# FreezeML

Complete and Easy Type Inference for First-Class Polymorphism

FRANK EMRICH, The University of Edinburgh
SAM LINDLEY, The University of Edinburgh and Imperial College London
JAN STOLAREK, The University of Edinburgh and Lodz University of Technology
JAMES CHENEY, The University of Edinburgh and The Alan Turing Institute
JONATHAN COATES, The University of Edinburgh

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 by augmenting ML with type annotations or other features.

We put forth a new proposal, with different goals to much of the existing literature. Our aim is to make a minimal extension to ML to support first-class polymorphism. We err on the side of explicitness over parsimony, extending 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. The resulting language is not always as concise as more sophisticated systems, but as we illustrate with an array of examples from the literature, in practice it does not appear to require a great deal more ink.

FreezeML is a conservative extension of ML, equipped with type-preserving translations back and forth between System F. It admits a type inference algorithm, an extension of algorithm W, that is sound and complete and which yields principal types.

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

## 1 INTRODUCTION

Functional programming languages, such OCaml and Haskell, employ algorithms based on Hindley-Milner type inference and go to great efforts to reduce the need to write type annotations on programs. Advanced programming techniques often rely on first-class, impredicative polymorphism, where type variables can be replaced by 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. Type inference in the presence of impredicative polymorphism is "a deep, deep swamp" and "no solution to impredicativity with a good benefit-to-weight ratio has been presented to date" [Serrano et al. 2018].

The basic Hindley-Milner algorithm [Damas and Milner 1982] only allows type variables to be instantiated with monomorphic types, and restricts the use of polymorphism in types to *type schemes* of the form $\forall \vec{a}.T$ where $T$ does not contain any further polymorphism. This means that given a function single : $\forall a.a \to \mathsf{List}\ a$, that constructs a list of one element, and a polymorphic identity function id : $\forall a.a \to a$, the expression single id is assigned the type $\forall a.\mathsf{List}\ (a \to a)$, which is a polymorphic list of identity functions. This type arises from instantiating the type quantifier of single with $a \to a$. But what if a programmer wishes to construct a list of polymorphic identity functions, i.e. an expression of type List $(\forall a.a \to a)$? This requires instantiating the quantifier of single with a polymorphic type $\forall a.a \to a$. In ML, it is forbidden to instantiate type variables with polymorphic types, thus it is impossible to construct a term of type List $(\forall a.a \to a)$. (Indeed, this System F type is not even a ML type scheme.) However, in a richer language such as System

F, the expression single id could be annotated as appropriate in order to obtain either the type $\forall a.\mathsf{List}\ (a \to a)$ or the type $\mathsf{List}\ (\forall a.a \to a)$.

Alas, type inference, and indeed type checking, is undecidable for System F [Wells 1994] without type annotations. Moreover, even in System F with type annotations but no explicit instantiation, type inference remains undecidable [Pfenning 1993]. There has been a plethora of research to bridge the gap in expressiveness between ML and System F without sacrificing desirable type inference properties of ML [Le Botlan and Rémy 2003; Leijen 2007, 2008, 2009; Russo and Vytiniotis 2009; Serrano et al. 2018; Vytiniotis et al. 2006, 2008]. However, none of the systems designed so far seems to have achieved widespread use as part of a production compiler. The proposed systems are either too difficult for the programmer to reason about, too verbose, too complicated to combine with other features of real-life programming languages, and often a combination of the above.

There are two ways of approaching the problem of maintaining tractable type inference in the presence of polymorphic instantiation. The first approach is to require that the programmer disambiguates polymorphic instantiations by adding type annotations. A problem here is that the verbosity of annotations can make the code hard to read and write. Moreover, such systems can often behave in an unintuitive way. For example, the programmer might write $\mathsf{single}\ \mathsf{id} : \mathsf{List}\ (\forall a.a \to a)$ and expect that the annotation on the whole expression will propagate to subexpressions, leading to a correct instantiation. In some systems, e.g. HMF [Leijen 2008], this is not the case and a more verbose annotation $(\forall a.a \to a) \to \mathsf{List}\ (\forall a.a \to a)$ has to be placed on single. The second approach is to try to guess polymorphic instantiations in a reasonable way (for some definition of "reasonable"). A difficulty here is that it is not always obvious from the local context what the right type is. Many existing systems lie between the two extremes, i.e. they try to infer polymorphism up to a certain point, and when that does not work fall back to user-provided annotations. A pitfall of such approaches is that they can be unpredictable and thus hard to reason about.

We propose FreezeML, a language in which the programmer can be explicit about polymorphic instantiation — in particular by opting out of instantiation when desired. The resulting system is not as concise as some existing alternative designs but reasoning about FreezeML programs is relatively easy thanks to its simple treatment of instantiation. FreezeML does not introduce new types, but rather shares its types with System F. For these reasons we have found FreezeML to be straightforward to implement; we have done so on top of the Links [Cooper et al. 2006] programming language (the FreezeML features and test cases for all of the examples in the paper are included in the master branch of Links available here: https://github.com/links-lang/links).

FreezeML is a conservative extension of ML with the expressive power of System F. The paper makes the following main contributions.

- A high-level introduction to FreezeML through a comprehensive collections of examples taken from the literature (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 [Damas and Milner 1982], which is sound, complete, and yields principal types (Section 5).

Section 6 discusses related work and Section 7 concludes.

$$
\begin{array}{rlrl}
\mathsf{head} & : \forall a.\mathsf{List}\ a \to a & \mathsf{id} & : \forall a.a \to a \\
\mathsf{tail} & : \forall a.\mathsf{List}\ a \to \mathsf{List}\ a & \mathsf{ids} & : [\forall a.a \to a] \\
[\,] & : \forall a.\mathsf{List}\ a & \mathsf{inc} & : \mathsf{Int} \to \mathsf{Int} \\
(::) & : \forall a.a \to \mathsf{List}\ a \to \mathsf{List}\ a & \mathsf{choose} & : \forall a.a \to a \to a \\
\mathsf{single} & : \forall a.a \to \mathsf{List}\ a & \mathsf{poly} & : (\forall a.a \to a) \to \mathsf{Int} \times \mathsf{Bool} \\
(@) & : \forall a.\mathsf{List}\ a \to \mathsf{List}\ a \to \mathsf{List}\ a & \mathsf{auto} & : (\forall a.a \to a) \to (\forall a.a \to a) \\
\mathsf{length} & : \forall a.\mathsf{List}\ a \to \mathsf{Int} & \mathsf{auto}' & : \forall b.(\forall a.a \to a) \to (b \to b) \\
\mathsf{runST} & : \forall a.(\forall s.ST\ s\ a) \to a & \mathsf{map} & : \forall a\ b.(a \to b) \to \mathsf{List}\ a \to \mathsf{List}\ b \\
\mathsf{argST} & : \forall s.ST\ s\ \mathsf{Int} & \mathsf{app} & : \forall a\ b.(a \to b) \to a \to b \\
\mathsf{pair} & : \forall a\ b.a \to b \to a \times b & \mathsf{revapp} & : \forall a\ b.a \to (a \to b) \to b \\
\mathsf{pair}' & : \forall b\ a.a \to b \to a \times b & &
\end{array}
$$

Fig. 1. Type signatures for functions used in the text; adapted from [Serrano et al. 2018].

## 2 FREEZEML BY EXAMPLE

We begin with a hands-on presentation of FreezeML via a collection of examples. We use functions with type signatures shown in Figure 1 (adapted from Serrano et al. [2018]). Note that the types of FreezeML are exactly those of System F.

### 2.1 Essential Features

*Implicit Instantiation.* In FreezeML (as in standard Hindley-Milner type inference), when variable occurrences are typechecked, the outer universally quantified type variables in the variable's type are instantiated implicitly. Suppose a programmer writes choose id. The identity function id has type $\forall a.a \to a$, but since it appears as a variable applied to a function, 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.* The programmer may explicitly prevent a variable from having its quantifiers instantiated by using the freeze operator $\lceil - \rceil$. Whereas the term id has type $a \to a$ for an unknown type $a$, the term $\lceil id \rceil$ has type $\forall a.a \to a$. More interestingly, whereas the term single id has type $\forall a.\mathsf{List}\ (a \to a)$, the term single $\lceil \mathsf{id} \rceil$ has type $\mathsf{List}\ (\forall a.a \to a)$. It is important that quantifiers are present when required. Consider the term auto id. Here id is implicitly instantiated and has 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 $\lceil \mathsf{id} \rceil$.

*Explicit Generalisation.* The freeze operator supports named polymorphic arguments, for instance: **let** id $= \lambda x.x$ **in** poly $\lceil id \rceil$. The explicit generalisation operator \$ supports anonymous polymorphic arguments. For instance, whereas the term $\lambda x.x$ has type $a \to a$, the term $\$(\lambda x.x)$ has type $\forall a \to a$, allowing us to write poly $\$(\lambda x.x)$. Explicit generalisation is macro-expressible [Felleisen 1991] 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 [Wright 1995]; hence generalisation only applies to syntactic values.

*Explicit Instantiation.* As in ML variables in FreezeML are implicitly instantiated, but general terms are not. Nevertheless, we can instantiate a term by binding it to a variable: **let** $x$ = head ids **in** $x$ 42. The explicit instantiation operator @ supports instantation of a term without binding it to a variable. For instance, whereas the term head ids has type $\forall a.a \to a$ the term (head ids)@ has type Int → Int, allowing us to write (head ids)@ 42. 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 a function f : $(\forall a\, b.a \to b \to a \times b) \to Int$ then $f\ \lceil pair \rceil$, $f$ \$*pair*, and $f$ \$*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 $f\ \lceil pair' \rceil$ is ill-typed as the order of quantifiers on $\lceil pair' \rceil$ does not match the requirements of the f function.

Other than type annotations on binders, the only extension to the syntax of ML in FreezeML is the freeze operator. Having annotated binders allows us to define functions with polymorphic arguments, and also handle polymorphic let-binders, including polymorphic partial applications.

### 2.2 Concrete Examples

Table 1. Example FreezeML Terms and Types

| 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′ | : | ✕ |
| A9★ | $f$ (choose $\lceil$id$\rceil$) ids | : | $\forall a.a \to a$ |
| | | where | $f : \forall a.(a \to a) \to \text{List}\ a \to a$ |
| A10★ | poly $\lceil$id$\rceil$ | : | Int × Bool |
| A11★ | poly \$($\lambda x.x$) | : | Int × Bool |
| A12★ | id poly \$($\lambda x.x$) | : | Int × Bool |
| B | INFERENCE WITH POLYMORPHIC ARGUMENTS | | |

| | | | |
|---|---|---|---|
| B1★ | $\lambda(f : \forall a.a \to a).(f\ 1, f\ \mathsf{True})$ | : | $(\forall a.a \to a) \to \mathsf{Int} \times \mathsf{Bool}$ |
| B2★ | $\lambda(xs : \mathsf{List}\ (\forall a.a \to a)).\mathsf{poly}\ (\mathsf{head}\ xs)$ | : | $\mathsf{List}\ (\forall a.a \to a) \to \mathsf{Int} \times \mathsf{Bool}$ |
| C | FUNCTIONS ON POLYMORPHIC LISTS | | |
| C1 | $\mathsf{length}\ \mathsf{ids}$ | : | $\mathsf{Int}$ |
| C2 | $\mathsf{tail}\ \mathsf{ids}$ | : | $\mathsf{List}\ (\forall a.a \to a)$ |
| C3 | $\mathsf{head}\ \mathsf{ids}$ | : | $\forall a.a \to a$ |
| C4 | $\mathsf{single}\ \mathsf{id}$ | : | $\mathsf{List}\ (a \to a)$ |
| C4• | $\mathsf{single}\ \lceil\mathsf{id}\rceil$ | : | $\mathsf{List}\ (\forall a.a \to a)$ |
| C5★ | $\lceil\mathsf{id}\rceil :: \mathsf{ids}$ | : | $\mathsf{List}\ (\forall a.a \to a)$ |
| C6★ | $\$(\lambda x.x) :: \mathsf{ids}$ | : | $\mathsf{List}\ (\forall a.a \to a)$ |
| C7 | $(\mathsf{single}\ \mathsf{inc}) +\!\!+ (\mathsf{single}\ \mathsf{id})$ | : | $\mathsf{List}\ (\mathsf{Int} \to \mathsf{Int})$ |
| C8★ | $\mathsf{g}\ (\mathsf{single}\ \lceil\mathsf{id}\rceil)\ \mathsf{ids}$ | : | $\forall a.a \to a$ |
| | | | where $\mathsf{g} : \forall a.\mathsf{List}\ a \to \mathsf{List}\ a \to a$ |
| C9★ | $\mathsf{map}\ \mathsf{poly}\ (\mathsf{single}\ \lceil\mathsf{id}\rceil)$ | : | $\mathsf{List}\ (\mathsf{Int} \times \mathsf{Bool})$ |
| C10 | $\mathsf{map}\ \mathsf{head}\ (\mathsf{single}\ \mathsf{ids})$ | : | $\mathsf{List}\ (\forall a.a \to a)$ |
| D | APPLICATION FUNCTIONS | | |
| D1★ | $\mathsf{app}\ \mathsf{poly}\ \lceil\mathsf{id}\rceil$ | : | $\mathsf{Int} \times \mathsf{Bool}$ |
| D2★ | $\mathsf{revapp}\ \lceil\mathsf{id}\rceil\ \mathsf{poly}$ | : | $\mathsf{Int} \times \mathsf{Bool}$ |
| D3★ | $\mathsf{runST}\ \lceil\mathsf{argST}\rceil$ | : | $\mathsf{Int}$ |
| D4★ | $\mathsf{app}\ \mathsf{runST}\ \lceil\mathsf{argST}\rceil$ | : | $\mathsf{Int}$ |
| D5★ | $\mathsf{revapp}\ \lceil\mathsf{argST}\rceil\ \mathsf{runST}$ | : | $\mathsf{Int}$ |
| E | $\eta$-EXPANSION | | |
| E1 | $k\ h\ l$ | : | ✕ |
| E2★ | $k\ \$(\lambda x.(h\ x)@)\ l$ | : | $\forall a.\mathsf{Int} \to a \to a$ |
| | | | where $k : \forall a.a \to \mathsf{List}\ a \to a$ |
| | | | $h : \mathsf{Int} \to \forall a.a \to a$ |
| | | | $l : \mathsf{List}\ (\forall a.\mathsf{Int} \to a \to a)$ |
| E3 | $r\ (\lambda x\ y.y)$ | : | ✕ |
| E3• | $r\ \$(\lambda x.\$(\lambda y.y))$ | : | $\mathsf{Int}$ |
| | | | where $r : (\forall a.a \to \forall b.b \to b) \to \mathsf{Int}$ |
| F | FreezeML PROGRAMS | | |
| F1 | $\mathsf{id}\ x = x$ | : | $\forall a.a \to a$ |
| F2 | $\mathsf{ids} = [\lceil\mathsf{id}\rceil]$ | : | $\mathsf{List}\ (\forall a.a \to a)$ |
| F3 | $\mathsf{auto}\ x = x\ \lceil x\rceil$ | : | $(\forall a.a \to a) \to (\forall a.a \to a)$ |
| F4 | $\mathsf{auto}'\ x = x\ x$ | : | $\forall b.(\forall a.a \to a) \to b \to b$ |
| F5★ | $\mathsf{auto}\ \lceil\mathsf{id}\rceil$ | : | $\forall a.a \to a$ |
| F6 | $(\mathsf{head}\ \mathsf{ids}) :: \mathsf{ids}$ | : | $\mathsf{List}\ (\forall a.a \to a)$ |
| F7★ | $(\mathsf{head}\ \mathsf{ids})@\ 3$ | : | $\mathsf{Int}$ |
| F8 | $\mathsf{choose}\ (\mathsf{head}\ \mathsf{ids})$ | : | $(\forall a.a \to a) \to (\forall a.a \to a)$ |
| F8• | $\mathsf{choose}\ (\mathsf{head}\ \mathsf{ids})@$ | : | $\forall a.(a \to a) \to (a \to a)$ |
| F9 | $\mathbf{let}\ f = \mathsf{revapp}\ \lceil\mathsf{id}\rceil\ \mathbf{in}\ f\ \mathsf{poly}$ | : | $\mathsf{Int} \times \mathsf{Bool}$ |
| F10 | $\mathsf{choose}\ \mathsf{id}\ (\lambda(x : \forall a.a \to a).\$(\mathsf{auto}'\ x))$ | : | $(\forall a.a \to a) \to (\forall a.a \to a)$ |

Table 1 presents a collection of FreezeML examples. Well-typed expressions are annotated with a type inferred in FreezeML, whilst ill-typed expressions are annotated with ✕. Sections A-E of the table are taken from [Serrano et al. 2018], which also contains a comparison of how five different systems (GI [Serrano et al. 2018], MLF [Le Botlan and Rémy 2003], HMF [Leijen 2008],

FPH [Vytiniotis et al. 2008], and HML [Leijen 2009]) behave for these examples. Section F of the table contains additional examples which highlight the behaviour of our system, as discussed above. 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 Table 1, 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⋆.

How does FreezeML compare with the five systems discussed in [Serrano et al. 2018]? 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 Table 1, MLF typechecks all but B1 and E1, placing it first in terms of expressiveness. FreezeML and HML rank second. FreezeML handles all examples except for A8, B1, B2, and E1; HML can't typecheck B1, B2, E1, and E3. 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.

