \theta : \Theta \Rightarrow \Theta'$, so we have $\Delta, \Theta', \Delta''; \theta(\Gamma) \vdash M_0 : \theta(A')$.
We also obtain by the induction hypothesis for N_0 that $\Delta, \Theta'; \theta(\Gamma), x : \theta(A_0) \vdash N_0 : \theta(B)$, since $\theta(\Gamma, x : A_0) = \theta(\Gamma), x : \theta(A_0)$. By Lemma G.14(1), we have that $(\Delta', \Delta'') = \mathsf{gen}((\Delta, \Theta'), \theta(A'), M_0)$ and by Lemma G.14(2), we also know that $((\Delta, \Theta'), \Delta'', M_0, \theta(A')) \Updownarrow \theta(A_0)$. By applying Lemma I.3 to $\mathsf{principal}((\Delta, \Theta), \Gamma, M_0, \Delta'', A')$ we obtain $\mathsf{principal}((\Delta, \Theta'), \theta(\Gamma), M_0, \Delta'', \theta(A$
We can conclude:

$$\frac{\begin{array}{c}(\Delta', \Delta'') = \mathsf{gen}((\Delta, \Theta'), \theta(A'), M_0) \qquad \Delta, \Theta', \Delta''; \theta(\Gamma) \vdash M_0 : \theta(A') \\ ((\Delta, \Theta'), \Delta'', M_0, \theta(A')) \Updownarrow \theta(A_0) \qquad \Delta, \Theta'; \theta(\Gamma), x : \theta(A_0) \vdash N : \theta(B) \qquad \mathsf{principal}((\Delta, \Theta'), \theta(\Gamma), M_0, \Delta'', \theta(A'))\end{array}}{\Delta, \Theta'; \theta(\Gamma) \vdash \mathbf{let}\ x = M_0\ \mathbf{in}\ N_0 : \theta(B)}$$

- Case $M = \mathbf{let}\ (x : A_0) = M_0\ \mathbf{in}\ N_0$. In this case we have derivations of the form:

$$\frac{(\Delta', A') = \mathsf{split}(A_0, M_0) \qquad \Delta, \Theta, \Delta'; \Gamma \vdash M_0 : A' \qquad A_0 = \forall \Delta'.A' \qquad \Delta, \Theta; \Gamma, x : A_0 \vdash N_0 : B}{\Delta, \Theta; \Gamma \vdash \mathbf{let}\ (x : A_0) = M_0\ \mathbf{in}\ N_0 : B}$$

$$\frac{\Delta \vdash A_0 : \star \qquad (\Delta', A') = \mathsf{split}(A_0, M_0) \qquad \Delta, \Delta' \Vdash M_0 \qquad \Delta \Vdash N_0}{\Delta \Vdash \mathbf{let}\ (x : A_0) = M_0\ \mathbf{in}\ N_0}$$

We have $A_0 = \forall \Delta'.A'$ and $\Delta' \# \Delta$. According to $\Delta, \Delta' \Vdash M_0$, annotations in M_0 may use type variables from Δ, Δ'. By alpha-equivalence, we can assume $\Delta' \# \Theta$ and $\Delta' \# \Theta'$. Note that this may require freshening variables from Δ' (but not Δ) in M_0 as well.
By induction (and rearranging contexts), we have that $\Delta, \Theta', \Delta'; \theta(\Gamma) \vdash M_0 : \theta(A')$ and $\Delta, \Theta'; \theta(\Gamma, x : A_0) \vdash N_0 : \theta(B)$.

Moreover, since $\Delta \vdash A_0 : \star$, we know that $\theta(A_0) = A_0$ since θ only replaces variables in Θ, which is disjoint from Δ. Furthermore, $(\Delta', A') = \mathsf{split}(A_0, M_0)$ implies that A' is a subterm of A_0 so $\theta(A') = A'$ also. As a result, we can construct the following derivation:

$$\frac{(\Delta', A') = \mathsf{split}(A_0, M_0) \qquad \Delta, \Theta', \Delta'; \theta(\Gamma) \vdash M_0 : A' \qquad A_0 = \forall \Delta'.A' \qquad \Delta, \Theta'; \theta(\Gamma), x : A_0 \vdash N_0 : \theta(B)}{\Delta, \Theta'; \theta(\Gamma) \vdash \mathbf{let}\ (x : A_0) = M_0\ \mathbf{in}\ N_0 : \theta(B)}$$

□

Lemma I.5. *Let $\Delta' = (a_1, \dots, a_n)$ and $\Delta'' = (b_1, \dots, b_n)$ for some $n \geq 0$. Let $\Delta, \Theta \vdash \delta : \Delta' \Rightarrow \Delta''$ such that $\delta(a_i) = b_i$ for all $1 \leq i \leq n$. Furthermore, let $\Delta \Vdash M$ and* $\mathsf{principal}((\Delta, \Theta), \Gamma, M, \Delta', A)$ *and* $\Delta, \Theta \vdash \Gamma$.

Then $\mathsf{principal}((\Delta, \Theta), \Gamma, M, \Delta'', \delta A)$ *holds.*

Proof. We first show that $\Delta, \Theta, \Delta''; \Gamma \vdash M : \delta A$ holds. By $\mathsf{principal}((\Delta, \Theta), \Gamma, M, \Delta', A)$ we have $\Delta, \Theta, \Delta'; \Gamma \vdash M : A$. We extend δ to a substitution θ with $\Delta \vdash \theta : \Theta, \Delta' \Rightarrow \Theta, \Delta''$ by defining $\theta(a) = \delta(a)$ for all $a \in \Delta'$ and by defining θ as the identity on all $a \in \Theta$.

We apply Lemma G.16(1), yielding $\Delta, \Theta, \Delta''; \theta(\Gamma) \vdash M : \theta A$. We have $\theta(\Gamma) = \Gamma$ as well as $\theta(A) = \delta(A)$ and obtain the desired judgement.

Now, let Δ_p and A_p such that $\mathsf{ftv}(A_p) - \Delta, \Theta = \Delta_p$ and $\Delta, \Theta, \Delta_p; \Gamma \vdash M : A_p$. By $\mathsf{principal}((\Delta, \Theta), \Gamma, M, \Delta', A)$ we have that there exists an instantiation δ_p s.t. $\Delta, \Theta \vdash \delta_p : \Delta' \Rightarrow_{\bullet} \Delta_p$ and $\delta_p(A) = A_p$.

We need to show that then there also exists an instantiation δ'_p with $\Delta, \Theta \vdash \delta'_p : \Delta'' \Rightarrow_{\bullet} \Delta_p$ and $\delta'_p(\delta A) = A_p$. We observe that this holds for $\delta'_p = \delta_p \circ \delta^{-1}$, where δ^{-1} is the inverse of δ. □

Lemma I.6. *Let the following conditions hold:*

$$\Delta \Vdash M \tag{1}$$
$$\theta = \theta'' \circ \theta' \tag{2}$$
$$\Delta \vdash \theta : \Theta \Rightarrow \Theta' \tag{3}$$
$$\Delta \vdash \theta'' : \Theta'' \Rightarrow \Theta', \Delta'' \tag{4}$$
$$\Delta' = \mathsf{ftv}(A) - \Delta - \mathsf{ftv}(\theta') \tag{5}$$
$$\Delta, \Theta''; \theta'\Gamma \vdash M : A \tag{6}$$
$$\mathsf{principal}((\Delta, \Theta'), \theta\Gamma, \Delta'', A') \tag{7}$$
$$\theta''(A) = A' \tag{8}$$
$$\tag{9}$$

Then $\theta''(\Delta') = \Delta''$ holds.

Proof. By (7), we have $\Delta'' = \mathsf{ftv}(A') - \Theta' - \Delta$. Further, (6) yields $\Delta, \Theta'' \vdash A$ **(10)**.

Let $\Delta' = (a'_1, \dots, a'_n)$ for some $n \geq 0$ and let $\Delta_F = (f_1, \dots, f_n)$ for pairwise different, fresh type variables f_i.

By (10), we have $\mathsf{ftv}(A) \subseteq \Delta, \Theta''$. Let $\Theta_{\theta'}$ be defined as $\mathsf{ftv}(\theta') - \Delta$. We then have $\Theta_{\theta'} \subseteq \Theta''$ **(11)** and $\Delta' \mathbin{\#} \Theta_{\theta'}$ **(12)** and $\Delta' \subseteq \mathsf{ftv}(\Theta'')$ **(13)**.

By (2) to (4) we have $\Delta, \Theta' \vdash \theta''(a) : K$ for all $(a : K) \in \Theta_{\theta'}$ **(14)**.

Let θ''_F be defined such that

$$\theta''_F(a) = \begin{cases} \theta''(a) & \text{if } a \in \Theta_{\theta'} \\ f_i & \text{if } a = a'_i \in \Delta' \\ A_D & \text{if } a \in \Theta'' - \Theta_{\theta'} - \Delta' \end{cases} \tag{15}$$

where A_D is some arbitrary type with $\Delta, \Theta' \vdash A_D : \bullet$ (e.g., Int, cf. fig. 1).

By (11) to (13), this definition is well-formed. Together with (14) we then have $\Delta \vdash \theta''_F : \Theta'' \Rightarrow \Theta', \Delta_F$ **(16)** and $\theta = \theta''_F \circ \theta'$ **(17)**.

By (10) and Lemma G.5, we then have $\Delta, \Theta', \Delta_F \vdash \theta''_F A$ which implies $\mathsf{ftv}(\theta''_F A) \subseteq \Delta_F, \Delta, \Theta'$. In general, for every $a \in \mathsf{ftv}(A)$, $\theta''_F(a)$ is part of $\theta''_F(A)$. In particular, for each $a'_i \in \Delta' \subseteq \mathsf{ftv}(A)$, $\theta''_F(a'_i) = f_i$ occurs in $\theta''_F(A)$. Thus, $\mathsf{ftv}(\theta''_F A) - \Delta, \Theta' = \Delta_F$ holds **(18)**.

By (1), (6) and (16), Lemma I.4 yields $\Delta, \Theta', \Theta_F; \theta''_F \theta' \Gamma \vdash M : \theta''_F(A)$, which by (17) is equivalent to $\Delta, \Theta', \Delta_F; \theta\Gamma \vdash M : \theta''_F(A)$ **(19)**. By definition of principal as well as (7), (18) and (19) there exists δ such that $\Delta, \Theta' \vdash \delta : \Delta'' \Rightarrow \Delta_F$ **(20)** and $\delta(A') = \theta''_F(A)$.

By (8), the latter is equivalent to $\delta(\theta''(A)) = \theta''_F(A)$ **(21)**
Let $a \in \Delta' \subseteq \mathrm{ftv}(A)$, which implies $a = a'_i$ for some $1 \leq i \leq n$. By (21), we have

$$\begin{array}{lrcll} & \delta\theta''(a_i) & = & \theta''_F(a_i) & \\ \text{equiv.} & \delta\theta''(a_i) & = & f_i & \text{(by (15))} \end{array}$$

We therefore have that for each such a'_i, $\theta''(a_i)$ maps to pairwise different type variables b_i. By (20) and $\Delta, \Theta', \Delta'' \# \Delta_F$, we have $\delta(b_i) \neq b_i$ and therefore $b_i \in \Delta''$. We have therefore shown that θ'' maps Δ' injectively into Δ'' **(22)**.

We now show that θ'' is also surjective from Δ' into Δ'', which means that $\theta''(\Delta')$ is a permutation of Δ''. To this end, assume that there exists $b \in \Delta''$ such that there exists no $a \in \Delta'$ with $\theta''(a) = b$. By $b \in \Delta'' \subseteq \mathrm{ftv}(A')$ and (8) we have that there must exist $a \in \mathrm{ftv}(A)$ such that $b \in \mathrm{ftv}(\theta''(a))$. By (22), $a \in \Delta'$ would immediately yield a contradiction. By $\mathrm{ftv}(A) \subseteq \Theta_{\theta'}, \Delta', \Delta$, we therefore consider the cases $a \in \Theta_{\theta'}$ and $a \in \Delta$. If $a \in \Theta_{\theta'}$, according to (14), we then have $\mathrm{ftv}(\theta''(a)) \subseteq \Delta, \Theta'$, which is disjoint from Δ''. If $a \in \Delta$, we have $\theta''(a) = a \notin \Delta''$. As all choices for a yield contradictions, we have shown that $\theta''(\Delta')$ is a permutation of Δ''

We now show that $\theta''(\Delta') = \Delta''$ holds (i.e., θ'' preserves the order of type variables). To this end, let $a \in \mathrm{ftv}(A) - \Delta'$, which implies $a \in \Delta, \Theta_{\theta'}$. If $a \in \Delta$, then $\theta''(a) = a \in \Delta \# \Delta''$. If $a \in \Theta_{\theta'}$ then by (14) we have $\mathrm{ftv}(\theta''(a)) \subseteq \Delta, \Theta' \# \Delta''$. Therefore, together with (8) for all $a'_i, a'_j \in \Delta'$ with $1 \leq i < j \leq n$, we have that the first occurrence of $\theta''(a'_i)$ in A' is located before the first occurrence of $\theta''(a'_j)$ in A'.

□

I.2 Soundness of type inference

Lemma I.7. *If* $\Delta \Vdash M$ *and* $(\Theta', \theta, A) = \mathsf{infer}(\Delta, \Theta, \Gamma, M)$ *then for all* $a \in (\Theta - \mathrm{ftv}(\Gamma))$ *we have* $\theta(a) = a$ *and* $a \notin \mathrm{ftv}(A)$.

Proof. Straightforward by induction on the structure of M, in each case checking that a successful evaluation of type inference only instantiates free variables present in Γ. Furthermore, each type variable in A is either fresh or results from using a type in $\theta(\Gamma)$. □

Theorem 6. *If* $\Delta, \Theta \vdash \Gamma$ *and* $\Delta \Vdash M_0$ *and* $\mathsf{infer}(\Delta, \Theta, \Gamma, M_0) = (\Theta', \theta, A_0)$ *then* $\Delta, \Theta'; \theta(\Gamma) \vdash M_0 : A_0$ *and* $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$.

Proof. By induction on structure of M_0. In each case, we have $\Delta, \Theta \vdash \Gamma$ **(1)**, $\Delta \Vdash M_0$ **(2)**, and $\mathsf{infer}(\Delta, \Theta, \Gamma, M_0) = (\Theta', \theta, A_0)$. For each case, we show:

I. $\Delta, \Theta'; \theta\Gamma \vdash M_0 : A_0$
II. $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$

We write (*I*) and (*II*) to indicate that we have shown the respective statement.

Case $\lceil x \rceil$**:** By definition of infer, we have $A_0 = \Gamma(x)$, $\Theta' = \Theta$, and $\theta = \iota_{\Delta,\Theta}$, which implies $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$ (II) and $\Delta, \Theta' \vdash \Gamma$. We can then derive:

$$\frac{x : A_0 \in \Gamma}{\Delta, \Theta'; \Gamma \vdash \lceil x \rceil : A_0 \ (I)} \textsc{Freeze}$$

Case x**:** By definition of infer, we have $(x : \forall \overline{a}.H) \in \Gamma$ and $\overline{b} \# \Delta, \Theta$ and $A_0 = H[\overline{b}/\overline{a}]$ as well as $\Theta' = \Theta, \overline{b : \star}$. Due to α-equivalence, we can assume $\overline{a} \# \overline{b}, \Delta, \Theta$.

Let $\delta = [\overline{b}/\overline{a}]$. We have $\Delta, \Theta, \overline{b} \vdash \delta(a) : \star$ for all $a \in \overline{a}$ and therefore $\Delta, \Theta, \overline{b} \vdash \delta : (\overline{a : \bullet}) \Rightarrow_\star \cdot$. By (1) and $\overline{b} \# \Delta, \Theta$, we have $\Delta, \Theta, \overline{b : \star} \vdash \Gamma$ and derive the following:

$$\frac{x : \forall \overline{a}.H \in \Gamma \qquad \Delta, \Theta, \overline{b} \vdash \delta : (\overline{a : \bullet}) \Rightarrow_\star \cdot}{\Delta, \Theta, \overline{b : \star} \vdash x : \delta(H) \ (I)} \textsc{Var}$$

We weaken $\Delta \vdash \iota_{\Delta,\Theta} : \Theta \Rightarrow \Theta$ to $\Delta \vdash \iota_{\Delta,\Theta} : \Theta \Rightarrow \Theta, \overline{b : \star}$ (II).

Case $\lambda x.M$**:** By definition of infer, we have $a \# \Delta, \Theta$, which implies $a \# \mathrm{ftv}(\Gamma)$ **(3)**. Let $\theta_1 = \theta[a \longrightarrow S]$ **(4)**.

Together with (1) we then have $\Delta, \Theta, a : \bullet \vdash \Gamma, x : a$. By induction, we further have

$$\begin{array}{ll} & \Delta, \Theta_1; \theta_1(\Gamma, x : a) \vdash M : B \\ \text{equiv.} & \Delta, \Theta_1; \theta\Gamma, x : S \vdash M : B \quad \text{(by (3), (4))} \end{array} \qquad \textbf{(5)}$$

as well as $\Delta \vdash \theta_1 : (\Theta, a : \bullet) \Rightarrow \Theta_1$, which implies $\Delta \vdash \theta : \Theta \Rightarrow \Theta_1$ (II).

By (5) we have $\Delta, \Theta_1 \vdash \theta\Gamma$, which allows us to derive the following:

$$\frac{\Delta, \Theta_1; \theta\Gamma, x : S \vdash M : B \ \ (\text{by } (5))}{\Delta, \Theta_1; \theta\Gamma \vdash \lambda x.M : S \to B \ (I)} \textsc{Lam}$$

Case $\lambda(x : A).M$**:** By $\Delta \Vdash M_0$ we have $\Delta \vdash A$ **(6)**, and in particular all free type variables of A in the judgement $\Delta, \Theta \vdash A$ are monomorphic. Together with (1) this yields $\Delta, \Theta \vdash \Gamma, x : A$. Induction then yields $\Delta, \Theta_1; \theta(\Gamma, x : A) \vdash M : B$ **(7)** and $\Delta \vdash \theta : \Theta \Rightarrow \Theta_1$ (II).