In FreezeML, type annotations are only ever required on binders and never on terms. For instance, in FreezeML Example A9⋆ in FreezeML requires us only to freeze id, where in GI it is necessary to add an annotation (id : $(\forall a.a \to a) \to (\forall a.a \to a)$). Examples C8 and C9 are similar. Of course, annotations are still required on polymorphic lambda-binders, as in B1⋆, which applies its argument at two different types (and acts as a possible definition for poly).

As well as annotations on lambda-binders FreezeML also supports annotations on let-binders. They are less critical than annotations on lambda-binders, but they do allow fine control over the structure of polymorphism, e.g. by allowing the order of quantifiers to be specified and by supporting polymorphic types in which some quantifiers are unused — features that are helpful for embedding System F into FreezeML, for instance (Section 4.1). Moreover, as in typical ML implementations they provide a means for explicitly ascribing non-principal types to terms.

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. This is the restriction we assumed when preparing the table. We can however $\eta$-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 [Vytiniotis et al. 2006].

### 2.3 Design Considerations

*Monomorphism Restriction*[1]. Unlike in ML we can write lambda abstractions that use their arguments polymorphically.

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

To avoid the "swamp" [Serrano et al. 2018] of undecidability and to keep type inference complete, we insist that unannotated $\lambda$-bound variables be monomorphic. If we were to remove the annotation from $f$ then in order to infer a type for $f$ we would have to rely on global reasoning, i.e. inspect all uses of $f$ together. This is not compositional. 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.

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

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 $\lambda$-bound variables be monomorphic.

Another more subtle instance of this same problem can occur due to the ML value restriction [Wright 1995] — which we choose to embrace. Consider the following two functions.

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

The value restriction disallows generalisation of non-values in let-bindings. Thus in both of these examples, because *bot bot* is a non-value, $f$ is initially assigned the type $a$ rather than the most general type $\forall a.a$. Assuming type inference proceeds from left to right then type inference will succeed on bad3 and fail on bad4 for the same reasons as above. In order to rule out this class of examples, we insist that non-values are first generalised and then instantiated with monomorphic types. So rather than assigning a completely unconstrained type $a$ for $f$, we constrain $a$ to only unify with monomorphic types. Our type system for FreezeML (Section 3.4) and inference algorithm (Section 5.4) enforce this constraint using a kind system that explicitly distinguishes monomorphic and polymorphic types. Hence type inference fails 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.* 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

[1]Not to be confused with Haskell's monomorphism restriction [Marlow 2010, Section 4.5.5].

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 follow the approach of Leijen [2008] and bake principality into the typing rule for let, which 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.

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

*Instantiation strategies.* In FreezeML (and indeed ML) the only terms that are implicitly instantiated are variables. We call the FreezeML instantiation strategy *restricted instantiation*. Thus (head ids) 42 is ill-typed and we must explicitly insert the instantiation operator @ to yield a type-correct expression: (head ids)@ 42. As mentioned above, one can easily refine our approach to implicitly instantiate any term appearing in a monomorphic elimination position (in particular application position), and thus, for instance, infer a type for bad5 without compromising completeness. We refer to this strategy as *elimination instantiation*.

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 elimination instantiation, pervasive instantiation infers a type for (head ids) 42. However, pervasive instantiation requires inserting explicit generalisation where it was previously unnecessary, for instance Example C3 would have to be rewritten as \$(head ids). Moreover, pervasive instantiation complicates the meta-theory, requiring two mutually recursive typing judgements instead of just one.

For the remainder of the paper, we confine ourselves to restricted instantiation and defer further investigation of alternative strategies to future work.

## 3 FREEZEML VIA SYSTEM F AND ML

In this section we give syntax-directed presentations of three calculi: a call-by-value variant of System F, mini-ML [Clément et al. 1986], and FreezeML.

*Notations.* We write $\mathsf{ftv}(A)$ for the sequence of distinct free type variables of a type in the order in which they appear in $A$. For example, $\mathsf{ftv}((a \to b) \to (a \to c)) = a, b, c$. We write $\mathsf{ftv}(\theta)$ for the sequence of distinct free type variables of a type substitution in the order in which they appear. For example, $\mathsf{ftv}([a \mapsto (a \to b)], [d \mapsto (a \to c)]) = a, b, c$. We write $\Delta - \Delta'$ for the restriction of $\Delta$ to those type variables that do not appear in $\Delta'$. We write $\Delta \# \Delta'$ to mean that the type variables in $\Delta$ and $\Delta'$ are disjoint.

### 3.1 System F

We wish to compile a call-by-value ML-like language into a core language with explicit first-class polymorphism, thus 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 2.

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 [Wadler 1990], but for the purposes of our examples we treat them specially. Types comprise type variables ($a$), fully-applied type constructors

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

Fig. 2. System F Syntax

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

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

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

$$\text{F-Var}\ \frac{x : A \in \Gamma}{\Delta; \Gamma \vdash x : A} \qquad \text{F-Lam}\ \frac{\Delta; \Gamma, x : A \vdash M : B}{\Delta; \Gamma \vdash \lambda x^A.M : A \to B} \qquad \text{F-App}\ \frac{\Delta; \Gamma \vdash M : A \to B \qquad \Delta; \Gamma \vdash N : A}{\Delta; \Gamma \vdash M\,N : B}$$

$$\text{F-PolyLam}\ \frac{\Delta, a; \Gamma \vdash V : A}{\Delta; \Gamma \vdash \Lambda a.V : \forall a.A} \qquad \text{F-PolyApp}\ \frac{\Delta; \Gamma \vdash M : \forall a.B \qquad \Delta \vdash A : \star}{\Delta; \Gamma \vdash M\,A : B[A/a]}$$

Fig. 3. System F Kinding and Typing Rules

$(D\,\overline{A})$, and polymorphic types $(\forall a.A)$. Type environments track the types of term variables in a term. Kind environments track the type 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 $\mathbf{let}\ x^A = M\ \mathbf{in}\ N$ 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 [Wright 1995]. This is inessential but we want to design a system that can be readily integrated into a full-fledged programming language with features such as references, which may be accommodated through the value restriction.

Well-formedness of types and the typing rules for System F are given in Figure 3. Standard equational rules $(\beta)$ and $(\eta)$ for System F are given in Figure 4. We make use of these to bootstrap reasoning principles for FreezeML in Section 4.3.

$\beta$-rules

$$(\lambda x^A.V)\, W \simeq V[W/x]$$
$$(\Lambda a.V)\, A \simeq V[A/a]$$

$\eta$-rules

$$\lambda x^A.M\, x \simeq M$$
$$\Lambda a.V\, a \simeq V$$

Fig. 4. System F Equational Rules

| | |
|---|---|
| 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 | $V, W ::= x \mid \lambda x.M$ |
| Kinds | $K ::= \bullet \mid \star$ |
| Kind Environments | $\Delta ::= \cdot \mid \Delta, a$ |
| Type Environments | $\Gamma ::= \cdot \mid \Gamma, x : P$ |

Fig. 5. ML Syntax

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

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

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

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

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

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

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

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

Fig. 6. ML Kinding and Typing Rules

### 3.2 ML

$$C\left[\!\!\left[\frac{x : \forall\Delta'.S \in \Gamma \quad \Delta \vdash \delta : \Delta' \Rightarrow \cdot}{\Delta;\Gamma \vdash x : \delta(S)}\right]\!\!\right] = x\ \delta(\Delta') \qquad C\left[\!\!\left[\frac{\Delta' = \mathsf{gen}(\Delta, S, M) \quad \Delta, \Delta';\Gamma \vdash M : S \quad P = \forall\Delta'.S \quad \Delta;\Gamma, x : P \vdash N : T}{\Delta;\Gamma \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : T}\right]\!\!\right] = \mathbf{let}\ x^{\forall\Delta'.S} = \Lambda\Delta'.C[\![M]\!]\ \mathbf{in}\ C[\![N]\!]$$

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

Fig. 7. Translation from ML to System F

We now outline a core fragment of ML. The syntax is given in Figure 5, well-formedness of types and the typing rules in Figure 6. 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-LetVal), 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 6. The judgement $\Delta \vdash \delta : \Delta' \Rightarrow \Delta''$ defines a finite map from type variables in $\Delta, \Delta'$ into type variables in $\Delta, \Delta''$, such that $\delta(a) = a$ for every $a \in \Delta$. As such, it is only well-defined if $\Delta$ and $\Delta'$ are disjoint and $\Delta$ and $\Delta''$ are disjoint. We may apply type instantiations to types and type schemes in the standard way:

$$\emptyset(S) = S \qquad \delta[a \mapsto S](a) = S \qquad \delta[a \mapsto S](b) = \delta(b) \qquad \delta(D\ \overline{S}) = D\ (\overline{\delta(S)})$$

Generalisation is defined at the bottom of Figure 6. 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 $\Delta$, 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.

### 3.3 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 7. 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

| | |
|---|---|
| Type Variables | $\mathsf{TVar} \ni a, b, c$ |
| Type Constructors | $\mathsf{Con} \ni D ::= \mathsf{Int} \mid \mathsf{Bool} \mid \mathsf{List} \mid \rightarrow \mid \times \mid \ldots$ |
| 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 | $\mathsf{Term} \ni M, N ::= \lceil x \rceil \mid x \mid \lambda x.M \mid \lambda(x : A).M \mid M\,N$ $\mid \ \mathbf{let}\ x = M\ \mathbf{in}\ N \mid \mathbf{let}\ (x : A) = M\ \mathbf{in}\ N$ |
| Values | $\mathsf{Val} \ni V, W ::= \lceil x \rceil \mid x \mid \lambda x.M \mid \lambda(x : A).M$ |
| Guarded Values | $\mathsf{GVal} \ni U ::= x \mid \lambda x.M \mid \lambda(x : A).M$ |
| Kinds | $\mathsf{Kind} \ni K ::= \bullet \mid \star$ |
| Kind Environments | $\mathsf{PEnv} \ni \Delta ::= \cdot \mid \Delta, a$ |
| Type Environments | $\mathsf{TEnv} \ni \Gamma ::= \cdot \mid \Gamma, x : A$ |

Fig. 8. FreezeML Syntax

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 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 $M$, $\Delta'$ is empty.

Theorem 1. *If* $\Delta; \Gamma \vdash M : S$ *then* $\Delta; \Gamma \vdash C[\![M]\!] : S$.

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.

### 3.4 FreezeML

We now extend ML to FreezeML, taking a rather different tack to most of the existing literature on extending ML with first-class polymorphism. Our aim is to make a minimal extension to ML to support first-class polymorphism. We err on the side of being explicit rather than maximising parsimony.

FreezeML is ML extended with two new features. First and foremost, let-bindings and lambda-bindings may be annotated with arbitrary System F types. Second, FreezeML adds a new form $\lceil x \rceil$, called *freezing*, for ensuring that variables are not instantiated.

The syntax of FreezeML is given in Figure 8. (We choose to explicitly name the syntactic categories for later use in Section 5.) The types are the same as in System F. We explicitly distinguish two subsets of types: monotypes ($S, T$), as in ML, and guarded types ($H$), which are types that do not have a top-level quantifier — so any polymorphism in them is guarded by a type constructor. The terms are the same as in ML extended with frozen variables ($\lceil x \rceil$) and lambda- and let-bindings with type ascriptions.

The FreezeML kinding judgement $\Delta \vdash A : K$ states that type $A$ has kind $K$ in kind environment $\Delta$. The kinding rules are given in at the top of Figure 9. As in ML we distinguish monomorphic types ($\bullet$) from polymorphic types ($\star$). Unlike in ML polymorphic types can appear inside data type constructors.

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

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

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

$$\textsc{Freeze}\ \frac{x : A \in \Gamma}{\Delta; \Gamma \vdash \lceil x \rceil : A} \qquad \textsc{Var}\ \frac{x : \forall \Delta'.H \in \Gamma \qquad \Delta \vdash \delta : \Delta' \Rightarrow_\star \cdot}{\Delta; \Gamma \vdash x : \delta(H)}$$

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

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

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

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

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

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

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

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

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

Fig. 9. FreezeML Kinding and Typing Rules

The FreezeML typing judgement $\Delta; \Gamma \vdash M : A$ states that term $M$ has type $A$ in kind environment $\Delta$ and type environment $\Gamma$ (Figure 9). Typing rules are adjusted accordingly with respect to ML to allow full System F types everywhere except in the types of lambda-bound variables and ungeneralised let-bound variables which must be monomorphic unless annotated with a type ascription. FE: The sentence above sounds wrong. In an expression **let** $x = M$ **in** $N$, the type of $M$ is either an arbitrary type if $M$ is not a guarded value or a guarded type (rather than a monomorphic one) if $M$ is a guarded value.

$$\emptyset(A) = A \qquad \delta(D\ \overline{A}) = D\ (\overline{\delta(A)})$$
$$\delta[a \mapsto A](a) = A \qquad \delta(\forall a.A) = \forall a.\delta'(A), \text{ where } \delta'(b) = \begin{cases} b, & \text{if } b = a \\ \delta(b), & \text{otherwise} \end{cases}$$
$$\delta[a \mapsto A](b) = \delta(b)$$

Fig. 10. Application of a Type Instantiation in FreezeML

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, because type variables may appear in monomorphic types, an unannotated lambda abstraction may subsequently be generalised and then be instantiated at different types. The Lam-Ascribe rule allows an argument to be used polymorphically inside the body of a lambda abstraction. The App rule is standard.

Following Leijen [2008] the Let rule asserts that $M$ must type check with a principal type. Without the principal type constraint we would lose completeness of our type inference algorithm. For instance, the program bad5 = **let** $f = \lambda x.x$ **in** $\lceil f \rceil$ 42 from Section 2.3 would be well-typed if we could assign $f$ the type Int $\to$ Int but not if we are required to assign it the principal type $\forall a.a \to a$. The principal relation is discussed further in Section 5. The auxiliary judgement $(\Delta, \Delta'', M, A') \Updownarrow A$ determines the type $A$ of $x$ in $N$ depending on whether $M$ is a guarded value or not. If $M$ is guarded, then $x$ is assigned the fully quantified principal type. If $M$ is not guarded then the principal type is instantiated with monomorphic types substituted for the free type variables. Together with the monomorphism restriction on unannotated function arguments, this restriction to monomorphic types ensures that we never need to guess any polymorphism.

The Let-Ascribe rule is similar to the Let rule, but instead of generalising the type of $M$, it uses the supplied type annotation and admits non-principal types. The split operator splits up the quantifiers and body of the type type annotation if $M$ is a guarded value. Rather like GHC's scoped type variables [Peyton Jones and Shields 2002], these variables may then be used in $M$.

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 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 need to be careful not to substitute for type variables shadowed by nested polymorphic quantifiers (Figure 10).

Generalisation in FreezeML returns two values. The first one is irrelevant for the purpose of typing. Notice in the Let rule how the $\Delta'$ returned by gen is ignored and the choice of whether to generalise the type of $M$ is relegated to the $\Updownarrow$ judgement. However, the first argument returned by gen is used in the translation from FreezeML to System F (Figure 13 in Section 4.2), where it is used to form a type abstraction, and in the type inference algorithm (Figure 17 in Section 5.4), where it allows us to avoid having an extra case to consider.

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

Theorem 2. *If* $\Delta; \Gamma \vdash M : S$ *in ML then* $\Delta; \Gamma \vdash M : S$ *in FreezeML.*

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

Since FreezeML terms contain type annotations, it turns out to be important to define a preliminary well-formedness judgement $\Delta \Vdash M$ which we call *well-scopedness*, defined in Figure 11.

$$\frac{}{\Delta \Vdash \lceil x \rceil} \qquad \frac{}{\Delta \Vdash x} \qquad \frac{\Delta \Vdash M}{\Delta \Vdash \lambda x.M} \qquad \frac{\Delta \vdash A : \star \qquad \Delta \Vdash M}{\Delta \Vdash \lambda(x : A).M} \qquad \frac{\Delta \Vdash M \qquad \Delta \Vdash N}{\Delta \Vdash M\,N}$$

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

Fig. 11. Well-Scopedness of FreezeML Terms

$$\begin{aligned}
\mathcal{E}[\![x]\!] &= \lceil x \rceil \\
\mathcal{E}[\![\lambda x^A.M]\!] &= \lambda(x : A).\mathcal{E}[\![M]\!] \\
\mathcal{E}[\![M\ N]\!] &= \mathcal{E}[\![M]\!]\ \mathcal{E}[\![N]\!] \\
\mathcal{E}[\![\Lambda\Delta.(x\ \overline{A})^B]\!] &= \mathbf{let}\ (y : \forall\Delta.B) = x\ \mathbf{in}\ \lceil y \rceil, && \text{if } \Delta \text{ is non-empty} \\
\mathcal{E}[\![\Lambda\Delta.(\lambda x^A.M)^B]\!] &= \mathbf{let}\ (y : \forall\Delta.B) = \lambda(x : A).\mathcal{E}[\![M]\!]\ \mathbf{in}\ \lceil y \rceil, && \text{if } \Delta \text{ is non-empty} \\
\mathcal{E}[\![M^{\forall a.B}\ A]\!] &= \mathbf{let}\ (x : \forall a.B) = \mathcal{E}[\![M]\!]\ \mathbf{in} \\
&\quad\ \mathbf{let}\ (y : B[A/a]) = x\ \mathbf{in}\ \lceil y \rceil
\end{aligned}$$

Fig. 12. Translation From System F to FreezeML

Well-scopedness is checkable by a straightforward syntax-directed algorithm, and this check is prerequisite to type inference. Well-scopedness checks that in $M$, the type annotations are well-formed with respect to kind environment $\Delta$. The main subtlety in this judgement is in how $\Delta$ grows when we encounter *annotated* let-bindings. For annotated lambdas, we just check that the type annotation is well-formed in $\Delta$ but do not add any type variables in $\Delta$. 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* $\Delta$ *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: $\mathbf{let}\ (f : \forall a.a) = \lambda(x : a).x\ \mathbf{in}\ldots$, 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 $\alpha$-vary freely; that is, the previous expression is $\alpha$-equivalent to $\mathbf{let}\ (f : \forall b.b \to b) = \lambda(x : b).x\ \mathbf{in}\ N$ but not to $\mathbf{let}\ (f : \forall b.b \to b) = \lambda(x : a).x\ \mathbf{in}\ N$. This is similar to other proposals for scoped type variables [Peyton Jones and Shields 2002].