According to (6) and the latter we further have $\theta(A) = A$ **(8)**. By (7) we have $\Delta, \Theta_1 \vdash \theta\Gamma$ and can derive the following:

$$\frac{\Delta, \Theta_1; \theta\Gamma, x : A \vdash M : B \ \ (\text{by } (7), (8))}{\Delta, \Theta_1; \theta\Gamma \vdash \lambda(x : A).M : A \to B \ (I)} \textsc{Lam-Ascribe}$$

Case $M\,N$**:** By definition of infer, we have:

$$(\Theta_1, \theta_1, A') = \mathsf{infer}(\Delta, \Theta, \Gamma, M) \qquad \textbf{(9)}$$
$$(\Theta_2, \theta_2, A) = \mathsf{infer}(\Delta, \Theta_1, \theta_1\Gamma, N) \qquad \textbf{(10)}$$

By induction, (9) yields $\Delta, \Theta_1; \theta_1\Gamma \vdash M : A'$ **(11)** and $\Delta \vdash \theta_1 : \Theta \Rightarrow \Theta_1$ **(12)**.

By (11) we have $\Delta, \Theta_1 \vdash \theta_1\Gamma$. Therefore, by induction, (10) yields $\Delta, \Theta_2; \theta_2\theta_1\Gamma \vdash N : A$ **(13)** and $\Delta \vdash \theta_2 : \Theta_1 \Rightarrow \Theta_2$ **(14)**. By definition of infer, we have:

$$b \mathrel{\#} \mathsf{ftv}(A') \quad b \mathrel{\#} \mathsf{ftv}(A) \quad b \mathrel{\#} \Theta \qquad \textbf{(15)}$$
$$(\Theta_3, \theta_3') = \mathsf{unify}(\Delta, (\Theta_2, b : \star), \theta_2 A', A \to b) \qquad \textbf{(16)}$$
$$\theta_3' = \theta_3[b \to B] \qquad \textbf{(17)}$$

By (11) we have $\Delta, \Theta_1 \vdash A'$ and by (14) further $\Delta, \Theta_2 \vdash \theta_2 A'$. This implies $\Delta, \Theta_2, b : \star \vdash \theta_2 A'$ by (15). By (13) we have $\Delta, \Theta_2 \vdash A$ and therefore also $\Delta, \Theta_2, b : \star \vdash A \to b$. Together, those properties allow us to apply Theorem 4, which gives us:

$$\begin{aligned} & \theta_3'\theta_2(A') = \theta_3'(A \to b) \\ \text{implies} \quad & \theta_3\theta_2(A') = \theta_3(A) \to B \quad (\text{by (15) and (17)}) \end{aligned} \qquad \textbf{(18)}$$

and

$$\begin{aligned} & \Delta \vdash \theta_3' : (\Theta_2, b : \star) \Rightarrow \Theta_3 \\ \text{implies} \quad & \Delta \vdash \theta_3 : \Theta_2 \Rightarrow \Theta_3 \qquad (\text{by (17)}) \end{aligned} \qquad \textbf{(19)}$$

By (14), (19), and composition, we have $\Delta \vdash \theta_3 \circ \theta_2 : \Theta_1 \Rightarrow \Theta_3$. By (11) and Lemma I.4, we then have $\Delta, \Theta_3; \theta_3\theta_2\theta_1\Gamma \vdash M : \theta_3\theta_2 A'$ **(20)**. Similarly, by (19), (13), and Lemma I.4, we have $\Delta, \Theta_3; \theta_3\theta_2\theta_1\Gamma \vdash N : \theta_3 A$ **(21)**

By (12), (14), (19), and Lemma G.6, we have $\Delta \vdash \theta_3\theta_2\theta_1\Gamma$. We can then derive:

$$\frac{\Delta, \Theta_3; \theta_3\theta_2\theta_1\Gamma \vdash M : \theta_3(A) \to B \ (\text{by } (20), (18)) \qquad \Delta, \Theta_3; \theta_3\theta_2\theta_1\Gamma \vdash N : \theta_3 A \ (\text{by } (21))}{\Delta, \Theta_3; \theta_3\theta_2\theta_1\Gamma \vdash M\,N : B \ (I)} \textsc{App}$$

Finally, we show $\Delta \vdash \theta_3 \circ \theta_2 \circ \theta_1 : \Theta \Rightarrow \Theta_3$. It follows from (12), (14), (19), and composition (II).

Case let $x = M$ **in** N**:** By definition of infer, we have $(\Theta_1, \theta_1, A) = \mathsf{infer}(\Delta, \Theta, \Gamma, M)$ **(22)**. By induction, this implies $\Delta, \Theta_1; \theta_1\Gamma \vdash M : A$ **(23)** and $\Delta \vdash \theta_1 : \Theta \Rightarrow \Theta_1$ **(24)**.

By definition of infer we further have

$$\begin{aligned} (\Delta'', \Delta''') &= \mathsf{gen}(\Delta', A, M) \\ &= \mathsf{gen}((\Delta, (\mathsf{ftv}(\theta_1\Theta) - \Delta)), A, M) \\ &\quad \text{where } \Delta''' = \mathsf{ftv}(A) - (\Delta, (\mathsf{ftv}(\theta_1) - \Delta)) \ = \ (\mathsf{ftv}(A) - \Delta) - \mathsf{ftv}(\theta_1) \ \textbf{(25)} \end{aligned}$$

By applying Lemma I.1 to (22), we obtain $\mathsf{principal}((\Delta, \Theta_1 - \Delta'''), \theta_1\Gamma, \Delta''', A)$ **(26)**.

We have $\Delta''' \subseteq \Theta_1$ and can therefore rewrite (23) as $\Delta, \Theta_1 - \Delta''', \Delta'''; \theta_1\Gamma \vdash M : A$ **(27)**.

Next, define $\Theta_1' = \mathsf{demote}(\bullet, \Theta_1, \Delta''')$. Again by definition of infer we have $(\Theta_2, \theta_2, B) = \mathsf{infer}(\Delta, \Theta_1' - \Delta'', (\theta_1(\Gamma), x : \forall\Delta''.A), N)$ **(28)**.

By definition of Δ''', we have $\Delta''' \mathrel{\#} \mathsf{ftv}(\theta_1)$ and thus $\Delta \vdash \theta_1 : \Theta \Rightarrow \Theta_1 - \Delta'''$ **(29)**.

We distinguish between M being a generalisable value or not. In each case, we show that there exist Δ_G'', Δ_G''', θ_2' and A' such that the following conditions are satisfied:

$$\theta_2' \circ \theta_1 = \theta_2 \circ \theta_1 \quad \textbf{(30)}$$

$$\Delta \vdash \theta_2' \circ \theta_1 : \Theta \Rightarrow \Theta_2 \quad \textbf{(31)}$$

$$(\Delta_G'', \Delta_G''') = \mathsf{gen}((\Delta, \Theta_2), \theta_2' A, M) \quad \textbf{(32)}$$

$$\Delta, \Theta_2, \Delta_G'''; \theta_2'\theta_1\Gamma \vdash M : \theta_2' A \quad \textbf{(33)}$$

$$(\Delta, \Theta_2), \Delta_G''', M, \theta_2' A) \Updownarrow A' \quad \textbf{(34)}$$

$$\Delta, \Theta_2; (\theta_2'\theta_1\Gamma, x : A') \vdash N : B \quad \textbf{(35)}$$

$$\mathsf{principal}((\Delta, \Theta_2), \theta_2'\theta_1\Gamma, M, \Delta_G''', \theta_2' A) \quad \textbf{(36)}$$

Sub-Case $M \in \mathsf{GVal}$**:** By definition of gen, we have $\Delta'' = \Delta'''$. We choose $\Delta_G'' := \Delta'''$ and $\Delta_G''' := \Delta'''$.
In order to apply the induction hypothesis to (28), we need to show $\Delta, \Theta_1' - \Delta'' \vdash \theta_1(\Gamma), x : \forall \Delta'''.A$. First, by (1) and (29) and Lemma G.6, we have $\Delta, \Theta_1 - \Delta'' \vdash \theta_1(\Gamma)$.
Second, by (23) we have $\Delta, \Theta_1 \vdash A$ and thus $\Delta, \Theta_1 - \Delta''' \vdash \forall \Delta'''.A$. It remains to show that for all $a \in \mathsf{ftv}(A) - \Delta'''$ we have $\Delta, \Theta_1 \vdash a : \bullet$. For $a \in \Delta$, this follows immediately. Otherwise, we have $a \in \Theta_1 - \Delta'''$ and $a \in \mathsf{ftv}(\theta_1)$, which implies that there exists $b \in \Theta$ such that $a \in \mathsf{ftv}(b)$. If $b \in \mathsf{ftv}(\Gamma)$, then by $\Delta, \Theta \vdash \Gamma$ we have $\Delta, \Theta_1 \vdash \theta(b) : \bullet$, which implies $\Delta, \Theta_1 \vdash a : \bullet$. Otherwise, if $b \notin \mathsf{ftv}(\Gamma)$, then by Lemma I.7 we have $\theta(b) = b = a$ and $a \notin \mathsf{ftv}(A)$, contradicting our earlier assumption. By $\Theta_1 - \Delta'' = \Theta_1' - \Delta''$ we then have $\Delta, \Theta_1' - \Delta'' \vdash \theta_1(\Gamma), x : \forall \Delta''.A$
In summary, we can apply the induction hypothesis by which we then have $\Delta, \Theta_2, \theta_2(\theta_1(\Gamma, x : \forall \Delta''.A) \vdash N : B$ **(37)** and $\Delta \vdash \theta_2 : \Theta_1 - \Delta''' \Rightarrow \Theta_2$ **(38)**. We choose $\theta_2' = \theta_2$ and $A' = \forall \Delta'''.\theta_2' A$, therefore satisfying (30) and (34).
By (29) and (38), condition (31) is also satisfied.
No type variable in Δ''' is freely part of the input to infer that resulted in (28). As all newly created variables are fresh, we then have $\Delta''' \# \Theta_2$ **(39)**.
Due to our choice of θ_2' we have $\theta_2'(\theta_1(\Gamma) = \theta_2(\theta_1\Gamma)$ and by (38) and (39) also $\theta_2(\forall \Delta'''.A) = \forall \Delta'''.\theta_2'(A)$. Therefore, (37) is equivalent to (35).
By applying Lemma I.3 to (26), (38) and (39) we show that (36) is satisfied.
Recall the following relationships:

$$\begin{aligned} \mathsf{ftv}(A) &\subseteq \Delta, \Theta_1 \\ \mathsf{ftv}(\theta) &\subseteq \Delta, \Theta_1 \\ \Delta''' = \mathsf{ftv}(A) - \Delta - \mathsf{ftv}(\theta) &\subseteq \Theta_1 \end{aligned}$$