## 4 RELATING SYSTEM F AND FREEZEML

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

### 4.1 From System F to FreezeML

Figure 12 defines an embedding $\mathcal{E}[\![-]\!]$ of System F terms into FreezeML. Variables are frozen in order to suppress instantiation. Term abstractions and applications are translated homomorphically. Because the body of a type abstraction is a value, every type abstraction must have the form

$\Lambda\Delta.x\,\overline{A}$ or $\Lambda\Delta.\lambda x^A.M$ where $\Delta$ is non-empty. We exploit this observation in order to translate type abstractions into annotated let-bindings, which may implicitly perform type abstraction provided the bound term is a guarded value (as per the definition of split). If we were to attempt to define the translation a single quantifier at a time, then we would run into problems because if the term being let-bound was itself a let, then no implicit type abstraction could take place. The result is frozen in order to preserve its potentially polymorphic type. A type application $M^{\forall a.B}\,A$ is translated into two annotated let-bindings. The binding to $x : \forall a.B$ is necessary to allow instantiation to take place in the case that $M$ is a frozen variable. The binding $y : B[A/a]$ performs instantiation as dictated by the type application. As with type abstractions, the result is frozen in order to preserve its potentially polymorphic type. Each translated term has the same type as the original:

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

### 4.2 From FreezeML to System F

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

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

$$C\left[\!\left[\begin{array}{c}(\Delta',\Delta'') = \mathsf{gen}(\Delta, A', M)\\ \Delta,\Delta'';\Gamma \vdash M : A'\\ (\Delta,\Delta'', M, A') \Updownarrow A\\ \Delta;\Gamma, x : A \vdash N : B\\ \hline \Delta;\Gamma \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : B\end{array}\right]\!\right] = \mathbf{let}\ x^A = \Lambda\Delta'.C[\![M]\!]\ \mathbf{in}\ C[\![N]\!] = C\left[\!\left[\begin{array}{c}(\Delta', A') = \mathsf{split}(A, M)\\ \Delta,\Delta';\Gamma \vdash M : A'\\ A = \forall\Delta'.A'\\ \Delta;\Gamma, x : A \vdash N : B\\ \hline \Delta;\Gamma \vdash \mathbf{let}\ (x : A) = M\ \mathbf{in}\ N : B\end{array}\right]\!\right]$$

Fig. 13. Translation from FreezeML to System F

Figure 13 shows the compilation of FreezeML to System F. It is an extension of the translation from ML to System F of Figure 7. Frozen variables in FreezeML are simply variables in System F. Annotated lambda abstractions are translated to annotated lambda abstractions in System F. Annotated let is translated to annotated let in System F, with a possible type abstraction around the bound term as dictated by split. The translation of let-bindings without type ascriptions remains the same as in the translation from ML to FreezeML.

As an example consider the following translation, where app, auto, and id have the types given in Figure 1. FE: Why is id in italics but auto is not?

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

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

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

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

Theorem 4 (Type preservation). *Let $\Delta; \Gamma \vdash M : A$ hold in FreezeML. Then $\Delta; \Gamma \vdash C[\![M]\!] : A$ holds in System F.*

### 4.3 Is FreezeML Reasonable?

To be usable as a programming language FreezeML must support reasoning principles. We can derive and verify reasoning principles by lifting from System F via the translations. We write $M \simeq N$ to mean $M$ is observationally equivalent to $N$.

*$\beta$-rules.* At a minimum we expect $\beta$-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.

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

The $\beta$-rules use explicit generalisation and instantiation operators. For this purpose it is helpful to consider them as built-in rather than macro-definitions as in Section 2.1. In particular it is important that we treat $V@$ as a value in order to ensure that substitution does not disrupt generalisation. A standard extension to the notion of value, which enables us to treat $V@$ as a value without building in @, is to treat terms of the form $\textbf{let } x = V \textbf{ in } W$ as values.

*$\eta$-rules.* We can also verify that $\eta$-rules hold for FreezeML.

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

In $\textbf{let } x = U \textbf{ in } x$ and $\textbf{let } (x : A) = U \textbf{ in } x$ generalisation and instantiation cancel out. In $\textbf{let } x = \lceil y\rceil \textbf{ in } x$ and $\textbf{let } (x : A) = \lceil y\rceil \textbf{ in } x$ only instantiation takes place. The $\eta$-rule for unannotated lambda abstractions is only defined if the argument type of the function is monomorphic. The $\eta$-rule for annotated lambda abstractions always applies.

*Freezing Let Generalisation.* In ML, due to the value restriction, reduction can change the type of a subterm, but not the type of a program. FreezeML is more subtle. For instance:

$$\begin{array}{lll} \textbf{let } x = (\lambda x.x)\, (\lambda x.x) & \textbf{in } \lceil x\rceil : a \to a \\ \textbf{let } x = \quad \lambda x.x \quad & \textbf{in } \lceil x\rceil : \forall a.a \to a \end{array}$$

Thus:

$$M \simeq M' \not\Longrightarrow \textbf{let } x = M \textbf{ in } N \simeq \textbf{let } x = M' \textbf{ in } N$$

The problem is that let-bindings are always generalised whether we like it or not. It is natural to ask whether, as well as suppressing instantiation of variables with $\lceil - \rceil$, it is also possible (or necessary) to suppress let generalisation; after all System F does neither. We write $\lceil\textbf{let}\rceil$ to denote a "frozen" let-binding that does not perform generalisation.

Frozen let-bindings are helpful for reasoning, as we have:

$$M \simeq M' \implies \lceil \mathbf{let} \rceil\ x = M\ \mathbf{in}\ N \simeq \lceil \mathbf{let} \rceil\ x = M'\ \mathbf{in}\ N$$

Moreover, if $M$ is not a syntactic value, then:

$$\mathbf{let}\ x = M\ \mathbf{in}\ N \simeq \lceil \mathbf{let} \rceil\ x = M\ \mathbf{in}\ N$$

Because of the monomorphism restrictions on unannotated lambdas and ungeneralised let-bindings, it is only possible to macro-express frozen let-bindings for terms of monomorphic type. In this case:

$$\lceil \mathbf{let} \rceil\ x = M\ \mathbf{in}\ N \quad \equiv \quad (\lambda x.M)\, N \quad \equiv \quad \mathbf{let}\ x = \mathrm{id}\, M\ \mathbf{in}\ N$$

Vytiniotis et al. [2010] argue that let-bound variables should not be generalised implicitly. Whilst our goal with FreezeML was to design a conservative extension of ML, another interesting (and simpler) point in the design space would be to build a language without let generalisation and also without \$, but with implicit variable instantiation, $\lceil - \rceil$ and $\lceil \mathbf{let} \rceil$.

FreezeML is a convenient syntactic sugar for *programming* System F. For *reasoning* (and defining a type-preserving reduction semantics) it is preferable to think in terms of \$, @, $\lceil - \rceil$, $\lceil \mathbf{let} \rceil$. If we annotate these operators appropriately then we obtain exactly System F.

## 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 [2008].

### 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 [Miller 1992]. 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 $\Delta$ 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 $\Theta$. 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 $\Delta$ and flexible type variables in a refined environment $\Theta$.

Refined kind environments $\Theta$ are given by the following grammar.

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

We often implicitly treat fixed kind environments $\overline{a}$ as refined kind environments $\overline{a : \bullet}$. The refined kinding rules are given in Figure 14.

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.

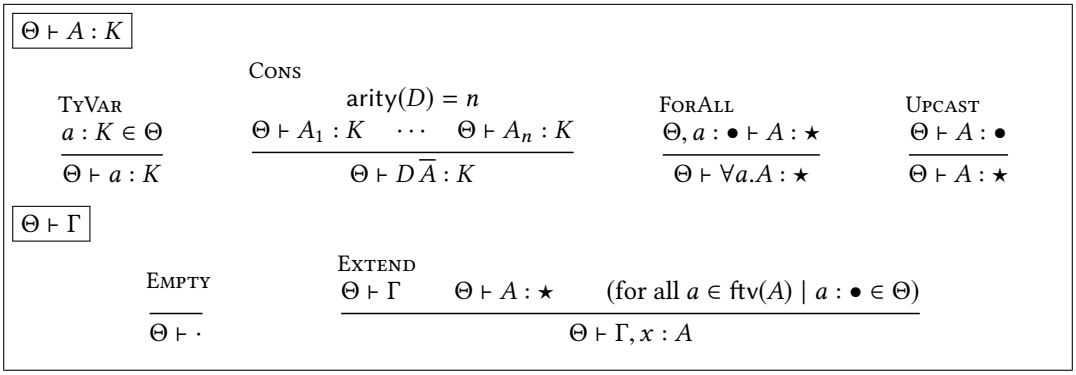


Fig. 14. Refined Kinding Rules

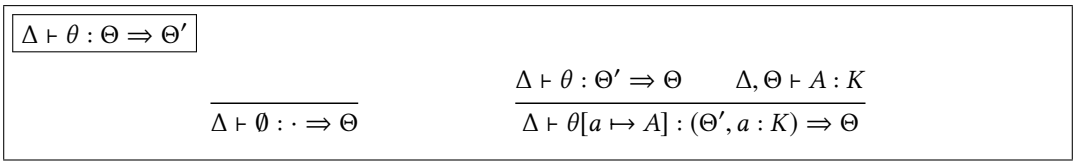


Fig. 15. Refined Type Substitutions

### 5.2 Type Substitutions

In order to define the type inference algorithm we will find it useful to define a judgement for type substitutions $\theta$, which operate on flexible type variables, unlike type instantiations $\delta$, which operate on rigid type variables. The type substitution rules are given in Figure 15. The rules are as in Figure 9, 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 $\iota_\Theta$ for the identity type substitution on $\Theta$, 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 following rules are admissible and we make use of them freely in our algorithms and proofs.

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

$$\textsc{S-Weaken}\ \frac{\Delta \vdash \theta : \Theta \Rightarrow \Theta'}{\Delta, \Delta' \vdash \theta : \Theta \Rightarrow \Theta', \Theta''} \qquad \textsc{S-Strengthen}\ \frac{\Delta \vdash \theta : \Theta \Rightarrow \Theta' \qquad \text{ftv}(\theta) \# \Delta', \Theta''}{\Delta - \Delta' \vdash \theta : \Theta \Rightarrow \Theta' - \Theta''}$$

### 5.3 Unification

A crucial ingredient for type inference is unification. The unification algorithm is defined in Figure 16. It is partial in that it either returns a result or fails. Following Leijen [2008] we explicitly indicate the successful return of a result $X$ by writing return $X$. Failure may be either explicit

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

Fig. 16. Unification Algorithm

or implicit (in the case that an auxiliary function is undefined). The algorithm takes a quadruple $(\Delta, \Theta, A, B)$ of a fixed kind environment $\Delta$, a refined kind environment $\Theta$, and types $A$ and $B$, such that $\Delta, \Theta \vdash A, B$. It returns a unifier, that is, a pair $(\Theta', \theta)$ of a new refined kind environment $\Theta'$ and a type substitution $\theta$, such that $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$.

A type variable unifies with itself yielding the identity type 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 $(\Delta, \Theta_1)$ 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 [2008] 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 type substitution.

Before giving the formal treatment of the correctness properties, we characterise the informal intuition behind them.

- *Soundness* states that if the algorithm returns a pair then the types are unifiable.
- *Completeness* states that if the types can be unified then the inference algorithm will return a pair.
- The *most general unifier* property states that if any unifier exists for $A$ and $B$ then every possible such unifier can be factored through the one returned by the algorithm.

Theorem 5 (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 6 (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* $\theta''$ *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 17. 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 $\Delta$, a refined kind environment $\Theta$, a type environment $\Gamma$, and a term $M$, such that $\Delta; \Theta \vdash \Gamma$. If successful, it returns a triple $(\Theta', \theta, A)$ of a new refined kind environment $\Theta'$, a type substitution $\theta$, 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 [Damas and Milner 1982] adapted to use explicit kind environments $\Delta, \Theta$. 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 $\theta_2$.

*Principality.* As mentioned in Section 3, the Let rule is constrained so that only a principal type can be assigned to $x$ when typechecking the body of the let. This is needed to preclude pathological situations as described in Section 2.3. We now explain the principal constraint and its properties. We first introduce some additional notation, shown in Figure 18.

For the proofs of correctness to go through, we need principal to satisfy the following properties:

*Definition 5.1.* A relation R *captures principality* if it satisfies the following properties:

(1) If $\Delta \Vdash M$ and $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$ and $\mathsf{R}((\Delta, \Theta), \Gamma, M, \Delta', A)$ then $\mathsf{R}((\Delta, \Theta'), \theta(\Gamma), M, \Delta', \theta(A))$.
(2) If $\mathsf{infer}(\Delta, \Theta, \Gamma, M) = (\Theta', \theta, A)$ then $\mathsf{R}(\Delta, \theta(\Gamma), M, \Delta', A)$ where $\Delta' = \mathsf{ftv}(A) - \Delta \subseteq \Theta'$.

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

Fig. 17. Type Inference Algorithm

(3) If $\mathsf{R}(\Delta, \Gamma, M, \Delta', A')$ then for all $A'', \Delta''$, if $\Delta, \Gamma \vdash M : \Delta''.A''$ then there exists $\delta$ such that $\Delta \vdash \delta : \Delta' \Rightarrow_\star \Delta''$ and $\delta(A') = A''$

The first property ensures typing is closed under substitution. The second is needed in the proof of soundness to record evidence in the derivation that the inferred type for $M$ is indeed principal. The third is needed in the proof of completeness for Let, to ensure that the type assigned to $M$ in its derivation was indeed principal.

$$\boxed{\Delta;\Gamma \vdash M : \Delta'.A' \qquad \Delta \vdash A \sqsubseteq B}$$

$$\Delta;\Gamma \vdash M : \Delta'.A' \quad \equiv \quad \Delta' = \mathsf{ftv}(A') - \Delta \text{ and } \Delta,\Delta';\Gamma \vdash M : A' \qquad \frac{\Delta \vdash \delta : \Delta' \Rightarrow_\star \Delta''}{\Delta \vdash \forall\Delta'.H \sqsubseteq \forall\Delta''.\delta(H)}$$

Fig. 18. Auxiliary relations for principality

The most obvious approach is to define principal as follows:

$$\begin{aligned}\mathsf{principal}'(\Delta,\Gamma,M,\Delta',A') = \Delta;\Gamma \vdash M : \Delta'.A' \text{ and}\\ (\text{for all } \Delta'', A'' \mid \text{ if } \Delta;\Gamma \vdash M : \Delta''.A''\\ \text{then } \Delta \vdash \forall\Delta'.A' \sqsubseteq \forall\Delta''.A'')\end{aligned}$$

This obviously satisfies property (3), but it is unclear how to prove that it is closed under substitution. This suggests further strengthening the property so that closure under substitution is built-in. It is worth noting that HMF [Leijen 2008] includes a principality constraint similar to principal′, but does not explain how to validate closure under substitution: even in the proofs in the extended technical report, closure under substitution is left as an "obvious" property.

The approach we take is to define principal in terms of infer, as follows.

$$\begin{aligned}\mathsf{principal}(\Delta,\Gamma,M,\Delta',A) \equiv \Delta;\Gamma \vdash M : \Delta'.A' \text{ and}\\ \mathsf{infer}(\Delta,\Delta',\Gamma,M) = (\Theta,\theta,A) \text{ where } \Delta' \subseteq \Theta,\ \theta(a) = a \text{ on } \Delta'\end{aligned}$$

This definition states that $\forall\Delta'.A'$ is a principal type for $M$ in $\Delta,\Gamma$ if rerunning type inference on it yields no further instantiation of the variables in $\Delta'$. This definition has all of the properties we need, but is, obviously, somewhat unsatisfactory: it feels a bit circular to define the typing judgement in terms of principal types, which are in turn defined in terms of type inference. We believe that principal can be defined in a way that does not depend on the inference algorithm, but all attempts so far have run aground on technicalities. We therefore leave it as a conjecture how to formulate a satisfying declarative definition of principality for FreezeML. In any case, our proofs assume that principal is defined so as to capture principality, but do not rely on the details.

FE: Draft for final definition:

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

*Correctness.* Before giving the formal treatment of the correctness properties, we characterise the informal intuition behind them.

- *Soundness* states that if the algorithm infers a triple then it gives rise to a correct typing judgement for $M$.
- *Completeness* states that if a result exists, then the inference algorithm will find one.
- The *principal types* property states that if any triple exists that yields a correct typing judgement for $M$ then the type component of every possible such triple is an *instance* of the particular type returned by the inference algorithm.

Theorem 7 (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* $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$.

Following Leijen [2008] we state completeness and the principal types property together.

Theorem 8 (Type inference is complete and principal). *If* $\Delta \Vdash M$ *and* $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$ *and* $\Delta, \Theta'; \theta(\Gamma) \vdash M : A$, *then* $\mathsf{infer}(\Delta, \Theta, \Gamma, M) = (\Theta'', \theta', A')$ *where there exists* $\theta''$ *satisfying* $\Delta \vdash \theta'' : \Theta'' \Rightarrow \Theta'$ *such that* $\theta = \theta'' \circ \theta'$ *and* $\theta''(A') = A$.

## 6 RELATED WORK

There are many previous attempts to bridge the gap between ML and System F by extending ML with first-class polymorphism. Some of the proposals stratify the type system, hiding polymorphism inside nominal types [Jones 1997; Läufer and Odersky 1994; Odersky and Läufer 1996; Rémy 1994]; others add features to the type system [Le Botlan and Rémy 2003; Leijen 2009; Russo and Vytiniotis 2009]; and others strive to stay within the System F type system whilst minimising the number of type annotations [Dunfield and Krishnaswami 2013; Leijen 2008; Serrano et al. 2018; Vytiniotis et al. 2006, 2008]. Below we will briefly discuss some of these systems.

MLF [Le Botlan and Rémy 2003] (sometimes stylised as $\mathrm{ML}^{\mathrm{F}}$) is considered to be the most expressive of the conservative ML extensions so far (c.f. Table 1 in Section 2). 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 [Le Botlan and Rémy 2003; Leijen 2007]. MLF also extends ML with type annotations on function 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. As a consequence, in MLF a term with two different type annotations can have two different, incomparable principal types. At the same time, if a binder is not used in a polymorphic way and is only "passed through" then its type variables can always be implicitly instantiated with polymorphic types. This means that a term $(\lambda x.x\ \mathsf{id})\ \mathsf{auto}$ is well-typed and does not require an annotation on the binder $x$. More generally, whenever an application $M\ N$ is well-typed in MLF, then so are the app $M\ N$ and revapp $N\ M$ applications. This last property is also true for HML, QML, and HMF, described below.

HML [Leijen 2009] is a simplification of MLF. In HML all polymorphic function arguments require annotations. This allows the type system to be restricted to significantly simplify the type inference algorithm compared to MLF, though polymorphically bounded types are still required in HML. Otherwise HML maintains the key features of MLF. Polymorphic instantiations are still inferred automatically. For example, identically to MLF, expression single id on its own receives a type $\forall a.\mathsf{List}\ (a \to a)$, but a polymorphic instantiation on single will be inferred whenever required, e.g. map poly (single id) is well-typed and has type $\mathsf{List}\ (\mathsf{Int} \times \mathsf{Bool})$. HML programs are also stable under rewrites, such as inlining let-bindings or abstracting common subexpressions to let-bindings.

The QML [Russo and Vytiniotis 2009] system distinguishes explicitly between polymorphic schemes and (potentially quantified) types. In QML let-bound expressions are assigned polymorphic schemes, enabling standard ML-style let-polymorphism, whereas a function parameter receives a type, not a scheme. It is therefore valid to write **let** id $x = x$ **in** (id 3, id false), since id is assigned a scheme $\Pi(a)(a \to a)$, but it is invalid to write $\lambda f.(\mathsf{f}\ 3, \mathsf{f}\ \mathsf{false})$ since $f$ is assigned a type $a \to a$ and not a scheme $\Pi(a)(a \to a)$. Neither **let**- nor $\lambda$-bound variables can be annotated with a type. Moreover, QML never infers introduction of quantifiers and, unlike in MLF and HML, quantified types are never implicitly used for instantiation. All polymorphic instantiations must therefore be made explicitly by annotating terms at call sites. Unlike in System F the programmer does not provide instantiations for the type variables but rather the whole polymorphic type itself. Therefore, fixing the example lambda abstraction above requires writing $\lambda f.(\mathsf{f}\ \{\forall a.a \to a\}\ 3, \mathsf{f}\ \{\forall a.a \to a\}\ \mathsf{false})$. Russo and Vytiniotis [2009] also consider extending QML with derived typing rules that would admit $\lambda(f : \forall a.a \to a).(f\ 3, f\ \mathsf{false})$ and thus reduce the amount of required annotations.

HMF [Leijen 2008] contrasts with the above systems in that it only uses regular System F types. To support polymorphic instantiations HMF, just like the systems above, requires type

annotations on function parameters that are used polymorphically. HMF however makes local decisions whether an instantiation is to be mono- or polymorphic. In unambiguous cases no annotations are required. In case of ambiguity HMF chooses a predicative instantiation and always introduces the least inner polymorphism possible. For example, a term single id is assigned a type $\forall a.\mathsf{List}(a \to a)$. An explicit type annotation must be provided at a call site if a polymorphic instantiation is desired instead. Thus $(\mathsf{single} :: (\forall a.a \to a) \to \mathsf{List}(\forall a.a \to a))\ \mathsf{id}$ has the type $\mathsf{List}(\forall a.a \to a)$. To disambiguate instantiations HMF considers all arguments in an application. HMF has simpler metatheory compared to MLF, HML, and QML, at the expense of possibly requiring more annotations.

Several systems for impredicative polymorphism were proposed in the context of the Haskell programming language (as implemented by the GHC compiler). These systems include boxy types [Vytiniotis et al. 2006], FPH [Vytiniotis et al. 2008], and GI [Serrano et al. 2018]. The boxy types system, used to implement GHC's `ImpredicativeTypes` extension, was very fragile and thus difficult to use in practice. For example, extra annotations were sometimes required even if polymorphic instantiations were unambiguous. Moreover, even if an impredicative application $M\ N$ was accepted, both app $M\ N$ and revapp $N\ M$ would require extra annotations. See [Leijen 2007] for a more detailed comparison of boxy types with MLF. The FPH system was simpler for the user but it was difficult to implement in practice. GI is the latest development in this line of research. It offers expressiveness comparable with FreezeML (see Section 2), and the authors show how to combine their system with the OutsideIn(X) [Vytiniotis et al. 2011] 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. However, as discussed in Section 2, FPH and GI sometimes require more annotations than FreezeML, and GI does not support let-generalisation.

## 7 CONCLUSIONS AND FUTURE WORK

In this paper, we have introduced FreezeML as a purely theoretical development. We have also implemented FreezeML as part of the Links programming language [Cooper et al. 2006], 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. All of the examples in Table 1 (modulo syntactic differences) type as expected and, perhaps more importantly, all of the failing examples from Section 2.3 are reported as errors.

Directions for future work include extending the FreezeML approach to accommodate richer features such as higher-kinds, GADTs, and dependent types, as well as exploring different implicit instantiation strategies (as discussed in Section 2.3). We also believe it would be instructive to rework our formal account using the methodology of Gundry et al. [2010] and use that as the basis for mechanised soundness and completeness proofs.

## REFERENCES

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

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/

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

Joshua Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ICFP*. ACM, 429–442.

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

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

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

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

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

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

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

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

Simon Marlow (Ed.). 2010. *Haskell 2010 Language Report*.

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

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

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

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

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.

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

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

Dimitrios Vytiniotis, Simon L. Peyton Jones, and Tom Schrijvers. 2010. Let should not be generalized. In *TLDI*. ACM, 39–50.

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

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

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

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

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

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

## A PROOFS FROM SECTION 4

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

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

$$\text{where } \overline{A} \text{ may be empty}\ \frac{\Delta, \Delta';\Gamma \vdash V : A}{\Delta;\Gamma \vdash \Lambda\,\Delta'.V : \forall \Delta'.A}\ \text{F-PolyLam}^*$$

Lemma A.1. *For any $A, B, a, x, y$ where $\Delta \vdash A : \star$ and $\Delta \vdash \forall a.B : \star$ and $a, x, y$ are fresh for $\Delta, \Gamma$, the judgement $\Delta;\Gamma \vdash \lambda(x : \forall a.B).\ \mathbf{let}\ (y : B[A/a]) = x\ \mathbf{in}\ \lceil y \rceil : \forall a.B \to B[A/a]$ is derivable in FreezeML.*

Proof. First, we write $B$ as $\forall \Delta'.H$, where $\Delta'$ may be empty. The derivation is as follows:

$$\dfrac{\dfrac{(x : \forall a, \Delta'.H) \in (\Gamma, x : \forall a, \Delta'.H) \qquad \vdash \delta : a, \Delta' \Rightarrow \Delta}{\Delta;\Gamma, x : \forall a.B \vdash x : \delta H} \qquad \dfrac{y : B[A/a] \in \Gamma, y : B[A/a]}{\Delta;\Gamma, y : B[A/a] \vdash \lceil y \rceil : B[A/a]}}{\dfrac{\Delta;\Gamma, x : \forall a.B \vdash \mathbf{let}\ (y : \forall \Delta'.H[A/a]) = x\ \mathbf{in}\ \lceil y \rceil : B[A/a]}{\Delta;\Gamma \vdash \lambda(x : \forall a.B).\ \mathbf{let}\ (y : B[A/a]) = x\ \mathbf{in}\ \lceil y \rceil : \forall a.B \to B[A/a]}}$$

Here, $\delta$ is defined such that $\delta(a) = A$ and $\delta(b) = b$ for all $b \in \Delta'$. Therefore, $\delta H = H[A/a]$. □

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

- Case F-Var:

$$\frac{x : A \in \Gamma}{\Delta;\Gamma \vdash x : A} \quad \rightsquigarrow_F \quad \frac{x : A \in \Gamma}{\Delta;\Gamma \vdash \lceil x \rceil : A}$$

- Case F-Lam:

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

- Case F-App:

$$\frac{\Delta;\Gamma \vdash M : A \to B \qquad \Delta;\Gamma \vdash N : A}{\Delta;\Gamma \vdash M\,N : B} \rightsquigarrow_F \frac{\Delta;\Gamma \vdash \mathcal{E}[\![M]\!] : A \to B \qquad \Delta;\Gamma \vdash \mathcal{E}[\![N]\!] : A}{\Delta;\Gamma \vdash \mathcal{E}[\![M]\!]\,\mathcal{E}[\![N]\!] : B}$$

- Case F-TAbs-F-Var-F-TApp: In the case of a type abstraction wrapping a variable with zero or more type applications, we proceed as follows:

$$\dfrac{\dfrac{\dfrac{x : \forall \Delta'' \Delta'''.H \in \Gamma}{\Delta, \Delta'; \Gamma \vdash x : \forall \Delta'' \Delta'''.H} \qquad \Delta'' = a_1, \ldots, a_n \qquad \overline{A} = A_1, \ldots, A_n}{\Delta, \Delta'; \Gamma \vdash x\,\overline{A} : \forall \Delta'''.H[A_1/a_1] \cdots [A_n/a_n]}}{\Delta; \Gamma \vdash \Lambda \Delta'.x\,\overline{A} : \forall \Delta'.B} \rightsquigarrow_F$$

$$\dfrac{\dfrac{x : \forall \Delta'' \Delta'''.H \in \Gamma \qquad \vdash \delta : \Delta'' \Delta''' \Rightarrow \Delta}{\Delta; \Gamma \vdash x : \delta(H)} \qquad \dfrac{y : \forall \Delta'.B \in \Gamma, (y : \forall \Delta'.B)}{\Delta; \Gamma, (y : \forall \Delta'.B) \vdash \lceil y \rceil : \forall \Delta'.B}}{\Delta; \Gamma \vdash \mathbf{let}\ (y : \forall \Delta'.B) = x\ \mathbf{in}\ \lceil y \rceil : \forall \Delta'.B}$$

  Here, $\delta : \Delta'' \Delta''' \Rightarrow \Delta$ is defined such that $\delta(a) = a$ for all $a \in \Delta'''$ and $\delta(a_i) = A_i$ for all $a_i \in (a_1 \ldots a_n) = \Delta''$, and $B = \forall \Delta'''.H[A_1/a_1] \cdots [A_n/a_n]$.
- Case F-TLam-F-Lam: In the case of a series of type abstractions wrapping a lambda-abstraction, we proceed as follows:

$$\dfrac{\Delta, \Delta'; \Gamma \vdash \lambda x^A.M : B}{\Delta; \Gamma \vdash \Lambda \Delta'.\lambda x^A.M : \forall \Delta'.B} \rightsquigarrow_F \dfrac{\Delta, \Delta'; \Gamma \vdash \lambda x^A.\mathcal{E}[\![M]\!] : B \qquad \dfrac{y : \forall \Delta'.B \in \Gamma, (y : \forall \Delta'.B)}{\Delta; \Gamma, (y : \forall \Delta'.B) \vdash \lceil y \rceil : \forall \Delta'.B}}{\Delta; \Gamma \vdash \mathbf{let}\ (y : \forall \Delta'.B) = \lambda x^A.\mathcal{E}[\![M]\!]\ \mathbf{in}\ \lceil y \rceil : \forall \Delta'.A}$$

- Case F-TApp: Finally, in the case of a type application we proceed as follows:

$$\dfrac{\Delta; \Gamma \vdash M : \forall a.B}{\Delta; \Gamma \vdash M\,A : B[A/a]} \rightsquigarrow_F \dfrac{\Delta; \Gamma \vdash F : \forall a.B \to B[A/a] \qquad \Delta; \Gamma \vdash \mathcal{E}[\![M]\!] : \forall a.B}{\Delta; \Gamma \vdash F\ \mathcal{E}[\![M]\!] : B[A/a]}$$

  where the first subderivation is provided by Lemma A.1, where $F = \lambda(x : \forall a.B).\ \mathbf{let}\ (y : B[A/a]) = x\ \mathbf{in}\ \lceil y \rceil$.

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

Theorem 4 (Type preservation). *Let $\Delta; \Gamma \vdash M : A$ hold in FreezeML. Then $\Delta; \Gamma \vdash C[\![M]\!] : A$ 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 $C[\![-]\!]$ can be extended to a function returning the desired derivation.

- Case Freeze:

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

- Case Var:

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

- Case Lam:

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

- Case Lam-Ascribe

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

- Case App:

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

- Case Let: In this case there are two subcases, depending on whether $M$ is a guarded value or not.
  - $M=V\in\mathrm{GVal}$: In this case, we have $\mathrm{gen}(\Delta,A',M)=(\Delta',\Delta')$ for some possibly nonempty $\Delta'$, and $(\Delta,\Delta',M,A')\Updownarrow\forall\Delta'.A'$. We proceed as follows:

$$C\left[\!\!\left[\frac{\Delta,\Delta';\Gamma\vdash V:A'\qquad\Delta;\Gamma,x:\forall\Delta'.A'\vdash N:B}{\Delta;\Gamma\vdash \mathbf{let}\ x=V\ \mathbf{in}\ N:B}\right]\!\!\right]\Longrightarrow\frac{\dfrac{\Delta,\Delta';\Gamma\vdash C[\![V]\!]:A'}{\Delta;\Gamma\vdash \Lambda\Delta'.C[\![V]\!]:\forall\Delta'.A'}\qquad\Delta;\Gamma,x:\forall\Delta'.A'\vdash C[\![N]\!]:B}{\Delta;\Gamma\vdash \mathbf{let}\ x^{A}=\Lambda\Delta'.C[\![V]\!]\ \mathbf{in}\ C[\![N]\!]:B}$$

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

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

    where we make use of a standard substitution lemma for System F to instantiate type variables from $\Delta'$ 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 $\Delta'$ since all inferred types are translated to explicit annotations.
- Case Let-Ascribe: This case is analogous to the case for Let.

□

## B 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. Note that when types appear on their own or in contexts $\Gamma$, we identify $\alpha$-equivalent types.

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

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 $\theta$, 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 B.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 B.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 B.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 $\theta$ is empty then $\Theta$ 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 B.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 B.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 $\Theta, \Theta', \Delta$. 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 B.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 $\Delta \vdash \theta \Rightarrow \Theta'$ 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 B.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 $a \in \mathrm{dom}\Delta$ this is immediate. If $a \in \mathrm{dom}(\Theta')$, then since $a \in \mathrm{ftv}(\theta(A))$ we know that there must exist $b \in \mathrm{dom}(Theta)$ 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 B.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 $\Delta, \Theta \vdash \delta : \Delta' \Rightarrow_K \Delta'$ then $\Delta, \Theta' \vdash \theta \circ \delta_1 : \Delta' \Rightarrow_K \Delta''$.*

Proof. In both cases, by straightforward induction on derivations. □

Lemma B.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 $\Theta$ and $\Theta'$, so that the induction hypothesis applies. □

Lemma B.9. *If $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$ and $\Delta' \# \Delta, \Theta'$ 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 $\Delta'$ is fresh for $\Delta$ and $\Theta'$. 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 $\Delta'$ is clearly fresh for $\Delta, \Theta, \Theta'$) we have $\Delta, \Delta' \vdash \theta : \Theta \Rightarrow \Theta'$. Moreover, by weakening (Lemma B.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 B.10. *Let the following conditions hold:*

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

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

Proof. According to Lemma B.14, (7) yields $\Delta, \Theta'' \vdash A$ **(9)**.
Let $\Delta' = (a'_1, \ldots, a'_n)$ for some $n \geq 0$ and let $\Delta_F = (f_1, \ldots, f_n)$ for pairwise different, fresh type variables $f_i$.
By (9), we have $\mathrm{ftv}(A) \subseteq \Delta, \Theta''$. Let $\Theta_{\theta'}$ be defined as $\mathrm{ftv}(\theta') - \Delta$. We then have $\Theta_{\theta'} \subseteq \Theta''$ **(10)** and $\Delta' \# \Theta_{\theta'}$ **(11)** and $\Delta' \subseteq \mathrm{ftv}(\Theta'')$ **(12)**.
By (1) to (3) we have $\Delta, \Theta' \vdash \theta''(a) : K$ for all $(a : K) \in \Theta_{\theta'}$ **(13)**.