Therefore, $\mathsf{ftv}(A) - \Delta - \mathsf{ftv}(\theta)$ (i.e., Δ''') is equal to $\mathsf{ftv}(A) - \Delta, (\Theta_1 - \Delta''')$. This results in $(\Delta'', \Delta''') = \mathsf{gen}((\Delta, \Theta_1 - \Delta'''), A, M)$ **(40)**. Together with (38) and $\Delta''' \# \Theta_2$ we can then apply Lemma G.14(1) to (40), yielding satisfaction of (32).
By applying Lemma I.4 to (27) and (38), we obtain (33).

Sub-Case $M \notin \mathsf{GVal}$**:** By definition of gen, we have $\Delta'' = \cdot$. Let Δ''' have the shape $(a_1, \ldots, a_n)$. We choose $\Delta_G'' := \cdot$ and $\Delta_G''' := (b_1, \ldots, b_n)$ for n pairwise different, fresh type variables b_i.
We show that the induction hypothesis is applicable to (28). To this end, we show $\Delta, \Theta_1' - \Delta'' \vdash \theta_1\Gamma, x : \forall \Delta''.A$. We have $\Delta, \Theta_1 \vdash \theta_1\Gamma$ and $\Delta, \Theta_1 \vdash A$ by (23). It remains to show that for all $a \in \mathsf{ftv}(A)$ we have $(a : \bullet) \in \Delta, \Theta_1'$. If $a \in \Delta'''$, then by definition of Θ_1' we have $(a : \bullet) \in \Theta_1'$. Otherwise, if $a \in \Theta_1 - \Delta'''$, we use the same reasoning as in the case $M \in \mathsf{GVal}$.
By induction, we then have $\Delta, \Theta_2; \theta_2(\theta_1(\Gamma), x : A) \vdash N : B$ **(41)** and $\Delta \vdash \theta_2 : \Theta_1' \Rightarrow \Theta_2$. By Lemma G.10 the latter implies $\Delta \vdash \theta_2 : \Theta_1 \Rightarrow \Theta_2$ **(42)**.
We define θ_2' such that

$$\theta_2'(c) = \begin{cases} b_i & \text{if } c = a_i \in \Delta''' \\ \theta_2(c) & \text{if } c \in \Theta_1 - \Delta''' \end{cases}$$

By (42) and the definition of Δ''' we then have $\Delta \vdash \theta_2' : \Theta_1 \Rightarrow \Theta_2, \Delta_G'''$ **(43)**. Observe that we have $\theta_2'(a) = \theta_2(a)$ for all $a \in \mathsf{ftv}(\theta_1) - \Delta$ and therefore (30) as well as (31) are satisfied.
Furthermore, we define θ_2'' such that $\theta_2''(a) = \theta_2(a)$ for all $a \in \Theta_1 - \Delta'''$, which implies $\Delta \vdash \theta_2'' : \Theta_1 - \Delta''' \Rightarrow \Theta_2$ **(44)** and $\theta_2'' \circ \theta_1 = \theta_2 \circ \theta_1$ **(45)**.

We define the instantiation δ such that $\delta(b_i) = \theta_2(a_i)$ for all $a_i \in \Delta'''$. By definition of Θ'_1 and (42) we then have $\Delta, \Theta_2 \vdash \delta(b_i) : \bullet$ for all $b_i \in \Delta'''_G$. This implies $\Delta \vdash \delta : \Delta'''_G \Rightarrow_{\bullet} \Theta_2$.

We define $A' := \delta(\theta'_2(A))$, which is identical to $\theta_2(A)$. Together with $\theta_2(\theta_1\Gamma) = \theta'_2(\theta_1\Gamma)$, this choice satisfies (34) and makes (41) equivalent to (35).

We have $\mathsf{ftv}(\theta_2 A) \subseteq \Delta, \Theta_2$ and $\Delta''' \subseteq \mathsf{ftv}(A)$. By $\theta'_2(\Delta''') = \Delta'''_G$ we have $\Delta'''_G \subseteq \mathsf{ftv}(\theta'_2 A)$. Together with $\mathsf{ftv}(\theta'_2(a)) = \mathsf{ftv}(\theta_2(a)) \# \Delta'''_G$ holding for all $a \in \mathsf{ftv}(A) - \Delta'''$, we then have $\mathsf{ftv}(\theta'_2 A) - \Delta, \Theta_2 = \Delta'''_G$. Therefore, we have $\mathsf{gen}((\Delta, \Theta_2), \theta'_2 A, M) = (\cdot, \Delta'''_G)$, satisfying (32).

Let δ_F be defined such that $\delta(a_i) = b_i$ for all $1 \leq i \leq n$, which implies $\Delta, \Theta \vdash \delta_F : \Delta''' \Rightarrow_{\bullet} \Delta'''_G$ and $\theta''_2\delta_F(A) = \theta'_2(A)$ **(46)** (by weakening θ''_2 such that $\Delta, \Delta'''_G \vdash \theta''_2 : \Theta_1 - \Delta''' \Rightarrow \Theta_2$) Using Lemma I.5, we then get $\mathsf{principal}((\Delta, \Theta_1 - \Delta'''), \theta_1\Gamma, M, \Delta'''_G, \delta_F A)$,

We apply Lemma I.3 to this freshened principality statement and (44), which gives us $\mathsf{principal}((\Delta, \Theta_2), \theta''_2\theta_1\Gamma, M, \Delta''', \theta''_2\delta_F A)$.

Using (46), we restate this as $\mathsf{principal}((\Delta, \Theta_2), \theta''_2\theta_1\Gamma, M, \Delta'''_G, \theta'_2 A)$, which by (30) and (45) is equivalent to (36).

By applying Lemma I.4 to (23) and (43), we obtain (33).

We have shown that (30) to (36) hold in each case. We can now derive the following:

$$\frac{\begin{array}{c}(\Delta''_G, \Delta'''_G) = \mathsf{gen}((\Delta, \Theta_2), \theta'_2 A, M) \text{ (by (32))} \\ \Delta, \Theta_2, \Delta'''_G; \theta'_2\theta_1\Gamma \vdash M : \theta'_2 A \text{ (by (33))} \\ ((\Delta, \Theta_2), \Delta'''_G, M, \theta'_2 A) \Updownarrow A' \text{ (by (34))} \\ \Delta, \Theta_2; (\theta'_2\theta_1\Gamma, x : A') \vdash N : B \text{ (by (35))} \\ \mathsf{principal}((\Delta, \Theta_2), \theta'_2\theta_1\Gamma, M, \Delta'''_G, \theta'_2 A) \text{ (by (36))}\end{array}}{\Delta, \Theta_2; \theta'_2\theta_1\Gamma \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : B}\ \textsc{Let}$$

By (30) and (31) we have therefore shown (I) and (II).

Case let $(x : A) = M$ **in** N : Let $A = \forall\Delta''.H$ for appropriate Δ'' and H. By alpha-equivalence, we assume $\Delta'' \# \Theta$. According to (2), we have $\Delta \vdash A$ **(47)**.

We distinguish between whether of not M is a guarded value. We show that the following conditions hold for the choice of A' and Δ' imposed by $(\Delta', A') = \mathsf{split}(A, M)$ **(48)** in each case.

$$\Delta, \Delta' \vdash A' \qquad \textbf{(49)}$$

$$\mathsf{ftv}(A) \# \Delta' \qquad \textbf{(50)}$$

$$\Delta' \# \Theta \qquad \textbf{(51)}$$

Sub-Case $M \in \mathsf{GVal}$**:** We have $\mathsf{split}(A, M) = (\Delta'', H)$ (i.e, $\Delta' = \Delta''$ and $A' = H$).

Together with (47) we have $\Delta, \Delta' \vdash H$ (satisfying (49)). Assumption $\Delta'' \# \Theta$ satisfies (51). By $A = \forall\Delta'.A'$ we further have $\mathsf{ftv}(A) \# \Delta'$.

Sub-Case $M \notin \mathsf{GVal}$**:** We have $\mathsf{split}(A, M) = (\cdot, A)$ (i.e, $\Delta' = \cdot$ and $A' = A$). This immediately satisfies (50) and (51). It further makes (47) equivalent to (49).

Moreover, by (2), we have $\Delta, \Delta' \Vdash M$ **(52)** using inversion.

We show that $\Delta, \Theta_1, \Delta'; \theta_1\Gamma \vdash M : A_1$ **(53)** holds. By (1) and since $\Delta' \# \Theta$, we have $\Delta, \Delta', \Theta \vdash \Gamma$. Together with (52), we then have $\Delta, \Delta', \Theta_1; \theta'_1\Gamma \vdash M : A_1$ and $\Delta, \Delta' \vdash \theta_1 : \Theta \Rightarrow \Theta_1$ **(54)** by induction. Further, this indicates $\Delta' \# \Theta_1$.