Let $\theta''_F$ be defined such that

$$\theta''_F(a) = \begin{cases} \theta''(a) & \text{if } a \in \Theta_{\theta'} \\ f_i & \text{if } a = a'_i \in \Delta' \\ A_D & \text{if } a \in \Theta'' - \Theta_{\theta'} - \Delta' \end{cases} \quad \textbf{(14)}$$

where $A_D$ is some arbitrary type with $\Delta, \Theta' \vdash A_D : \bullet$. By (10) to (12), this definition is well-formed. Together with (13) we then have $\Delta \vdash \theta''_F : \Theta'' \Rightarrow \Theta', \Delta_F$ **(15)** and $\theta = \theta''_F \circ \theta'$ **(16)**.

By (9) and Lemma B.5, we then have $\Delta, \Theta', \Delta_F \vdash \theta''_F A$ which implies $\mathrm{ftv}(\theta''_F A) \subseteq \Delta_F, \Delta, \Theta'$. In general, for every $a \in \mathrm{ftv}(A)$, $\theta''_F(a)$ is part of $\theta''_F(A)$. In particular, for each $a'_i \in \Delta' \subseteq \mathrm{ftv}(A)$, $\theta''_F(a'_i) = f_i$ occurs in $\theta''_F(A)$. Thus, $\mathrm{ftv}(\theta''_F A) - \Delta, \Theta' = \Delta_F$ holds **(17)**.

By (7) and (15) and **??**, Lemma B.16 yields $\Delta, \Theta', \Theta_F; \theta''_F \theta' \Gamma \vdash M : \theta''_F A$, which by (16) is equivalent to $\Delta, \Theta', \Delta_G; \theta\Gamma \vdash M : \theta''_F(A)$ **(18)**. According to condition 3 of definition 5.1 as well as (8), (17) and (18) there exists $\delta$ such that $\Delta \vdash \delta : \Delta'' \Rightarrow \Delta_F$ **(19)** and $\delta(A') = \theta''_F(A)$.

By (6), the latter is equivalent to $\delta(\theta''(A)) = \theta''_F(A)$ **(20)**

Let $a \in \Delta' \subseteq \mathrm{ftv}(A)$, which implies $a = a'_i$ for some $1 \leq i \leq n$. By (20), we have

$$\begin{array}{rlll} & \delta\theta''(a_i) & = & \theta''_F(a_i) \\ \text{equiv.} & \delta\theta''(a_i) & = & f_i \qquad\qquad \text{(by (14))} \end{array}$$

We therefore have that for each such $a'_i$, $\theta''(a_i)$ maps to pairwise different type variables $b_i$. By (19) and $\Theta', \Delta'' \# \Delta_F$, we have $\delta(b_i) \neq b_i$ and therefore $b_i \in \Delta''$. We have therefore shown that $\theta''$ maps all $a \in \Delta'$ to pairwise different variables in $\Delta''$ **(21)**.

We now show that $\theta''(\Delta')$ is a permutation of $\Delta''$. 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 (6) we have that there must exist $a \in \mathrm{ftv}(A)$ such that $b \in \mathrm{ftv}(\theta''(a))$. By (21), $a \in \Delta'$ would immediately yield a contradiction. By $\mathrm{ftv}(A) \subseteq \Theta_{\theta'}, \Delta'$, we therefore consider the case $b \in \Theta_{\theta'}$. According to (13), we then have $\mathrm{ftv}(\theta''(b)) \subseteq \Delta, \Theta'$, which is disjoint from $\Delta''$. As this yields another contradiction, we have shown that $\theta''(\Delta')$ is a permutation of $\Delta''$

We now show that $\theta''(\Delta') = \Delta''$ holds (i.e., $\theta''$ preserves the order of type variables). By $\mathrm{ftv}(A) \subseteq \Theta_{\theta'}, \Delta'$ and (13), we have that for all $b \in \mathrm{ftv}(A) - \Delta'$, we have $\mathrm{ftv}(\theta''(b)) \subseteq \Delta, \Theta' \# \Delta''$. Therefore, together with (6) for all $a'_i, a'_j \in \Delta'$ with $0 \leq i < j \leq n$, we have that the first occurrence of $\theta''(a'_i)$ in $A'$ is located before the first occurrence of $\theta''(a'_j)$ in $A'$.

□

Lemma B.11. *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 $\Theta_D$. 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:

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

□

FE: The following lemma is no major result by itself, as it relies on completeness and soundness. It just states that if a term has a principal type $A$ and we infer a type $A'$ for that term, then there is a bijection between their "generalizable" type variables. (We could also express this property by actually quantifying $A$ and $A'$ and using alpha-equivalence)

Lemma B.12 (Inferred types are principal). *Let the following conditions hold:*

$$\Delta, \Theta \vdash \Gamma \quad \textbf{(1)}$$
$$\Delta, \Theta \Vdash M \quad \textbf{(2)}$$
$$\Delta, \Theta, \Delta'; \Gamma \vdash M : A \quad \textbf{(3)}$$
$$\Delta' = \mathsf{ftv}(A) - \Delta, \Theta \quad \textbf{(4)}$$
$$\mathsf{principal}((\Delta, \Theta), \Gamma, \Delta', A) \quad \textbf{(5)}$$
$$\mathsf{infer}(\Delta, \Theta, \Gamma, M) = (\Theta', \theta, A') \quad \textbf{(6)}$$
$$\Delta'' = \mathsf{ftv}(A') - \Delta - \mathsf{ftv}(\theta) \quad \textbf{(7)}$$

*Then there exists $\delta$ such that $\Delta, \Theta \vdash \delta : \Delta' \Rightarrow_\star \Delta''$ and $\delta(\Delta') = \Delta''$ and $\delta(A) = A'$.*

Proof. We have $\Delta \vdash \iota_{\Delta,\Theta} : \Theta \Rightarrow \Theta$ **(8)** and therefore by $\Delta' \# \Theta$ and weakening also $\Delta \vdash \iota_{\Delta,\Theta} : \Theta \Rightarrow \Theta, \Delta'$ **(9)**. Trivially, we can rewrite (3) and (5) as $\Delta, \Theta, \Delta'; \iota_{\Delta,\Theta}(\Gamma) \vdash M : A$ **(10)** and $\mathsf{principal}((\Delta, \Theta), \iota_{\Delta,\Theta}(\Gamma), \Delta', A)$ **(11)**, respectively. We can apply theorem 8, using (2), (9) and (10), which yields existence of $\theta''$ such that $\Delta \vdash \theta'' : \Theta' \Rightarrow \Theta$ **(12)** and $\iota_{\Delta,\Theta} = \theta'' \circ \theta$ **(13)** and $\theta''(A') = A$ **(14)**.

We apply Lemma B.10 using (3), (4), (7), (8), (11) and (12) to (14), which yields $\theta''(\Delta'') = \Delta'$. As $\theta''$ is a bijection between $\Delta''$ and $\Delta'$, we then choose $\delta$ to be the inverse of $\theta''$, restricted to $\Delta'$.

FE: We also need to show that $\Delta'' \# \Theta, \Delta$ holds, for which we need a freshness lemma.

□

Lemma B.13 (Stability of principality under substitution). *Let the following conditions hold:*

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

*Then* $\mathsf{principal}((\Delta, \Theta'), \theta\Gamma, \Delta'', \theta A)$ *holds, where* $\Delta'' = \mathsf{ftv}(\theta A) - \Delta, \Theta'$.

Proof. Let $A_p$ and $\Delta_p$ such that $\Delta_p = \mathsf{ftv}(A_p) - \Delta, \Theta'$ and $\Delta, \Theta', \Delta_p; \theta\Gamma \vdash M : A_p$. FE: We show that there exists $\delta$ such that $\Delta, \Theta', \Delta_p \vdash \delta : \Delta'' \Rightarrow \cdot$ and $\delta(\theta A) = A_p$

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

$$\Delta \vdash \theta'' : \Theta'' \Rightarrow \Theta', \Delta_p \tag{6}$$
$$\theta = \theta'' \circ \theta' \tag{7}$$
$$\theta''(A') = A_p \tag{8}$$

Let $\Delta''' = \mathsf{ftv}(A') - \Delta - \mathsf{ftv}(\theta')$. FE: We should then have that there exists $\delta_b$ such that $\Delta, \Theta, \Delta''' \vdash \delta_b : \Delta' \Rightarrow \cdot$ and $\delta(\Delta') = \Delta'''$ and $\delta_b(A) = A'$ **(9)**

We show

$$\begin{array}{lll} & A_p & \\ = & \theta''(A') & \text{(by (8))} \\ = & \theta''\theta'(A') & \text{(FE: todo by idempotency of } \theta' \text{ on } A') \\ = & \theta(A') & \text{(by (7))} \\ = & \theta(\delta_b(A)) & \text{(by (9))} \\ = & \delta_b(\theta((A)) & \text{(TODO)} \end{array}$$

□

Lemma B.14. *If* $\Delta; \Gamma \vdash M : A$, *then* $\Delta \vdash \Gamma$ *and* $\Delta \vdash A : \star$.

Proof. We adopt a convention that typing judgements $\Delta; \Gamma \vdash M : A$ are only constructed when $\Delta \vdash \Gamma$ and $\Delta \vdash A : \star$. Alternatively, we can build appropriate checks into the typing judgement so that these judgements can be proved by structural induction. □

Lemma B.15. *If* $\Delta \Vdash M$, *and* $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$, *then:*

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

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

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 $\Delta, \Theta$. Since $\theta$ only affects type variables in $\Theta$, it follows that $a \in \mathsf{ftv}(\theta(A))$. Moreover, by assumption $\mathsf{ftv}(A) \# \Theta'$ 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 $\Delta, \Theta'$. Since $a \notin \Delta, \Theta'$, we must have $\theta(a) = a$ since $\theta$ was a well-formed substitution mentioning only type variables in $\Delta, \Theta'$. This implies that $a \in \mathrm{ftv}(A)$ since $a$ cannot have been introduced by $\theta$.

(2) For part 2: We consider two cases.

- If the derivation is of the form

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

then we may derive

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

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

- If the derivation is of the form

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

where we assume without loss of generality that $\Delta' \# \Delta, \Theta, \mathrm{ftv}(\theta)$. Then first we observe (by property B.7) that $\Delta, \Theta' \vdash \theta \circ \delta : \Delta' \Rightarrow_\bullet \cdot$, so we can derive

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

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

□

Lemma B.16. *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. 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 $\Delta'$ 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 B.7 we have $\Delta, \Theta' \vdash \theta \circ \delta : \Delta' \Rightarrow_! poly\cdot$. 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 $\Gamma$ 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 $\theta$ is a well-kinded substitution, 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 $M = \lambda(x : A_0).M_0$:

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

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 $\theta$ are those in $\Theta$, which is disjoint from $\Delta$, we know that $\theta(A_0) = A_0$. Thus, we can proceed as follows:

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

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 $\Delta \Vdash M$ 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 $M = \textbf{let}\ x = M_0\ \textbf{in}\ N_0$. In this case we have derivations of the form:

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

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

To apply the induction hypothesis to $M_0$, we need to extend $\theta$ to a substitution $\theta'$ satisfying $\Delta \vdash \theta' : \Theta, \Delta'' \Rightarrow \Theta', \Delta''$, which can be done by weakening by $\Delta''$. Then by induction we have $\Delta, \Theta', \Delta''; \theta'(\Gamma) \vdash M_0 : \theta'(A')$. Since $\theta'$ acts as the identity on $\Delta''$ its behaviour is the same as $\theta$ 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 B.15(1), we have that $(\Delta', \Delta'') = \mathsf{gen}((\Delta, \Theta'), \theta(A'), M_0)$ and by Lemma B.15(2), we also know that $((\Delta, \Theta'), \Delta'', M_0, \theta(A')) \Updownarrow \theta(A_0)$. Finally, because principal captures principality, it preserves substitution so $\mathsf{principal}((\Delta, \Theta'), \theta(\Gamma), M_0, \Delta'', \theta(A'))$.
We can conclude:

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

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

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

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

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 $\theta$ only replaces variables in $\Theta$, which is disjoint from $\Delta$. 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:

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

□

# C CORRECTNESS OF UNIFICATION PROOFS

## C.1 Soundness of unification

Lemma C.1. *If* $\Theta' = \mathsf{demote}(K, \Theta, \Delta)$ *then* $\mathsf{dom}(\Theta) = \mathsf{dom}(\Theta')$ *and* $\Delta \vdash \iota : \Theta \Rightarrow \Theta'$.

Proof. Proof by case analysis on $K$ and induction on $\Theta$. 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 $\mathsf{demote}(K, \Theta, \Delta) = \mathsf{demote}(K, \Theta_1, \Delta), a : K'$, where $\Theta_1' = \mathsf{demote}(K, \Theta_1, \Delta)$ and $K'$ is $\bullet$ if $a \in \Delta$, otherwise $K = K'$. Then by induction we have

$\mathsf{dom}(\Theta_1) = \mathsf{dom}(\Theta_1')$ and $\Delta \vdash \iota : \Theta_1 \Rightarrow \Theta_1'$. Clearly, $\mathsf{dom}(\Theta_1, a : K) = \mathsf{dom}(\Theta_1', a : K')$. 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. □

Theorem 5 (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. By considering the cases where unification succeeds and theorem assumptions hold.

(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 $\mathsf{dom}(\Theta_1) = \mathsf{dom}(\Theta)$. Therefore

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

Next, by Lemma C.1 we know that $\Delta \vdash \iota : \Theta \Rightarrow \Theta_1$. Moreover, by assumption $\Delta, \Theta_1 \vdash A : K$ so we can conclude that $\Delta \vdash \iota[a \mapsto A] : \Theta, a : K \Rightarrow \Theta_1$.

(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 $n = \mathsf{arity}(D)$. From the inductive hypothesis we get $\theta_1 = \iota_{\Delta,\Theta}$ and $\theta'$ such that $\theta'(A_i) = \theta'(B_i)$, $\theta_{i+1} = \theta' \circ \theta_i$, and $\Delta \vdash \theta_{i+1} : \Theta_{i+1} \Rightarrow \Theta$. From Lemma B.1 we know that a substitution $\theta_{i+1}$ maintains equalities established by $\theta_i$, and so, by induction on $i$, the final substitution $\theta$ unifies each $A_i$ with corresponding $B_i$. 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 $\Delta \vdash \theta : \Theta_{n+1} \Rightarrow \Theta$.
(4) $\mathsf{unify}(\Delta, \Theta, \forall a.A, \forall b.B)$: In this case we must have

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

so from the inductive hypothesis we have $\theta(A[c/a]) = \theta(B[c/b])$ **(2)**, where $c$ is fresh and $\Delta \vdash \theta : \Theta \Rightarrow \Theta_1$. We now derive:

$$\begin{array}{rl} & \theta(\forall a.A) \\ = & \quad \{(4), \text{Lemma B.2}\} \\ & \theta(\forall c.A[c/a]) \\ = & \quad \{\text{Definition of substitution}\} \\ & \forall c.\theta(A[c/a]) \end{array}$$

and by exactly the same reasoning, $\theta(\forall b.B) = \forall c.\theta(B[c/b])$. Then by (2) 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 $\Delta$ without damaging the well-formedness of $\theta$.

□

### C.2 Completeness of unification

Lemma C.2. *If* $\mathsf{demote}(K', \Theta, \mathsf{ftv}(B) - \Delta) = \Theta'$ *and* $\Delta, \Theta \vdash B : K$ *then* $\Delta, \Theta' \vdash B : K'$.

Proof. We consider the possible cases for $K'$. If $K' = \star$ then the conclusion follows immediately by Upcast (if necessary). If $K' = \bullet$ then since all of the flexible free type variables of $B$ are demoted to $\bullet$ in $\Theta'$, it is straightforward to show by induction on the structure of $B$ that $\Delta, \Theta' \vdash B : \bullet$. □

Theorem 6 (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* $\theta''$ *satisfying* $\Delta \vdash \theta'' : \Theta'' \Rightarrow \Theta'$ *such that* $\theta = \theta'' \circ \theta'$.

Proof. By induction on the shape of types $A$ and $B$. That is, we assume the induction hypothesis holds for all $A'$ and $B'$ that appear as strict subexpressions of $A$ and $B$ respectively, and proceed by case analysis of the possible forms of $A$ and $B$.

(1) Case $A = a = B$: In this case $\mathsf{unify}(\Delta, \Theta, a, a)$ succeeds and returns $(\Theta, \iota_\Theta)$. Moreover, we may choose $\theta'' = \theta$ and conclude that $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$ and $\theta = \theta \circ \iota$, 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 $K'$. Also, since types are finite syntax trees we must have $a \neq \mathsf{ftv}(B)$. We now define $\Theta_1 = \mathsf{demote}(K', \Theta'_1, \mathsf{ftv}(B) - \Delta)$. Since $\Delta, \Theta \vdash B : K$, and $a \notin \mathsf{ftv}(B)$, we have $\Delta, \Theta'_1 \vdash B : K$ and by Lemma C.2, we also have that $\Delta, \Theta_1 \vdash B : K'$. Hence unification succeeds in this case with $\mathsf{unify}(\Delta, \Theta, a, B) = (\Theta_1, \iota[a \mapsto B])$. Finally, we choose $\theta''$ to agree with $\theta$ on $\Theta_1$, and undefined on $a$. Thus, $\Delta \vdash \theta'' : \Theta_1 \Rightarrow \Theta'$ because $\theta''(b) = \theta(b)$ on $\Theta_1$ and all free type variables of $B$ (which are mapped to $K'$ in $\Theta_1$) must have kind $K'$. (If $K' = \star$ then this is immediate, if not then $\theta(B)$ must be a monotype so all free type variables in it must also be mapped to monotypes by $\theta = \theta''$; however, no other type variables' kinds change). Clearly, $\theta'' \circ (\iota[a \mapsto B]) = (\theta'' \circ \iota)[a \mapsto \theta''(B)] = \theta$ since $\theta''$ agrees with $\theta$ on all variables other than $a$, while $\theta''(a)$ is undefined and $\theta(a) = \theta(B)$.

(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 $i \in 0, \ldots, n$. We proceed by induction on $i$:

  (a) $i = 0$: unification succeeds with $\theta' = \theta_1 = \iota$ and the theorem holds for $\theta'' = \theta$ and $\Theta'' = \Theta$.

  (b) $0 < i \leq n$: by induction, if theorem holds for $i - 1$ then we have

$$(\Theta_{i+1}, \theta_t) = \mathsf{unify}(\Delta, \Theta_i, \theta_i(A_i), \theta_i(B_i))$$

  and $\theta_{i+1} = \theta_t \circ \theta_i$ (by definition of unify). By induction hypothesis we have that $\theta = \theta'' \circ \theta_{i+1}$, and therefore the unification succeeds with $\theta' = \theta_{n+1}, \Theta'' = \Theta_{n+1}$ and we have $\theta = \theta'' \circ \theta_{n+1}$.

(4) $\theta(\forall a.A) = \theta(\forall b.B)$: we take fresh $c \notin \mathsf{ftv}(\theta, A, B)$. By Lemma B.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 $(\Theta_1, \theta')$ such that $\theta = \theta'' \circ \theta'$ and $\Delta, c \vdash \theta'' : \Theta_1 \to \Theta'$. **(1)**. Since we place $c$ in a general kind environment during unification we know that $c \notin \mathsf{ftv}(\theta')$ **(2)**. This means that $\mathsf{unify}(\Delta, \Theta, \forall a.A, \forall b.B)$ succeeds with $(\Theta_1, \theta')$ **(3)**. From this and (1) we can conclude using the same witnessing $\theta''$ which satisfies $\theta = \theta'' \circ \theta'$ and by (3) and the fact that $c$ is fresh for both $\theta$ and $\theta'$ (2), we can also show $\Delta \vdash \theta'' : \Theta_1 \to \Theta'$.

□

## D CORRECTNESS OF TYPE INFERENCE PROOFS

### D.1 Soundness of type inference

Lemma D.1. *If* $\Delta \vdash M$ *and* $(\Theta', \theta, A) = \mathsf{infer}(\Delta, \Theta, \Gamma, M)$ *then for all* $a \in (\Theta - \mathsf{ftv}(\Gamma))$ *we have* $\theta(a) = a$.

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 $\Gamma$. □

Lemma D.2. *If* $\Delta, \Theta \vdash \Gamma$ *and* $\Delta \Vdash M$ *and and* $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$. $(\Theta', \theta, A) = \mathsf{infer}(\Delta, \Theta, \Gamma, M)$ *hold, then for all* $a \in \Theta$ *s.t.* $\theta(a) \neq a$ *we have* $\Delta, \Theta' \vdash \theta(a) : \bullet$

Proof. If $\theta(a) \neq a$ then by Lemma D.1 we have $a \in \mathsf{ftv}(\Gamma)$. Therefore, by $\Delta, \Theta \vdash \Gamma$, we have $(a : \bullet) \in \Theta$. By $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$ we then have $\Delta, \Theta' \vdash \theta(a) : \bullet$. □

Lemma D.3. *If* $\Delta \# \Theta$ *and* $(\Theta_1, \theta, A) = \mathsf{infer}(\Delta, \Theta, \Gamma, M)$ *then* $\Theta_1 \# \Delta$.

Proof. Straightforward by induction, since we only generate variables fresh for $\Delta$. □

Theorem 7. *If* $\Delta, \Theta \vdash \Gamma$ *and* $\Delta \Vdash \tilde{M}$ *and* $\mathsf{infer}(\Delta, \Theta, \Gamma, \tilde{M}) = (\Theta', \theta, \tilde{A})$ *then* $\Delta, \Theta'; \theta(\Gamma) \vdash \tilde{M} : \tilde{A}$ *and* $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$.

Proof. Per induction on structure of $\tilde{M}$. In each case, we have $\Delta, \Theta \vdash \Gamma$ **(1)**, $\Delta \Vdash \tilde{M}$ **(2)**, and $\mathsf{infer}(\Delta, \Theta, \Gamma, \tilde{M}) = (\Theta', \theta, \tilde{A})$ **(3)**. For each case, we show:

I. $\Delta, \Theta'; \theta\Gamma \vdash \tilde{M} : \tilde{A}$
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 $\tilde{A} = \Gamma(x)$, $\Theta' = \Theta$, and $\theta = \iota_{\Delta,\Theta}$, which implies $\Delta \vdash \theta : \Theta \Rightarrow \Theta'$ (II) and $\Delta, \Theta' \vdash \Gamma$. We can then derive:

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

**Case** $x$**:** By definition of infer, we have $(x : \forall \overline{a}.H) \in \Gamma$ **(4)** and $\overline{b} \# \Delta, \Theta$ **(5)** and $\tilde{A} = H[\overline{b}/\overline{a}]$. Due to $\alpha$-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 $\Delta, \Theta, \overline{b} \vdash \delta : (\overline{a} : \bullet) \Rightarrow_\star \cdot$ **(6)**. By (1), (5), and Prop. B.6 we have $\Delta, \Theta, \overline{b} \vdash \Gamma$ and derive the following:

$$\frac{x : \forall \overline{a}.H \in \Gamma\ (\text{by } (4)) \qquad \Delta, \Theta, \overline{b} \vdash \delta : (\overline{a : \bullet}) \Rightarrow \cdot\ \ (\text{by } (6))}{\Delta, \Theta, \overline{b : \star} \vdash x : \delta(H)\ (I)}\ \textsc{Var}$$

By $\Delta \vdash \iota_{\Delta,\Theta} : \Theta \Rightarrow \Theta$, we have $\Delta \vdash \mathsf{weaken}(\iota_{\Delta,\Theta}, \overline{b}) : \Theta \Rightarrow \Theta, \overline{b}$ (II).

**Case** $\lambda x.M$**:** By definition of infer, we have $a \# \Delta, \Theta$ **(7)**, which implies $a \# \mathsf{ftv}(\Gamma)$ **(8)**. Let $\theta_1 = \theta[a \to S]$ **(9)**.
By (1) and (7) we have $\Delta, \Theta, a : \bullet \vdash \Gamma, x : a$. By induction, we then have

$$\begin{aligned} &\Delta, \Theta_1; \theta_1(\Gamma, x : a) \vdash M : B \\ \equiv &\Delta, \Theta_1; \theta\Gamma, x : S \vdash M : B \quad (\text{by } (8), (9)) \end{aligned} \tag{10}$$

as well as $\Delta \vdash \theta_1 : (\Theta, a : \bullet) \Rightarrow \Theta_1$, which implies $\Delta \vdash \theta : \Theta \Rightarrow \Theta_1$ (II).
By (10) we have $\Delta, \Theta_1 \vdash \theta\Gamma$, which allows us to derive the following:

$$\frac{\Delta, \Theta_1; \theta\Gamma, x : S \vdash M : B\ \ (\text{by } (10))}{\Delta, \Theta_1; \theta\Gamma \vdash \lambda x.M : S \to B\ (I)}\ \textsc{Lam}$$

**Case** $\lambda(x : A).M$**:** By (2) we have $\Delta, \Theta \vdash A$. Together with (1) this yields $\Delta, \Theta \vdash \Gamma, x : A$. Induction then yields $\Delta, \Theta_1; \theta(\Gamma, x : A) \vdash M : B$ **(11)** and $\Delta \vdash \theta : \Theta \Rightarrow \Theta_1$ (II).

By $\Delta \# \Theta$ and (2) we have $\mathsf{ftv}(A) \# \Theta$, which implies $\theta(A) = A$ **(12)**. The former implies $\Delta, \Theta_1 \vdash \theta\Gamma$ and we can derive

$$\frac{\Delta, \Theta_1; \theta\Gamma, x : A \vdash M : B \quad (\text{by } (11), (12))}{\Delta, \Theta_1; \theta\Gamma \vdash \lambda(x : A).M : A \to B \ (I)} \textsc{Lam-Ascribe}$$

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

$$(\Theta_1, \theta_1, H) = \mathsf{infer}(\Delta, \Theta, \Gamma, M) \quad \textbf{(13)}$$
$$(\Theta_2, \theta_2, A) = \mathsf{infer}(\Delta, \Theta_1, \theta_1\Gamma, N) \quad \textbf{(14)}$$

By induction, (13) yields $\Delta, \Theta_1; \theta_1\Gamma \vdash M : H$ **(15)** and $\Delta \vdash \theta_1 : \Theta \Rightarrow \Theta_1$ **(16)**.
By (15) we have $\Delta, \Theta_1 \vdash \theta_1\Gamma$. Therefore, by induction, (14) yields $\Delta, \Theta_2; \theta_2\theta_1\Gamma \vdash N : A$ **(17)** and $\Delta \vdash \theta_2 : \Theta_1 \Rightarrow \Theta_2$ **(18)**. By definition of infer, we have:

$$b \# \mathsf{ftv}(H) \quad b \# \mathsf{ftv}(A) \quad b \# \Theta \quad \textbf{(19)}$$
$$(\Theta_3, \theta_3') = \mathsf{unify}(\Delta, (\Theta_2, b : \star), \theta_2 H, A \to b) \quad \textbf{(20)}$$
$$\theta_3' = \theta_3[b \to B] \quad \textbf{(21)}$$

By (15), Lemma B.14, (18), and Lemma B.5, we have $\Delta, \Theta_2 \vdash \theta_2 H$, which implies $\Delta, \Theta_2, b : \star \vdash \theta_2 H$ by (19) and Lemma B.6. By (17) and Lemma B.14 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 5, which gives us:

$$\begin{aligned} & \theta_3'\theta_2(H) = \theta_3'(A \to b) \\ \text{implies} \quad & \theta_3\theta_2(H) = \theta_3(A) \to B \quad (\text{by } (19) \text{ and } (21)) \end{aligned} \quad \textbf{(22)}$$

and

$$\begin{aligned} & \Delta \vdash \theta_3' : (\Theta_2, b : \star) \Rightarrow \Theta_3 \\ \text{implies} \quad & \Delta \vdash \theta_3 : \Theta_2 \Rightarrow \Theta_3 \quad (\text{by } (21) \text{ and Lemma B.3}) \end{aligned} \quad \textbf{(23)}$$

By (18), (23), and composition, we have $\Delta \vdash \theta_3 \circ \theta_2 : \Theta_1 \Rightarrow \Theta_3$. By (15) and Lemma B.16, we then have $\Delta, \Theta_3; \theta_3\theta_2\theta_1\Gamma \vdash M : \theta_3\theta_2 H$ **(24)**. Similarly, by (23), (17), and Lemma B.16, we have $\Delta, \Theta_3; \theta_3\theta_2\theta_1\Gamma \vdash N : \theta_3 A$ **(25)**
By (16), (18), (23), and Lemma B.6, we have $\Delta \vdash \theta_3\theta_2\theta_1\Gamma$. We can then derive:

$$\frac{\Delta, \Theta_3; \theta_3\theta_2\theta_1\Gamma \vdash M : \theta_3(A) \to B \ (\text{by } (24), (22)) \qquad \Delta, \Theta_3; \theta_3\theta_2\theta_1\Gamma \vdash N : \theta_3 A \ (\text{by } (25))}{\Delta, \Theta_3; \theta_3\theta_2\theta_1\Gamma \vdash M\ N : B \ (I)} \textsc{App}$$

Finally, we show $\Delta \vdash \theta_3 \circ \theta_2 \circ \theta_1 : \Theta \Rightarrow \Theta_3$. It follows from (16), (18), (23), and composition (II).

**Case let** $x = M$ **in** $N$**:** We consider two cases, depending on whether $M \in \mathrm{GVal}$. We show the details for the case $M \in \mathrm{GVal}$; the details for the non-generalising case are similar. By definition of infer, we have $(\Theta_1, \theta_1, A) = \mathsf{infer}(\Delta, \Theta, \Gamma, M)$. By induction, this implies $\Delta, \Theta_1; \theta_1\Gamma \vdash M : A$ **(26)** and $\Delta \vdash \theta_1 : \Theta \Rightarrow \Theta_1$.
By definition of infer, since $M \in \mathrm{GVal}$, we have

$$\begin{aligned} (\Delta'', \Delta''') &= \mathsf{gen}(\Delta', A, M) \\ &= \mathsf{gen}((\Delta, (\mathsf{ftv}(\theta_1\Theta) - \Delta)), A, M) \\ &\quad \text{where } \Delta'' = \mathsf{ftv}(A) - (\Delta, (\mathsf{ftv}(\theta_1\Theta) - \Delta)) \ = \ (\mathsf{ftv}(A) - \Delta) - \mathsf{ftv}(\theta_1\Theta) \end{aligned} \quad \textbf{(27)}$$

Next, define $\Theta_1' = \mathsf{demote}(\bullet, \Theta_1, \Delta'')$, and 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 induction, we then have $\Delta, \Theta_2; \theta_2(\theta_1(\Gamma), x : \forall\Delta''.A) \vdash N : B$ **(29)** and $\Delta \vdash \theta_2 : \Theta_1 - \Delta'' \Rightarrow \Theta_2$ **(30)**.

We have $\mathsf{ftv}(\theta_1\Gamma) \# \Delta''$. Therefore, no type variable in $\Delta''$ is part of the input to infer yielding (28). As all newly created variables are fresh, we then have $\Delta'' \# \Theta_2$ **(31)**.
We have $(\Delta'', \Delta'') = \mathsf{gen}((\Delta, \Theta_2), \theta_2 A, M)$**(32)** by Lemma B.15.
Since $\Theta_2 \# \Delta''$ we have $\theta_2(\theta_1(\Gamma), x : \forall\Delta''.A) = \theta_2(\theta_1(\Gamma)), x : \forall\Delta''.\theta_2 A$ **(33)**. By (26) and Lemma B.14 we have $\Delta, \Theta_1 \vdash A$ and therefore $\Theta_1 \subseteq \Delta''$. Thus, $\Delta, \Theta_1; \theta_1\Gamma \vdash M : A$ (cf. **??**) is equal to $\Delta, \Delta'', \Theta_1 - \Delta''; \theta_1\Gamma \vdash M : A$. By Lemma B.16 and (30) we then have $\Delta, \Delta'', \Theta_2; \theta_2\theta_1\Gamma \vdash M : \theta_2 A$ **(34)**.
Observe that since $M \in \mathsf{GVal}$ we have $((\Delta, \Theta_2), \Delta'', M, \theta_2 A) \Updownarrow \forall\Delta''.\theta_2 A$ **(35)**.
Finally we need to establish $\mathsf{principal}((\Delta, \Theta_2), \theta_2\theta_1\Gamma, M, \Delta'', \theta_2 A)$. Recall that $\mathsf{infer}(\Delta, \Theta, \Gamma, M) = (\Theta_1, \theta_1, A)$. Thus, by property (2) of principal, we know $\mathsf{principal}((\Delta, \Theta_1), \theta_1\Gamma, M, \Delta'', A)$. Moreover, since principal is closed under substitution we have $\mathsf{principal}((\Delta, \Theta_2), \theta_2\theta_1\Gamma, M, \Delta'', \theta_2 A)$ **(36)**.
We can conclude:

$$\frac{\begin{array}{c}(\Delta'', \Delta'') = \mathsf{gen}((\Delta, \Theta_2), \theta_2 A, M) \text{ (by (32))} \qquad \Delta, \Theta_2, \Delta''; \theta_2\theta_1\Gamma \vdash M : \theta_2 A \text{ (by (34))} \\ ((\Delta, \Theta_2), \Delta'', M, \theta_2 A) \Updownarrow \forall\Delta''.\theta_2 A \text{(35)} \qquad \Delta, \Theta_2; (\theta_2\theta_1\Gamma, x : \forall\Delta''.\theta_2 A) \vdash N : B \text{ (by (29), (33))} \\ \mathsf{principal}((\Delta, \Theta_2), \theta_2\theta_1\Gamma, M, \Delta'', \theta_2 A) \text{(36)}\end{array}}{\Delta, \Theta_2; \theta_2\theta_1\Gamma \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : B\ (I)}\ \textsc{Let}$$

Since $\Delta''$ consists of free variables of $A$ that are not present in $\theta_1(\Theta)$, we have $\Delta \vdash \theta_1 : \Theta \Rightarrow \Theta_1 - \Delta''$, which then gives us $\Delta \vdash \theta_2 \circ \theta_1 : \Theta \Rightarrow \Theta_2$ (II).