By (53) we also have $\Delta, \Delta', \Theta_1 \vdash A_1$. Recall $\Delta, \Delta' \vdash A'$ and therefore $\Delta, \Delta', \Theta_1 \vdash A'$. Thus, by Theorem 4, we have $\theta'_2(A_1) = \theta'_2(A')$ **(55)** and $\Delta, \Delta' \vdash \theta'_2 : \Theta_1 \Rightarrow \Theta_2$ **(56)**.

According to the assertion, we have $\mathsf{ftv}(\theta'_2 \circ \theta_1) \# \Delta'$ **(57)**. By definition of infer, we have $\theta_2 = \theta'_2 \circ \theta_1$ **(58)**, yielding $\Delta, \Delta' : \theta_2 : \Theta \Rightarrow \Theta_2$, which further implies $\Delta' \# \Theta_2$**(59)**. By (57), we can strengthen θ_2 s.t. $\Delta \vdash \theta_2 : \Theta \Rightarrow \Theta_2$ **(60)**.

By (52), (53) and (56), and Lemma I.4, we have $\Delta, \Delta', \Theta_2; \theta'_2\theta_1\Gamma \vdash M : \theta'_2 A_1$. By (54), (56) and (58), this is equivalent to $\Delta, \Delta', \Theta_2; \theta_2\Gamma \vdash M : \theta'_2 A_1$ **(61)**.

By (49) and (56) we have $\theta'_2(A') = A'$. Together with (55), this makes (61) equivalent to $\Delta, \Delta', \Theta_2; \theta_2\Gamma \vdash M : A'$ **(62)**.

By definition of infer, we have $(\Theta_3, \theta_3, B) = \mathsf{infer}(\Delta, \Theta_2, (\theta_2\Gamma, x : A), N)$. Due to (2), we have $\Delta \Vdash N$. By (47) and $\Theta_2 \# \Delta$, we have $\Delta, \Theta_2 \vdash x : A$. Together with (1) and (60) we then have $\Delta, \Theta_2 \vdash (\theta_2\Gamma, x : A)$. Therefore, by induction, we have $\Delta, \Theta_3; \theta_3(\theta_2\Gamma, x : A) \vdash N : B$ **(63)** and $\Delta \vdash \theta_3 : \Theta_2 \Rightarrow \Theta_3$ **(64)**.

According to (50) and (59), none of the variables in Δ' are freely part of the input to infer, yielding $\Delta' \# \Theta_3$. Together with (59), we can then weaken (64) to $\Delta, \Delta' \vdash \theta_3 : \Theta_2 \Rightarrow \Theta_3$. By the latter, (62), (52), and Lemma I.4, we have $\Delta, \Theta_3, \Delta'; \theta_3\theta_2\Gamma \vdash M : \theta_3 A'$ **(65)**.

Using a similar line of reasoning as before, we have $\theta_3(A') = A'$ **(66)** and $\theta_3(A) = A$ **(67)**.

By (60), (64), and composition, we have $\Delta \vdash \theta_3 \circ \theta_2 : \Theta \Rightarrow \Theta_3$. (II).
Together with (1), we obtain $\Delta, \Theta_3 \vdash \theta_3\theta_2\Gamma$ and can derive the following:

$$\dfrac{\begin{array}{c}(\Delta', A') = \mathsf{split}(A, M) \text{ (by (48))} \\ A = \forall\Delta'.A' \text{(by (48))} \\ \Delta, \Theta_3, \Delta'; \theta_3\theta_2\Gamma \vdash M : A' \text{ (by (65) and (66))} \\ \Delta, \Theta_3; \theta_3\theta_2\Gamma, x : A \vdash N : B \text{ (by (63) and (67))}\end{array}}{\Delta, \Theta_3; \theta_3\theta_2\Gamma \vdash \mathbf{let}\ (x : A) = M\ \mathbf{in}\ N : B\ (I)} \textsc{Let-Ascribe}$$

□

I.3 Completeness of type inference

Theorem 7 (Type inference is complete and principal)**.** *Let* $\Delta \Vdash M_0$ *and* $\Delta, \Theta \vdash \Gamma$. *If* $\Delta \vdash \theta_0 : \Theta \Rightarrow \Theta'$ *and* $\Delta, \Theta'; \theta_0(\Gamma) \vdash M_0 : A_0$, *then* $\mathsf{infer}(\Delta, \Theta, \Gamma, M_0) = (\Theta'', \theta', A_R)$ *where there exists* θ'' *satisfying* $\Delta \vdash \theta'' : \Theta'' \Rightarrow \Theta'$ *such that* $\theta_0 = \theta'' \circ \theta'$ *and* $\theta''(A_R) = A_0$.

Proof. By induction on the structure of M_0. In each case, we assume $\Delta \Vdash M_0$ **(1)**, and $\Delta, \Theta \vdash \Gamma$ **(2)**, and $\Delta \vdash \theta_0 : \Theta \Rightarrow \Theta'$ **(3)**, and $\Delta, \Theta'; \theta_0\Gamma \vdash M_0 : A_0$ **(4)**, which implies $\Delta, \Theta' \vdash \theta_0\Gamma$ **(5)**, and $\Delta, \Theta' \vdash A_0$ **(6)**. For each case, we show:

I. $\mathsf{infer}(\Delta, \Theta, \Gamma, M_0) = (\Theta'', \theta', A_R)$
II. $\Delta \vdash \theta'' : \Theta'' \Rightarrow \Theta'$
III. $\theta_0 = \theta'' \circ \theta'$
IV. $\theta''(A_R) = A_0$

We reference the proof obligations above to indicate when we have shown them.

Case $\lceil x \rceil$**:** By (4) and Freeze, we have $(x : A_0) \in \theta_0\Gamma$. infer succeeds, and we have $\Theta'' = \Theta$, $\theta' = \iota_{\Delta,\Theta}$, and $A_R = \Gamma(x)$. The latter implies $A_0 = \theta_0(A_R)$.
We have $\Delta \vdash \theta' : \Theta \Rightarrow \Theta$. Let $\theta'' := \theta_0$. By (3) we then have $\Delta \vdash \theta'' : \Theta \Rightarrow \Theta'$ (II). We observe $\theta_0 = \theta'' = \theta'' \circ \iota_{\Delta,\Theta} = \theta'' \circ \theta'$ (III).
Finally, this yields $\theta''(A_R) = \theta_0(A_R) = A_0$ (IV).

Case x**:** The derivation for (4) must be of the following form:

$$\dfrac{\begin{array}{l}\textsc{Var} \\ x : \forall\Delta'.H' \in \theta_0\Gamma \qquad \Delta, \Theta' \vdash \delta : \Delta' \Rightarrow_\star \cdot\end{array}}{\Delta, \Theta'; \theta_0\Gamma \vdash x : \delta(H')}$$

Therefore, there exists $x : \forall\Delta''.H \in \Gamma$ such that $\forall\Delta'.H' = \theta_0(\forall\Delta''.H)$. By alpha-equivalence, we assume that Δ'' is fresh, yielding $\theta_0(\forall\Delta''.H) = \forall\Delta''.\theta_0 H$. By (2), all free type variables in H are monomorphic, meaning that $\theta_0 H$ cannot have toplevel quantifiers. Thus, the quantifier structure is preserved by θ_0; in particular $\Delta' = \Delta''$, $H' = \theta_0(H)$.
Further, due to our freshness assumption about $\Delta'' = \Delta'$, we have $\Delta' \# \Delta, \Theta$ **(7)** and $\Delta' \# \Delta, \Theta'$.
In total, we have $A_0 = \delta\theta_0 H$ **(8)** and $\Gamma(x) = \forall\Delta'.H$ **(9)** and $\theta_0\Gamma(x) = \theta_0(\forall\Delta'.H) = \forall\Delta'.\theta_0 H$ **(10)**.
Let $\Delta' = \overline{a} = (a_1, \dots, a_n)$ with corresponding fresh $\overline{b} = (b_1, \dots, b_n)$ for some $n \geq 0$. Then infer succeeds with $\Theta'' = (\Theta, \overline{b : \star})$, and $\theta' = \iota_{\Delta,\Theta}$ **(11)**, and $A_R = H[\overline{b}/\overline{a}]$ **(12)**. Due to $\Theta \subseteq \Theta''$ and the freshness of $\overline{b}$, we have $\Delta \vdash \theta' : \Theta \Rightarrow \Theta''$ **(13)**.
We define θ'' such that

$$\theta''(c) = \begin{cases} \theta_0(c) & \text{if } c \in \Theta \\ \delta(a_i) & \text{if } c = b_i \text{ for some } b_i \in \overline{b} \end{cases} \tag{14}$$

By (3) and (14), for all $(c : K) \in \Theta$ we have $\Delta, \Theta' \vdash \theta_0(c) : K$ **(15)**. By $\Delta, \Theta' \vdash \delta : \Delta' \Rightarrow_\star \cdot$, we have $\Delta, \Theta' \vdash \delta(a) : \star$ for all $a \in \Delta'$ and thus $\Delta, \Theta' \vdash \theta''(b) : \star$ for all $b \in \overline{b}$. Together, we then have $\Delta \vdash \theta'' : \Theta'' \Rightarrow \Theta'$ (II).
By (11) and (14), we have $\theta''\theta'(c) = \theta''(c) = \theta_0(c)$ for all $c \in \Theta$ (III).
It remains to show that $\theta''(H[\overline{b}/\overline{a}]) = A_0 = \delta(\theta_0(H))$.
By (2) and (9), we have $\Delta, \Theta \vdash \forall\Delta'.H$ and further $\Delta, \Theta, \Delta' \vdash H$.
We show that for all $c \in \mathsf{ftv}(H) \subseteq \Delta, \Theta, \Delta'$, we have $\theta''(c[\overline{b}/\overline{a}]) = \delta\theta_0(c)$ **(16)**. We distinguish three cases:
1. Let $c = a_i \in \Delta'$. We then have

$$\begin{array}{rcll} a_i & = & \theta_0(a_i) & \text{(by (3))} \\ \text{implies} \quad \theta''(a_i[\overline{b}/\overline{a}]) & = & \delta(\theta_0(a_i)) & \qquad \text{(by (14): } \theta''(b_i) = \delta(a_i)) \end{array}$$

2. Let $c \in \Theta$. We then have

$$\begin{array}{rrcll} & \theta_0(c) & = & \theta_0(c) & \\ \text{implies} & \theta_0(c[\overline{b}/\overline{a}]) & = & \theta_0(c) & \text{(by (7): } c[\overline{b}/\overline{a}] = c) \\ \text{implies} & \theta''(c[\overline{b}/\overline{a}]) & = & \delta\theta_0(c) & \text{(by (14) : } \theta''(c) = \delta(c) \text{ for all } c \in \Theta) \end{array}$$

3. Let $c \in \Delta$. Then all involved substitutions/instantiations return c unchanged.
By (12) and (8), (16) then yields $\theta''(A_R) = A_0$ (IV).

Case $\lambda x.M$**:** By (4) and Lam, we have $A_0 = S' \to B'$ for some S', B' as well as $\Delta, \Theta'; \theta_0\Gamma, (x : S') \vdash M : B'$ **(17)**. The latter implies $\Delta, \Theta' \vdash S' : \bullet$ **(18)**.
Let a be the fresh variable as in the definition of infer; in particular $a \# \Theta$ **(19)**. Let θ_a be defined such that $\theta_a(b) = \theta_0(b)$ for all $b \in \Theta$ **(20)** and $\theta_a(a) = S'$ **(21)**. By (3) and (18), we have $\Delta \vdash \theta_a : (\Theta, a : \bullet) \Rightarrow \Theta'$**(22)**. This definition makes (17) equivalent to $\Delta, \Theta'; \theta_a(\Gamma, x : a) \vdash M : B'$.
By induction, we therefore have that infer$(\Delta, (\Theta, a : \bullet), (\Gamma, x : a), M)$ succeeds **(23)**, returning (Θ_1, θ'_1, B) and there exists θ''_1 s.t.

$$\Delta \vdash \theta''_1 : \Theta_1 \Rightarrow \Theta' \quad \textbf{(24)}$$

$$\theta_a = \theta''_1 \circ \theta'_1 \quad \textbf{(25)}$$

$$\theta''_1(B) = B' \quad \textbf{(26)}$$

By Theorem 6, we have $\Delta \vdash \theta'_1 : (\Theta, a : \bullet) \Rightarrow \Theta_1$ **(27)**[4]. By preservation of kinds under substitution, we have $\Delta, \Theta_1 \vdash \theta'_1(a) : \bullet$. This implies that $\theta'_1(a)$ is a syntactic monotype. Thus, $\theta'_1 = \theta[a \mapsto S]$ **(28)** is well-defined, yielding a substitution $\Delta \vdash \theta : \Theta \Rightarrow \Theta_1$. Hence, all steps of infer succeed.
According to the return values of infer, we have $A_R = S \to B$, $\Theta'' = \Theta_1$, and $\theta' = \theta$ **(29)**.
Let θ'' be defined as θ''_1 **(30)**. By (24), this choice immediately satisfies (II).
We show (III) as follows: Let $b \in \Theta$. We then have

$$\begin{array}{cll} & \theta_0(b) & \\ = & \theta_a(b) & \text{(by (19) and (20))} \\ = & \theta''_1\theta'_1(b) & \text{(by (22), (24), (25) and (27))} \\ = & \theta''_1\theta(b) & \text{(by (19), (28))} \\ = & \theta''\theta'(b) & \text{(by (29), (30))} \end{array}$$

By (28), we have $\theta'_1(a) = S$. By (21), we have $\theta_a(a) = S'$. By (25) we therefore have $\theta''_1(S) = \theta_a(a) = S'$. Together with (26), $A_0 = S' \to B'$, and $A_R = S \to B$ we have shown (IV).

Case $\lambda(x : A).M$**:** This case is analogous to the previous one; the only difference is as follows:
By (4) and Lam-Ascribe, we have $A_0 = A \to B'$ for some B' as well as $\Delta, \Theta'; \theta_0\Gamma, (x : A) \vdash M : B'$. However, by (1), we have $\Delta \vdash A$ and therefore $\theta_0(A) = A$.
Hence, we can apply the induction hypothesis directly to the typing judgement above, rather than having to construct θ_a.

Case $M\ N$**:** By (4) and App, we have $\Delta, \Theta'; \theta_0\Gamma \vdash M : A_N \to A_0$ and $\Delta, \Theta'; \theta_0\Gamma \vdash N : A_N$ **(31)** for some type A_N. The former implies $\Delta, \Theta' \vdash A_0$ **(32)**
By induction, infer$(\Delta, \Theta, \Gamma, M)$ succeeds, returning (Θ_1, θ_1, A') and there exists θ''_1 such that the following conditions hold:

$$\Delta \vdash \theta''_1 : \Theta_1 \Rightarrow \Theta' \quad \textbf{(33)}$$

$$\theta_0 = \theta''_1 \circ \theta_1 \quad \textbf{(34)}$$

$$\Delta, \Theta' \vdash \theta''_1(A') = A_N \to A_0 \quad \textbf{(35)}$$

By (35), A' must not have toplevel quantifiers. Let B_N and B_M such that $A' = B_N \to B_M$ **(36)**. This yields $\theta''_1(B_N) = A_N$ **(37)** and $\theta''_1(B_M) = A_0$ **(38)**.
By Theorem 6, we have $\Delta \vdash \theta_1 : \Theta \Rightarrow \Theta_1$ **(39)** and $\Delta, \Theta_1; \theta_1(\Gamma) \vdash M : A'$, which implies $\Delta, \Theta_1 \vdash A'$. By choosing b as fresh, we have $b \# \Delta$, and $b \# \Theta_1$, and $b \# \Theta_2$ and $b \# \Theta'$ **(40)**

[4] Observe that we cannot deduce this from (24) and (25). A counter-example would be the following: $\Theta = (a : \bullet)$, $\Theta'' = (b : \star)$, $\theta' = (c : \bullet)$, $\theta' = [a \mapsto b]$, $\theta'' = [b \mapsto c]$. We have $\vdash (\theta'' \circ \theta') : \Theta \Rightarrow \Theta'$ and $\vdash \theta'' : \Theta'' \Rightarrow \Theta'$, but not $\vdash \theta' : \Theta' \Rightarrow \Theta''$.

By (34), we can rewrite (31) as $\Delta, \Theta'; \theta_1''\theta_1\Gamma \vdash N : A_N$. By induction (using (33)), we then have that $\mathrm{infer}(\Delta, \Theta_1, \theta_1\Gamma, N)$ succeeds, returning (Θ_2, θ_2, A) and there exists θ_2'' such that

$$\Delta \vdash \theta_2'' : \Theta_2 \Rightarrow \Theta' \qquad \textbf{(41)}$$

$$\theta_1'' = \theta_2'' \circ \theta_2 \qquad \textbf{(42)}$$

$$\Delta, \Theta' \vdash \theta_2''(A) = A_N \qquad \textbf{(43)}$$

By Theorem 6, $\Delta \vdash \theta_2 : \Theta_1 \Rightarrow \Theta_2$ **(44)** as well as $\Delta, \Theta_2; \theta_2\theta_1\Gamma \vdash N : A$, which implies $\Delta, \Theta_2 \vdash A$ **(45)**.
Let θ_b be defined such that

$$\theta_b(c) = \begin{cases} \theta_2''(c) & \text{if } c \in \Theta_2 \\ \theta_2''\theta_2(B_M) & \text{if } c = b \end{cases} \qquad \textbf{(46)}$$

We have $\theta_b(b) = \theta_2''\theta_2(B_M) = \theta_1''(B_M) = A_0$**(47)**. By (32) and (41) we thus have $\Delta \vdash \theta_b : (\Theta_2, b : \star) \Rightarrow \Theta'$. Due to (45), we further have $\theta_b(A) = \theta_2''(A)$ **(48)**.
We show applicability of the completeness of unification theorem:

$$\begin{array}{rrcll} & \theta_b\theta_2(A') & & & \\ = & \theta_b\theta_2(B_N) & \to & \theta_b\theta_2(B_M) & \text{(by (36))} \\ = & \theta_2''\theta_2(B_N) & \to & \theta_2''\theta_2(B_M) & \text{(by (40) and (46))} \\ = & \theta_1''(B_N) & \to & \theta_2''\theta_2(B_M) & \text{(by (42))} \\ = & A_N & \to & \theta_2''\theta_2(B_M) & \text{(by (37))} \\ = & \theta_2''(A) & \to & \theta_2''\theta_2(B_M) & \text{(by (43))} \\ = & \theta_b(A) & \to & \theta_b(b) & \text{(by (46) and (48))} \\ = & \theta_b(A & \to & b) & \text{(by (36))} \end{array}$$