**Case let** $(x : A) = M$ **in** $N$ : Again there are two cases, and we focus on the case $M \in \mathsf{GVal}$. We have $\mathsf{split}(\forall\Delta'.H, M) = (\Delta', H)$ where $A = \forall\Delta'.H$, and $\Delta'$ are assumed fresh.
By (2), we have $\Delta \vdash \forall\Delta''.H$ and thus $\Delta, \Delta'' \vdash H$ **(37)** as well as $\Delta, \Delta' \Vdash M$ **(38)**.
We show that $\Delta, \Theta_1, \Delta'; \theta_1\Gamma \vdash M : A_1$ **(39)** holds. By (1) and since $\Delta' \# \Theta$, by Lemma B.6, we have $\Delta, \Delta', \Theta \vdash \Gamma$. Together with (38), we then have $\Delta, \Delta', \Theta_1; \theta_1'\Gamma \vdash M : A_1$ by induction.
By (**??**) and Lemma B.14, we have $\Delta, \Delta', \Theta_1 \vdash A_1$. Recall $\Delta, \Delta' \vdash H$ and therefore $\Delta, \Delta', \Theta_1 \vdash H$. Thus, by Theorem 5, we have $\theta_2'(A_1) = \theta_2'(H)$ **(40)** and $\Delta, \Delta' \vdash \theta_2' : \Theta_1 \Rightarrow \Theta_2$ **(41)**.
According to the assertion, we have $\mathsf{ftv}(\theta_2'\theta_1\Theta) \# \Delta'$ **(42)**. By definition of infer, we have $\theta_2 = \theta_2' \circ \theta_1$ **(43)**, yielding $\Delta, \Delta' : \theta_2 : \Theta_1 \Rightarrow \Theta_2$. **(44)** By restriction, we also have $\Delta \vdash \theta_2 : \Theta \Rightarrow \Theta_2$ **(45)**.
By (39), (41), (38), and Lemma B.16, we have $\Delta, \Delta', \Theta_2; \theta_2'\theta_1\Gamma \vdash M : \theta_2 A_1$. By (43), this is equivalent to $\Delta, \Delta', \Theta_2; \theta_2\Gamma \vdash M : \theta_2 A_1$ **(46)**.
By (37), we have $\mathsf{ftv}(H) \subseteq \Delta, \Delta'$. Since $\Delta, \Delta' \# \Theta_1$ and (44) we then have $\theta_2(H) = H$ **(47)**. This makes (46) equivalent to $\Delta, \Delta', \Theta_2; \theta_2\Gamma \vdash M : H$ **(48)**.
By definition of infer, we have $(\Theta_3, \theta_3, B) = \mathsf{infer}(\Delta, \Theta_2, (\theta_2\Gamma, x : \forall\Delta'.H), N)$.
Due to (2), we have $\Delta \Vdash N$. Therefore, by induction, we have $\Delta, \Theta_3; \theta_3(\theta_2\Gamma, x : \forall\Delta'.H) \vdash N : B$ **(49)** and $\Delta \vdash \theta_3 : \Theta_2 \Rightarrow \Theta_3$ **(50)**. By the latter, (48), (38), and Lemma B.16, we have $\Delta, \Theta_3, \Delta'; \theta_3\theta_2\Gamma \vdash M : \theta_3 H$ **(51)**.
Using the same reasoning as for (47), we obtain $\theta_3(H) = H$ **(52)**.
Clearly $\theta_3(\forall\Delta'.H) = \forall\Delta'.H$, since $\theta_3$ cannot use variables from $\Delta'$, so by (49) is equivalent to $\Delta, \Theta_3; \theta_3\theta_2\Gamma, x : \forall\Delta'.H \vdash N : B$ **(53)**.
By (45), (50), and composition, we have $\Delta \vdash \theta_3 \circ \theta_2 : \Theta \Rightarrow \Theta_3$. (II).
By (1) and Lemma B.6, we obtain $\Delta, \Theta_3 \vdash \theta_3\theta_2\Gamma$ and can derive the following:

$$\frac{\begin{array}{c}(\Delta', H) = \mathsf{split}(\forall\Delta'.H, M) \text{ (by def. infer)} \\ \Delta, \Theta_3, \Delta'; \theta_3\theta_2\Gamma \vdash M : H \text{ (by (51), (52))} \\ \Delta, \Theta_3; \theta_3\theta_2\Gamma, x : \forall\Delta'.H \vdash N : B \text{ (by (53))}\end{array}}{\Delta, \Theta_3; \theta_3\theta_2\Gamma \vdash \mathbf{let}\ (x : \forall\Delta'.H) = M\ \mathbf{in}\ N : B\ (I)}\ \textsc{Let-Ascribe}$$

□

### D.2 Completeness of type inference

Theorem 8 (Type inference is complete and principal). *If* $\Delta \Vdash \tilde{M}$ *and* $\Delta \vdash \tilde{\theta} : \Theta \Rightarrow \Theta'$ *and* $\Delta, \Theta'; \tilde{\theta}(\Gamma) \vdash \tilde{M} : \tilde{A}$*, then* $\mathsf{infer}(\Delta, \Theta, \Gamma, \tilde{M}) = (\Theta'', \theta', A_R)$ *where there exists* $\theta''$ *satisfying* $\Delta \vdash \theta'' : \Theta'' \Rightarrow \Theta'$ *such that* $\tilde{\theta} = \theta'' \circ \theta'$ *and* $\theta''(A_R) = \tilde{A}$*. FE: Again, restated using different meta vars*

Proof. FE: TODO: Double check that all $\Gamma$ are well-formed w.r.t. new monomorphism restriction.

Per induction on structure of $\tilde{M}$. In each case, we assume $\Delta \Vdash \tilde{M}$ **(1)**, and $\Delta \vdash \tilde{\theta} : \Theta \Rightarrow \Theta'$ **(2)**, and $\Delta, \Theta'; \theta\Gamma \vdash \tilde{M} : \tilde{A}$ **(3)**, which implies $\Delta, \Theta' \vdash \theta\Gamma$ **(4)**, and $\Delta, \Theta' \vdash \tilde{A}$ **(5)**. For each case, we show:

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

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

**Case** $\lceil x \rceil$**:** By (3) and Freeze, we have $(x : \tilde{A}) \in \theta\Gamma$. infer succeeds (I), and we have $\Theta'' = \Theta$, $\theta' = \iota_{\Delta,\Theta}$, and $A_R = \Gamma(x)$. The latter implies $\tilde{A} = \tilde{\theta}(A_R)$.
We have $\Delta \vdash \theta' : \Theta \Rightarrow \Theta$. Let $\theta'' := \tilde{\theta}$. By (2) we then have $\Delta \vdash \theta'' : \Theta \Rightarrow \Theta'$ (II). We have $\tilde{\theta} = \theta'' = \theta'' \circ \iota_{\Delta,\Theta} = \theta'' \circ \theta'$ (III).
We have $\theta''(A_R) = \tilde{\theta}(A_R) = \tilde{A}$ (IV).

**Case** $x$**:** By (3) and Var, we have $\tilde{A} = \delta\tilde{\theta}H$ **(6)** and $\tilde{\theta}\Gamma(x) = \tilde{\theta}(\forall\Delta'.H) = \forall\Delta'.\tilde{\theta}H$ **(7)** for some $\delta$ with $\Delta, \Theta' \vdash \delta : \Delta' \Rightarrow_\star \cdot$ **(8)**. Here, by $\alpha$-equivalence we assume $\Delta' \# \Delta, \Theta$ **(9)** and $\Delta' \# \Delta, \Theta'$. By (7), we have $\Gamma(x) = \forall\Delta'.H$. Thus, infer succeeds (I) with $\Theta'' = (\Theta, \overline{b : \star})$, and $\theta' = \mathsf{weaken}(\iota_{\Delta,\Theta}, \overline{b : \star})$ **(10)**, and $A_R = H[\overline{b}/\overline{a}]$ **(11)**. We also have $\overline{a} = \Delta'$, and $\overline{b} \# \Theta$, and $\overline{b} \# \mathsf{ftv}(H)$. Clearly, we also have $\Delta \vdash \theta' : \Theta \Rightarrow \Theta''$ **(12)**.
Let $\overline{a} = a_1, \ldots, a_n$ and $\overline{b} = b_1, \ldots, b_n$ for some $n \geq 0$. We define $\theta''$ such that

$$\theta''(c) = \begin{cases} \tilde{\theta}(c) & \text{if } c \in \Theta \\ \delta(a_i) & \text{if } c = b_i \text{ for some } b_i \in \overline{b} \end{cases} \tag{13}$$

By (2) and (13), for all $(c : K) \in \Theta$ we have $\Delta, \Theta' \vdash \tilde{\theta}(c) : K$ **(14)**. By (8), we have $\Delta, \Theta' \vdash \delta(a) : \star$ for all $a \in \Delta'$ and thus $\Delta \vdash \theta''(b) : \star$ for all $b \in \overline{b}$. Together, we then have $\Delta \vdash \theta'' : \Theta'' \Rightarrow \Theta'$ (II).
By (12) and (II), $\theta'' \circ \theta'$ is well-formed.
By (10) and (13), we have $\theta''\theta'(c) = \theta''(c) = \tilde{\theta}(c)$ for all $c \in \Theta$ (III).
By (4) and (7), we have $\mathsf{ftv}(\tilde{\theta}(\forall\Delta'.H)) = \mathsf{ftv}(\forall\Delta'.\tilde{\theta}H) \subseteq \Delta, \Theta'$ and therefore also $\mathsf{ftv}(\tilde{\theta}H) \subseteq \Delta, \Delta', \Theta'$ **(15)**. Assume $c \in \mathsf{ftv}(H) \setminus \Delta, \Delta', \Theta$, which by (2) implies $\tilde{\theta}(c) = c$ and therefore $c \in \mathsf{ftv}(\tilde{\theta}H)$. This contradicts (15), and we therefore have $\mathsf{ftv}(H) \subseteq \Delta, \Delta', \Theta$ **(16)**. FE: We may want to turn this into a lemma. Do we assume that $\tilde{\theta}$ is applicable to any type and is just the identity on anything other than $\Delta, \Theta$, or do we assume that $\tilde{\theta}$ is only applicable to types $A$ such that $\Delta, \Theta \vdash A$?
We show that for all $c \in \mathsf{ftv}(H)$, we have $\theta''(c[\overline{b}/\overline{a}]) = \delta\tilde{\theta}(c)$ **(17)**. According to (16), we distinguish three cases:

(1) Let $a_i \in \Delta'$. We then have

$$\begin{array}{rrcll} & \delta(a_i) & = & \delta(a_i) & \\ \text{equiv.} & \theta''(a_i[\overline{b/a}]) & = & \theta_H(a_i) & \text{(by (13): } \theta''(b_i) = \delta(a_i)) \\ \text{equiv.} & \theta''(a_i[\overline{b/a}]) & = & \delta\tilde{\theta}(a_i) & \text{(by (9), (2): } \tilde{\theta}(a_i) = a_i \text{ )} \end{array}$$

(2) Let $c \in \Theta$ We then have

$$\begin{array}{rrcll} & \tilde{\theta}(c) & = & \tilde{\theta}(c) & \\ \text{equiv.} & \tilde{\theta}(c[\overline{b/a}]) & = & \tilde{\theta}(c) & \text{(by (9))} \\ \text{equiv.} & \theta''(c[\overline{b/a}]) & = & \delta\tilde{\theta}(c) & \text{(by (13))} \end{array}$$

(3) Let $c \in \Delta$ We then have

$$\begin{array}{rcl} & \delta\tilde{\theta}(c) & \\ = & \delta(c) & \text{(by (2))} \\ = & c & \text{(by (8))} \\ = & c[\overline{b}/\overline{a}] & \text{(by (9))} \\ = & \theta''(c[\overline{b}/\overline{a}]) & \text{(by (II))} \end{array}$$

By (11) and (6), (17) then yields $\theta''(A_R) = \tilde{A}$. By (5), we have shown (IV).

**Case** $\lambda x.M$**:** By (3) and Lam, we have $\tilde{A} = S' \to B'$ for some $S', B'$ as well as $\Delta, \Theta'; \tilde{\theta}\Gamma, (x : S') \vdash M : B'$ **(18)**. The latter implies $\Delta, \Theta' \vdash S'$ **(19)**.
Let $a$ be the fresh variable as in the definition of infer; in particular $a \# \Theta$ **(20)**. Let $\tilde{\theta}_a(b) = \tilde{\theta}(b)$ for all $b \in \Theta$ **(21)** and $\tilde{\theta}_a(a) = S'$ **(22)**. By (2) and (19), we have $\Delta \vdash \tilde{\theta}_a : (\Theta, a : \bullet) \Rightarrow \Theta'$. This definition makes (18) equivalent to $\Delta, \Theta'; \tilde{\theta}_a(\Gamma, x : a) \vdash M : B'$.
By induction, we therefore have have the following:

- infer$(\Delta, (\Theta, a : \bullet), (\Gamma, x : a), M)$ succeeds **(23)**, returning $(\Theta_1, \theta_1', B)$
- There exists $\theta_1''$ such that
  - $\Delta \vdash \theta_1'' : \Theta_1 \Rightarrow \Theta'$ **(24)**
  - $\tilde{\theta}_a = \theta_1'' \circ \theta_1'$ **(25)**
  - $\Delta, \Theta' \vdash \theta_1''(B) = B'$ **(26)**

By (24) and (24), we have $\Delta \vdash \theta_1' : (\Theta, a : \bullet) \Rightarrow \Theta_1$. By preservation of kinds under substitution, we have $\Delta, \Theta_1 \vdash \theta_1'(a) : \bullet$. Thus, $\theta_1' = \theta[a \mapsto S]$ **(27)** is well-defined, yielding a substitution $\Delta \vdash \theta : \Theta \Rightarrow \Theta_1$. Together with (23), this means that infer succeeds (I).
According to the return values of infer, we have $A_R = S \to B$, $\Theta'' = \Theta_1$, and $\theta' = \theta$ **(28)**.
Let $\theta''$ be defined as $\theta_1''$ **(29)**. By (24), this choice immediately satisfies (II).
We show (III) as follows: Let $b \in \Theta$. We then have

$$\begin{array}{rcll} & \tilde{\theta}(b) & & \\ = & \tilde{\theta}_a(b) & \text{(by (21), (20))} & \\ = & \theta_I''\theta_I'(b) & \text{(by (24))} & \\ = & \theta_I'\theta(b) & \text{(by (20), (27))} & \\ = & \theta''\theta'(b) & \text{(by (28), (29))} & \end{array}$$

By (27), we have $\theta_1'(a) = S$. By (22), we have $\tilde{\theta}_a(a) = S'$. By (25) we therefore have $\theta_1''(S) = \tilde{\theta}_a(a) = S$. Together with (26), $\tilde{A} = S \to B$, and $A_R = S \to B$ we have shown ( IV).

**Case**$\lambda(x : A).M$**:** Analogous to previous case. FE: TODO, should be simpler than previous case

**Case** $M\ N$**:** By (3) and App, we have $\Delta, \Theta'; \tilde{\theta}\Gamma \vdash M : A_N \to \tilde{A}$ and $\Delta, \Theta'; \tilde{\theta}\Gamma \vdash N : A_N$ **(30)** for some type $A_N$.