By the equality above as well as $\Delta, \Theta_2 \vdash \theta_2(A')$ and $\Delta, \Theta_2, b : \star \vdash (A \to b)$, Theorem 5 states that $\mathrm{unify}(\Delta, (\Theta_2, b : \star), \theta_2(H), A \to b)$ succeeds, returning (Θ_3, θ_3'), and there exists θ_3'' such that $\Delta \vdash \theta_3'' : \Theta_3 \Rightarrow \Theta'$ **(49)** and $\theta_b = \theta_3'' \circ \theta_3'$ **(50)**. The latter implies $\Delta \vdash \theta_3' : (\Theta_2, b : \star) \Rightarrow \Theta_3$. This makes defining $\theta_3' = \theta_3[b \to B]$ **(51)** succeed, resulting in $\Delta \vdash \theta_3 : \Theta_2 \Rightarrow \Theta_3$ **(52)**.
Observe that θ_2'' arises from θ_b in the same way as θ_3 arises from θ_3' by removing b from its domain. Therefore, (50) yields $\theta_2'' = \theta_3'' \circ \theta_3$ **(53)**.
By (39), (44), (52), and composition, we have $\Delta \vdash \theta_3 \circ \theta_2 \circ \theta_1 : \Theta \Rightarrow \Theta_3$.
We have shown that all steps of the algorithm succeed and it returns $(\Theta'', \theta', A_R) = (\Theta_3, \theta_3 \circ \theta_2 \circ \theta_1, B)$ **(54)**.
Let θ'' be defined as θ_3'', satisfying (II), by (49).
We show satisfaction of (III) as follows:

$$\begin{array}{rll} & \theta_0 & \\ = & \theta_1'' \circ \theta_1 & \text{(by (34))} \\ = & (\theta_2'' \circ \theta_2) \circ \theta_1 & \text{(by (42))} \\ = & ((\theta_3'' \circ \theta_3) \circ \theta_2) \circ \theta_1 & \text{(by (53))} \\ = & \theta'' \circ \theta' & \end{array}$$

We show (IV):

$$\begin{array}{rll} & \theta''(A_R) & \\ = & \theta_3''(B) & (\text{by } \theta'' = \theta_3'', A_R = B) \\ = & \theta_3''\theta_3'(b) & \text{(by (51))} \\ = & \theta_b(b) & \text{(by (50))} \\ = & A_0 & \text{(by (47))} \end{array}$$

Case let $x =\ M$ in N: By (4) and Let, there exist A', A_x, and Δ_G such that

$$\Delta_G = \mathsf{ftv}(A') - (\Delta, \Theta') \quad \textbf{(55)}$$
$$\Delta, \Theta', \Delta_G; \theta_0\Gamma \vdash M : A' \quad \textbf{(56)}$$
$$((\Delta, \Theta'), \Delta_G, M, A') \Updownarrow A_x \quad \textbf{(57)}$$
$$\Delta, \Theta'; \theta_0\Gamma, x : A_x \vdash N : A_0 \quad \textbf{(58)}$$
$$\mathsf{principal}((\Delta, \Theta'), \theta_0\Gamma, \Delta_G, A') \quad \textbf{(59)}$$

We assume without loss of generality that Δ_G is fresh, in particular $\Delta_G \# \Theta$. This is justified, as we may otherwise apply Lemma I.4 to (56) using a substitution that does the necessary freshening. This would yield corresponding judgements for deriving $\Delta, \Theta'; \theta_0\Gamma \vdash M_0 : A_0$.
By (3) and weakening, we have $\Delta \vdash \theta_0 : \Theta \Rightarrow \Theta', \Delta_G$. Together with (56) we then have that $\mathsf{infer}(\Delta, \Theta, \Gamma, M)$ succeeds, returning (Θ_1, θ_1, A), and there exists θ_1'' such that

$$\Delta \vdash \theta_1'' : \Theta_1 \Rightarrow (\Theta', \Delta_G) \quad \textbf{(60)}$$
$$\theta_0 = \theta_1'' \circ \theta_1 \quad \textbf{(61)}$$
$$\theta_1''(A) = A' \quad \textbf{(62)}$$

By (1) and (2), Theorem 6 yields $\Delta \vdash \theta_1 : \Theta \Rightarrow \Theta_1$ **(63)** and $\Delta, \Theta_1; \theta_1\Gamma \vdash M : A$, which implies $\Delta, \Theta_1 \vdash A$ **(64)**.
Note that Δ_G does not appear as part of the input to infer, and we therefore have $\Delta_G \# \Theta_1$.
Let $\Theta_{\theta_1} = \mathsf{ftv}(\theta_1) - \Delta$, which implies $\Theta_{\theta_1} \subseteq \Theta_1$ and $\Delta''' \# \Theta_{\theta_1}$ and $\Delta'' \# \Theta_{\theta_1}$. By (3), (60) and (61) we have $\Delta, \Theta' \vdash \theta_1''(a) : K$ for all $(a : K) \in \Theta_{\theta_1}$ **(65)**.
By (3), (55), (59), (60) to (62) and (64), we can apply Lemma I.6, yielding $\theta_1''(\Delta''') = \Delta_G$ **(66)**.
We have $\Delta'' \# \Theta_{\theta_1}$ and can therefore strengthen (63) to $\Delta \vdash \theta_1 : \Theta \Rightarrow \Theta_1 - \Delta''$ **(67)**.
We distinguish two cases based on the shape of M. In each case we show that there exists θ_N'' such that

$$\Delta \vdash \theta_N'' : (\Theta_1' - \Delta'') \Rightarrow \Theta' \quad \textbf{(68)}$$
$$\Delta, \Theta'; \theta_N''(\theta_1(\Gamma), x : \forall\Delta''.A) \vdash N : A_0 \quad \textbf{(69)}$$
$$\theta_0 = \theta_N'' \circ \theta_1 \quad \textbf{(70)}$$

Subcase 1, $M \in \mathsf{GVal}$: We have $\Delta'' = \Delta'''$. By (57), we have that $A_x = \forall\Delta_G.A'$ holds.
According to $\Delta'' = \Delta'''$ and $\Theta_1' = \mathsf{demote}(\bullet, \Theta_1, \Delta''')$ we have that $\Theta_1' - \Delta'' = \Theta_1 - \Delta''$.
Let θ_N'' be defined as follows for all $c \in \Theta_1 - \Delta'' = \Theta_1' - \Delta''$:

$$\theta_N''(c) = \begin{cases} \theta_1''(c) & \text{if } c \in \Theta_{\theta_1} \\ A_D & \text{if } c \in \Theta_1 - \Delta'' - \Theta_{\theta_1} \end{cases}$$

Where A_D is some arbitrary type with $\Delta, \Theta' \vdash A_D : \bullet$ (e.g., Int). By $\Theta_{\theta_1} \subseteq \Theta_1$, this definition is well-formed.
By $\Delta'' = \Delta''' = \mathsf{ftv}(A) - \Delta - \Theta_{\theta_1}$ we have $\theta_N''(c) = \theta''(c)$ for all $c \in \mathsf{ftv}(A) - \Delta''$ **(71)**.
Together with (65) and $\Delta, \Theta' \vdash A_D : \bullet$, we then have $\Delta, \Theta' \vdash \theta_N''(c) : K$ for all $(c : K) \in \Theta_1 - \Delta''$ and therefore $\Delta \vdash \theta_N'' : \Theta_1' - \Delta'' \Rightarrow \Theta'$.
By (61) and (67) and $\theta_N''(c) = \theta_1''(c)$ for all $c \in \Theta_{\theta_1}$ we also have $\theta_0 = \theta_N'' \circ \theta_1$.
We have

$$\begin{array}{lll} = & \theta_N''(\forall\Delta''.A) & \\ = & \theta_N''(\forall\Delta_G.A[\Delta_G/\Delta'']) & \\ = & \forall\Delta_G.\theta_N''(A[\Delta_G/\Delta'']) & (\text{by } \mathsf{ftv}(\theta_N'') \subseteq \Delta, \Theta' \text{ and } \Delta, \Theta' \# \Delta_G \# \Theta_1) \\ = & \forall\Delta_G.\theta_1''(A) & (\text{by } \Delta'' = \Delta''' \text{ and (66) and (71) }) \\ = & A_x & (\text{by } A_x = \forall\Delta_G.A' \text{ and (62)}) \end{array}$$

Thus, (58) is equivalent to $\Delta, \Theta'; \theta_N''((\theta_1\Gamma), x : \forall\Delta''.A) \vdash N : A_0$.
Subcase 2, $M \notin \mathsf{GVal}$: We have $\Delta'' = \cdot$. By (57), we have $A_x = \delta(A')$ for some δ with $\Delta, \Theta' \vdash \delta : \Delta_G \Rightarrow_\bullet \cdot$ **(72)**.
Let θ_N'' be defined as follows for all $c \in \Theta_1 - \Delta'' = \Theta_1$:

$$\theta_N''(c) = \begin{cases} \theta_1''(c) & \text{if } c \in \Theta_{\theta_1} \\ A_D & \text{if } c \in \Theta_1 - \Delta''' - \Theta_{\theta_1} \\ \delta(\theta_1''(c)) & \text{if } c \in \Delta''' \end{cases} \quad \textbf{(73)}$$

Here, A_D is defined as before.
By $\Delta'' \# \Theta_{\theta_1}$ and $\Delta''' \subseteq \Theta_1$ and $\Theta_{\theta_1} \subseteq \Theta_1$, the three cases are non-overlapping and exhaustive for Θ_1.
Using (65), we have that $\Delta, \Theta' \vdash \theta''_N(c) : K$ for all $(c : K) \in \Theta_{\theta_1}$. Note that by $\Delta''' \# \Theta_{\theta_1}$ we have $\Theta_1(c) = \Theta'_1(c)$ for all $c \in \Theta_{\theta_1}$.
By (72), we have $\Delta, \Theta' \vdash \delta(c) : \bullet$ for all $c \in \Delta_G$ and therefore $\Delta, \Theta' \vdash \theta''_N(c') : \bullet$ for all $(c' : K) \in \Delta'''$.
Together with $\Delta, \Theta' \vdash A_D : \bullet$, we then have $\Delta \vdash \theta''_N : \Theta'_1 \Rightarrow \Theta'$. By Lemma G.10, we also have $\Delta \vdash \theta''_N : \Theta_1 \Rightarrow \Theta'$. We have $\theta''_N(c) = \theta''(c)$ for all $c \in \Theta_{\theta_1}$ and together with (63), (61), and $\Delta'' = \cdot$ we then have $\theta_0 = \theta''_N \circ \theta_1$.
We have

$$\begin{array}{lll} & \theta''_N(\forall \Delta''.A) & \\ = & \theta''_N(A) & \text{(by } \Delta'' = \cdot) \\ = & \theta''_1(A)[\delta(\Delta_G)/\Delta_G] & \text{(by (66) and (73))} \\ = & A'[\delta(\Delta_G)/\Delta_G] & \text{(by (62))} \\ = & \delta(A') & \\ = & A_x & \end{array}$$

We have shown that in each case, (68), (69), and (70) hold. Using the same reasoning as in the case for unannotated **let** in the proof of Theorem 6, we obtain $\Delta, \Theta_1 - \Delta'' \vdash \theta_1 : \Gamma$.
Thus, by induction, we have that $\mathsf{infer}(\Delta, \Theta'_1 - \Delta'', \theta_1\Gamma, N)$ succeeds, returning (Θ_2, θ_2, B), and there exists θ''_2 such that

$$\Delta \vdash \theta''_2 : \Theta_2 \Rightarrow \Theta' \quad \textbf{(74)}$$
$$\theta''_N = \theta''_2 \circ \theta_2 \quad \textbf{(75)}$$
$$\theta''_2(B) = A_0 \quad \textbf{(76)}$$

By the return values of infer, we have $\Theta' := \Theta_2$, and $\theta' := \theta_2 \circ \theta_1$ and $A_R := B$.
Let $\theta'' = \theta''_2$. By (74), this choice immediately satisfies (II).
We have

$$\begin{array}{lll} & \theta_0 & \\ = & \theta''_N \circ \theta_1 & \text{(by (70))} \\ = & \theta''_2 \circ \theta_2 \circ \theta_1 & \text{(by (75))} \end{array}$$

and therefore $\theta_0 = \theta'' \circ \theta'$ (III).
We show satisfaction of (IV) as follows:

$$\begin{array}{lll} & A_0 & \\ = & \theta''_2(B) & \text{(by (76))} \\ = & \theta''(B) & \text{(by } \theta'' := \theta''_2) \end{array}$$

Case let $(x : A) =\ M$ **in** N**:** By (4) and LET-ASCRIBE, there exist Δ_G and A_M such that we have

$$\Delta_G, A_M = \mathsf{split}(A, M) \quad \textbf{(77)}$$
$$\Delta, \Theta', \Delta_G; \theta_0\Gamma \vdash M : A_M \quad \textbf{(78)}$$
$$A = \forall \Delta_G.A_M \quad \textbf{(79)}$$
$$\Delta, \Theta'; \theta_0(\Gamma), (x : A) \vdash N : A_0 \quad \textbf{(80)}$$

By alpha-equivalence, we assume $\Delta_G \# \Theta$.
Note that by definition of infer and split, we have $\Delta_G = \Delta'$ and $A' = A_M$ **(81)**. By (78), we have $\Delta' \# \Theta'$ **(82)**. We weaken (3) to $\Delta, \Delta' \vdash \theta_0 : \Theta \Rightarrow \Theta'$.
By inversion on (1), we have $\Delta, \Delta' \Vdash M$ and $\Delta \Vdash N$ and and $\Delta \vdash A$ **(83)**, which implies $\Delta, \Delta' \vdash A'$ **(84)**.
Together with (78) we then have the following by induction: $\mathsf{infer}((\Delta, \Delta'), \Theta, \Gamma, M)$ succeeds, returning $(\Theta_1, \theta_1, A_1)$ and there exists θ''_1 such that

$$\Delta, \Delta' \vdash \theta''_1 : \Theta_1 \Rightarrow \Theta' \quad \textbf{(85)}$$
$$\theta_0 = \theta''_1 \circ \theta_1 \quad \textbf{(86)}$$
$$\theta''_1(A_1) = A_M \quad \textbf{(87)}$$

Theorem 6 yields $\Delta, \Delta' \vdash \theta_1 : \Theta \Rightarrow \Theta_1$ **(88)** and $\Delta, \Delta', \Theta_1; \theta_1(\Gamma) \vdash M : A_1$, which implies $\Delta, \Delta', \Theta_1 \vdash A_1$ **(89)**.

We then have

$$\begin{array}{lll} & \theta_1''(A_1) & \\ = & A_M & \text{(by (87))} \\ = & A' & \text{(by (81))} \\ = & \theta_1''(A') & \text{(by (84) and (85))} \end{array}$$

In addition to above equality and (85) as well as (89), we have $\Delta, \Delta', \Theta_1 \vdash A'$ by weakening (84). Hence, Theorem 5 yields the following: $\mathsf{unify}((\Delta, \Delta'), \Theta_1, A', A_1)$ succeeds, returning (Θ_2, θ_2'), and there exists θ_2'' such that

$$\Delta, \Delta' \vdash \theta_2'' : \Theta_2 \Rightarrow \Theta' \tag{90}$$

$$\theta_1'' = \theta_2'' \circ \theta_2' \tag{91}$$

By Theorem 4, we have $\Delta, \Delta' \vdash \theta_2' : \Theta_1 \Rightarrow \Theta_2$. Together with (88) and composition, we then have $\Delta, \Delta' \vdash \theta_2 : \Theta \Rightarrow \Theta_2$ **(92)**. By (86) and (91), we have $\theta_0 = \theta_2'' \circ \theta_2' \circ \theta_1 = \theta_2'' \circ \theta_2$ **(93)**. We show $\mathsf{ftv}(\theta_2) \subseteq \Delta, \Theta_2$: Otherwise, if $a \in \Theta$ and $b \in \Delta'$ such that $b \in \mathsf{ftv}(\theta_2(a))$, then by (90), $\theta_2''(b) = b$ and $b \in \mathsf{ftv}(\theta_2''(\theta_2(a))) = \mathsf{ftv}(\theta_0(a))$, violating (3).
Therefore, the assertion $\mathsf{ftv}(\theta_2) \# \Delta'$ succeeds, allowing us to strengthen (92) to $\Delta \vdash \theta_2 : \Theta \Rightarrow \Theta_2$ **(94)**.
By (83) we have $\mathsf{ftv}(A) \subseteq \Delta$, and together with (90) this yields $\theta_2''(A) = A$ **(95)**.
We have

$$\begin{array}{lll} & \Delta, \Theta'; \theta_0\Gamma, x : A \vdash N : A_0 & \text{(by (80))} \\ \text{implies} & \Delta, \Theta'; \theta_2''\theta_2\Gamma, x : A \vdash N : A_0 & \text{(by (90), (92) and (93))} \\ \text{implies} & \Delta, \Theta'; \theta_2''(\theta_2(\Gamma), x : A) \vdash N : A_0 & \text{(by (95))} \end{array} \tag{96}$$

By (2) and (94), we have $\Delta, \Theta_2 \vdash \theta_2(\Gamma)$. Together with (83), we then have $\Delta, \Theta_2 \vdash \theta_2(\Gamma), x : A$.
Hence, induction on (94) and (96) shows that $\mathsf{infer}(\Delta, \Theta_2, (\theta_2\Gamma, x : A), N)$ succeeds, returning (Θ_3, θ_3, B) and there exists θ_3'' such that

$$\Delta \vdash \theta_3'' : \Theta_3 \Rightarrow \Theta' \tag{97}$$

$$\theta_2'' = \theta_3'' \circ \theta_3 \tag{98}$$

$$\theta_3''(B) = A_0 \tag{99}$$

We have shown that all steps of the algorithm succeed. According to the return values of infer, we have $\Theta'' = \Theta_3$, $\theta' = \theta_3 \circ \theta_2$, and $A_R = B$. Let $\theta'' = \theta_3''$. By (97), this choice immediately satisfies (II).
We show (III):

$$\begin{array}{lll} & \theta_0 & \\ = & \theta_2'' \circ \theta_2 & \text{(by (93))} \\ = & \theta_3'' \circ \theta_3 \circ \theta_2 & \text{(by (98))} \\ = & \theta'' \circ \theta' & \text{(by } \theta' := \theta_3 \circ \theta_2, \theta'' := \theta_3'' \text{)} \end{array}$$

By $\theta'' = \theta_3''$, (99) yields (IV).

□