By induction, $\mathsf{infer}(\Delta, \Theta, \Gamma, M)$ succeeds, returning $(\Theta_1, \theta_1, A'_I)$ and there exists $\theta''_1$ such that the following conditions holds:
- $\Delta \vdash \theta''_1 : \Theta_1 \Rightarrow \Theta'$ **(31)**
- $\tilde{\theta} = \theta''_1 \circ \theta_1$ **(32)**
- $\Delta, \Theta' \vdash \theta''_1(A'_I) = A_N \to \tilde{A}$ **(33)**

By (33), $A'_I$ must not have toplevel quantifiers. This makes setting $H := A'_I$ in the algorithm well-defined. Let $B_N$ and $B_M$ such that $H = B_N \to B_M$ **(34)**. By (33), we have $\theta''_1(B_N) = A_N$ **(35)** and $\theta''_1(B_M) = \tilde{A}$ **(36)**.
By (31) and (32), we have $\Delta \vdash \theta_1 : \Theta \Rightarrow \Theta_1$ **(37)**. By the choice of $b$, we have $b \# \Delta$, and $b \# \Theta_1$, and $b \# \Theta_2$ and $b \# \Theta'$ **(38)**
By (32), we can rewrite (30) as $\Delta, \Theta'; \theta''_1\theta_1\Gamma \vdash N : A_N$. By induction, we then have that $\mathsf{infer}(\Delta, \Theta_1, \theta_1\Gamma, N)$ succeeds and returns $(\Theta_2, \theta_2, A)$. In addition, there exists $\theta''_2$ such that
- $\Delta \vdash \theta''_2 : \Theta_2 \Rightarrow \Theta'$ **(39)**
- $\theta''_1 = \theta''_2 \circ \theta_2$ **(40)**
- $\Delta, \Theta' \vdash \theta''_2(A) = A_N$ **(41)**

By (39) and (40), $\Delta \vdash \theta_2 : \Theta_1 \Rightarrow \Theta_2$ holds **(42)**.
Let $\theta_b$ be defined such that

$$\theta_b(c) = \begin{cases} \theta''_2(c) & \text{if } c \in \Theta_2 \\ \theta''_2\theta_2(B_M) & \text{if } c = b \end{cases} \quad \textbf{(43)}$$

FE: By TODO We have $\Delta \vdash \theta_b : (\Theta_2, b : \star) \Rightarrow \Theta'$.
We have

$$\begin{array}{lrcll} & A_N \to \tilde{A} & = & A_N \to \tilde{A} & \\ \text{implies} & \theta''_I(B_N) \to \theta''_I(B_M) & = & \theta''_2(A) \to \theta''_I(B_M) & \text{(by (35), (36), (41))} \\ \text{implies} & \theta''_2\theta_2(B_N) \to \theta''_2\theta_2(B_M) & = & \theta''_2(A) \to \theta''_2\theta_2(B_M) & \text{(by (40))} \\ \text{implies} & \theta_b\theta_2(B_N) \to \theta_b\theta_2(B_M) & = & \theta_b(A) \to \theta_b(b) & \text{(by (43), (38))} \\ \text{implies} & \theta_b\theta_2(H) & = & \theta_b(A \to b) \textbf{ (44)} & \text{(by (34))} \end{array}$$

Note that the reasoning above also shows $\theta_b(b) = \tilde{A}$. By (42) and FE: Need to show that the input types are well-kinded? Theorem 6, $\mathsf{unify}(\Delta, (\Theta_2, b : \star), \theta_2(H), A \to b)$ succeeds, returning $(\Theta_3, \theta'_3)$, and there exists $\theta''_3$ such that $\Delta \vdash \theta''_3 : \Theta_3 \Rightarrow \Theta'$ **(45)** and $\theta_b = \theta''_3 \circ \theta'_3$ **(46)**. The latter implies $\Delta \vdash \theta'_3 : (\Theta_2, b : \star) \Rightarrow \Theta_3$. This makes requiring $\theta'_3 = \theta_3[b \to B]$ **(47)** well-defined, resulting in $\Delta \vdash \theta_3 : \Theta_2 \Rightarrow \Theta_3$ **(48)**.
By (37), (42), (48), 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 succeeds (I), and it returns $(\Theta'', \theta', A_R) = (\Theta_3, \theta_3 \circ \theta_2 \circ \theta_1, B)$ **(49)**.
Let $\theta''$ be defined as $\theta''_3$, satisfying (II), by (45).
By (32), (40), and (46), we have $\tilde{\theta} = \theta''_3 \circ (\theta_3 \circ \theta_2 \circ \theta_1) = \theta'' \circ \theta'$ (III).
We show (IV):

$$\begin{array}{lll} & \theta''(A_R) & \\ = & \theta''_3(B) & (\text{by } \theta'' := \theta''_U, A_R := B) \\ = & \theta''_3\theta_3(b) & \text{(by (47))} \\ = & \theta_b(b) & \text{(by (46))} \\ = & \tilde{A} & \end{array}$$

**Case let $x = M$ in $N$:** By (3) and LET, there exist $A'$, $A_x$, and $\Delta_G$ such that

$$\Delta_G = \mathsf{ftv}(A') - (\Delta, \Theta') \tag{50}$$
$$\Delta, \Theta', \Delta_G; \tilde{\theta}\Gamma \vdash M : A' \tag{51}$$
$$((\Delta, \Theta'), \Delta_G, M, A') \Updownarrow A_x \tag{52}$$
$$\Delta, \Theta'; \tilde{\theta}\Gamma, x : A_x \vdash N : \tilde{A} \tag{53}$$
$$\mathsf{principal}((\Delta, \Theta'), \tilde{\theta}\Gamma, \Delta_G, A') \tag{54}$$

By (2) and FE: needs $\Delta_G \# \Theta'$ weakening, we have $\Delta \vdash \tilde{\theta} : \Theta \Rightarrow \Theta', \Delta_G$. Together with (51) we then have that $\mathsf{infer}(\Delta, \Theta, \Gamma, M)$ succeeds, returning $(\Theta_1, \theta_1, A)$, and there exists $\theta_1''$ such that

$$\Delta \vdash \theta_1'' : \Theta_1 \Rightarrow (\Theta', \Delta_G) \tag{55}$$
$$\tilde{\theta} = \theta_1'' \circ \theta_1 \tag{56}$$
$$\theta_1''(A) = A' \tag{57}$$

By (2), (55) and (56), we have $\Delta \vdash \theta_1 : \Theta \Rightarrow \Theta_1$ **(58)**.
Let $\Theta_{\theta_1} = \mathsf{ftv}(\theta_1) - \Delta$. Note that FE: by TODO, this yields $\Theta_{\theta_1} \subseteq \Theta_1$ **(59)** and $\Delta''' \# \Theta_{\theta_1}$ and $\Delta'' \# \Theta_{\theta_1}$. By (2), (55) and (56) we have $\Delta, \Theta' \vdash \theta_1''(a) : K$ for all $(a : K) \in \Theta_{\theta_1}$ **(60)**.
By definition of infer, we have $(\Theta_1, \theta_1, A) = \mathsf{infer}(\Delta, \Theta, \Gamma, M)$. By (1) and (4), theorem 7, we therefore have $\Delta, \Theta_1; \theta_1\Gamma \vdash M : A$ **(61)**, which implies $\Delta, \Theta_1 \vdash A$ **(62)**
By (2), (50), (54), (55) to (57) and (62), we can apply Lemma B.10, yielding $\theta_1''(\Delta''') = \Delta_G$ **(63)**.
By FE: TODO $\Delta'' \# \Theta_{\theta_1}$, we can strengthen (58) to $\Delta \vdash \theta_1 : \Theta \Rightarrow \Theta_1 - \Delta''$ **(64)**.
We distinguish two cases based on the shape of $M$. In each case we show that there exists $\theta_N''$ such that $\Delta \vdash \theta_N'' : (\Theta_1' - \Delta'') \Rightarrow \Theta'$ **(65)** and $\Delta, \Theta'; \theta_N''(\theta_1(\Gamma), x : \forall \Delta''.A) \vdash N : \tilde{A}$ **(66)** and $\tilde{\theta} = \theta_N'' \circ \theta_1$ **(67)** hold.

**Subcase 1,** $M \in \mathsf{GVal}$**:** We have $\Delta'' = \Delta'''$. By (52), 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 $\theta_N''$ 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 $\Delta, \Theta' \vdash A_D : \bullet$. By $\Delta'' \# \Theta_{\theta_1}$ and $\Delta'' \subseteq \Theta_1$ and $\Theta_{\theta_1} \subseteq \Theta_1$, this definition is well-formed.
By $\Delta'' = \Delta''' = \mathsf{ftv}(A) - \Delta - \Theta_{\theta_1}$ we have $\theta_N''(c) = \theta''(c)$ for all $c \in \mathsf{ftv}(A) - \Delta''$ **(68)**.
Together with (60) 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 (56) and (64) and $\theta_N''(c) = \theta_1''(c)$ for all $c \in \Theta_{\theta_1}$ we also have $\tilde{\theta} = \theta_N'' \circ \theta_1$.
We have

$$\begin{array}{lll} = & \theta_N''(\forall \Delta''.A) & \\ = & \theta_N''(\forall \Delta_G.A[\Delta_G/\Delta'']) & \\ = & \forall \Delta_G.\theta_N''(A[\Delta_G/\Delta'']) & (\text{by } \mathsf{ftv}(\theta_N'') \subseteq \Delta, \Theta' \text{ and } \Delta, \Theta' \# \Delta_G) \\ = & \forall \Delta_G.\theta''(A) & (\text{by } \Delta'' = \Delta''' \text{ and (63) and (68)}) \\ = & A_x & (\text{by } A_x = \forall \Delta_G.A' \text{ and (57)}) \end{array}$$

Thus, (53) is equivalent to $\Delta, \Theta'; \theta_N''((\theta_1\Gamma), x : \forall \Delta''.A) \vdash N : \tilde{A}$.

**Subcase 2,** $M \notin \mathsf{GVal}$**:** We have $\Delta'' = \cdot$. By (52), we have $A_x = \delta(A')$ for some $\delta$ with $\Delta, \Theta' \vdash \delta : \Delta_G \Rightarrow_\bullet \cdot$ **(69)**.

Let $\theta''_N$ be defined as follows for all $c \in \Theta_1 - \Delta'' = \Theta_1$:

$$\theta''_N(c) = \begin{cases} \theta''_1(c) & \text{if } c \in \Theta_{\theta_1} \\ A_D & \text{if } c \in \Theta_1 - \Delta''' - \Theta_{\theta_1} \\ \delta(\theta''_1(c)) & \text{if } c \in \Delta''' \end{cases} \quad \textbf{(70)}$$

Where $A_D$ is defined as before.
By $\Delta'' \# \Theta_{\theta_1}$ and $\Delta''' \subseteq \Theta_1$ and $\Theta_{\theta_1} \subseteq \Theta_1$, the first two cases are well-formed. By (63) and (69), the third case is well-formed.
Using (60), we have that $\Delta, \Theta' \vdash \theta''_N(c) : K$ for all $(c : K) \in \Theta_{\theta_1}$. By (69), 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 B.11, 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 (58), (56), and $\Delta'' = \cdot$ we then have $\tilde{\theta} = \theta''_N \circ \theta_1$.
We have

$$\begin{array}{lll} & \theta''_N(\forall \Delta''.A) & \\ = & \theta''_N(A) & \text{(by } \Delta'' = \cdot\text{)} \\ = & \theta''_1(A)[\delta(\Delta_G)/\Delta_G] & \text{(by (63) and (70))} \\ = & A'[\delta(\Delta_G)/\Delta_G] & \text{(by (57))} \\ = & \delta(A') & \\ = & A_x & \end{array}$$

We have shown that in each case, (65), (66), and (67) hold.
Thus, by induction, we have that $\text{infer}(\Delta, \Theta'_1 - \Delta'', \theta_1\Gamma, N)$ succeeds, returning $(\Theta_2, \theta_2, B)$, and there exists $\theta''_2$ such that

$$\Delta \vdash \theta''_2 : \Theta_2 \Rightarrow \Theta' \quad \textbf{(71)}$$
$$\theta''_N = \theta''_2 \circ \theta_2 \quad \textbf{(72)}$$
$$\theta''_2(B) = \tilde{A} \quad \textbf{(73)}$$

By (58) and $\Delta'' \# \text{ftv}(\theta_1)$ we have that $\Delta \vdash \theta_1 : \Theta \Rightarrow \Theta - \Delta''$ holds.
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 (71), this choice immediately satisfies (II).
We have

$$\begin{array}{lll} & \tilde{\theta} & \\ = & \theta''_N \circ \theta_1 & \text{(by (67))} \\ = & \theta''_2 \circ \theta_2 \circ \theta_1 & \text{(by (72))} \end{array}$$

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

$$\begin{array}{lll} & \tilde{A} & \\ = & \theta''_2(B) & \text{(by (73))} \\ = & \theta''(B) & \text{(by } \theta'' := \theta''_2\text{)} \end{array}$$

**Case let** $(x : A) = M$ **in** $N$ By (3) and Let-Ascribe, there exist $\Delta_G$ and $A_M$ such that we have

- $\Delta_G = \text{gen}^\star(A, M)$
- $\Delta, \Theta', \Delta_G; \tilde{\theta}\Gamma \vdash M : A_M$ **(74)**
- $A = \forall \Delta_G.A_M$ **(75)**
- $\Delta, \Theta'; \tilde{\theta}(\Gamma), (x : A) \vdash N : \tilde{A}$ **(76)**

Note that by definition of infer, we have $\Delta_G = \Delta'$. By (74), we have $\Delta' \# \Theta'$ **(77)**. By (2) and Lemma B.9, FE: This also requires $\Delta' \# \Theta$ we then have $\Delta, \Delta' \vdash \tilde{\theta} : \Theta \Rightarrow \Theta'$.
Together with (74) we then have the following by induction: infer$((\Delta, \Delta'), \Theta, \Gamma, M)$ succeeds, returning $(\Theta_1, \theta_1, A_1)$ and there exists $\theta_1''$ such that

- $\Delta, \Delta' \vdash \theta_1'' : \Theta_1 \Rightarrow \Theta'$ **(78)**
- $\tilde{\theta} = \theta_1'' \circ \theta_1$ **(79)**
- $\theta_1'' A_1 = A_M$ **(80)**

Let $\Delta''$ and $H$ such that $A = \forall \Delta''.H$. We show that $A_M = A'$ **(81)** holds. Consider the case $M \in \text{GVal}$: By definition of split, we have $\Delta' = \Delta''$. and $A' = H$. By (75), we have $A_R = H$.
Conversely, consider the case $M \notin \text{GVal}$: By definition of split, we have $\Delta' = \cdot$. and $A' = A$. By (75), we have $A_M = A$.
We then have

$$
\begin{array}{lll}
 & \theta_1'' A_1 & \\
= & A_M & \text{(by (80))} \\
= & A' & \text{(by (81))} \\
= & \theta_1'' A' & \text{(This only holds if we have } \text{ftv}(A) \# \Theta_1 \text{ (e.g., by } \Delta \Vdash \tilde{M}\text{))}
\end{array}
$$

By the above equality and (78), we have FE: Show $\Delta, \Delta', \Theta \vdash A_1 : K$ and $\Delta, \Delta', \Theta \vdash A' : K$ the following according to Theorem 6: unify$((\Delta, \Delta'), \Theta_1, A', A_1)$ succeeds, returning $(\Theta_2, \theta_2')$, and there exists $\theta_2''$ such that $\Delta, \Delta' \vdash \theta_2'' : \Theta_2 \Rightarrow \Theta'$ **(82)** and $\theta_1'' = \theta_2'' \circ \theta_2'$ **(83)**. The latter implies $\Delta, \Delta' \vdash \theta_2' : \Theta_1 \Rightarrow \Theta_2$ **(84)** and $\Delta' \# \Delta, \Theta_2$ **(85)**.
By (79), we have $\Delta, \Delta' \vdash \theta_1 : \Theta \Rightarrow \Theta_1$. Together with (90), this makes $\theta_2' \circ \theta_1$ well-defined **(86)**.
By (79) and (83), we have $\tilde{\theta} = \theta_2'' \circ \theta_2' \circ \theta_1$ **(87)**. By (2), we have that for all $a \in \Theta$, we have $\text{ftv}(\tilde{\theta}(a)) \subseteq \Delta, \Theta'$ **(88)**.
Let $a \in \Theta$ and let $b \in \text{ftv}(\theta_2' \theta_1(a))$. Assume $b \in \Delta'$. By (85), we have $\theta_2''(b) = b$ and therefore $b \in \text{ftv}(\theta_2'' \theta_2' \theta_1(a))$ which by (87) implies $b \in \text{ftv}(\tilde{\theta}(a))$. By (85), this contradicts (88). We have therefore shown that for all $a \in \Theta$, we have $\text{ftv}(\theta_2' \theta_1(a)) \# \Delta'$. Together with (86), this makes $\theta_2 = (\theta_2' \circ \theta_1) \setminus \Delta''$ **(89)** well-defined, yielding $\Delta \vdash \theta_2 : \Theta \Rightarrow \Theta_2$ **(90)**.
By FE: TODO $\text{ftv}(A) \subseteq \Delta$, and (82), we have $\theta_2''(A) = A$ **(91)**.
We have

$$
\begin{array}{lll}
 & \Delta, \Theta'; \tilde{\theta}\Gamma, x : A \vdash N : \tilde{A} & \text{(by (76))} \\
\text{implies} & \Delta, \Theta'; \theta_2'' \theta_2 \Gamma, x : A \vdash N : \tilde{A} & \text{(by (87), (89))} \\
\text{implies} & \Delta, \Theta'; \theta_2''(\theta_2(\Gamma), x : A) \vdash N : \tilde{A} & \text{(by (91))}
\end{array}
$$

By induction, the judgement above and (82) shows that infer$(\Delta, \Theta_2, (\theta_\Gamma, x : A), N)$ succeeds, returning $(\Theta_3, \theta_3, B)$ and there exists $\theta_3''$ such that

- $\Delta \vdash \theta_3'' : \Theta_3 \Rightarrow \Theta'$ **(92)**
- $\theta_2'' = \theta_3'' \circ \theta_3$ **(93)**
- $\theta_3''(B) = \tilde{A}$ **(94)**

By (93), we have $\theta_3 : \Theta_2 \Rightarrow \Theta_3$. Thus, by (90), $\theta_3 \circ \theta_2$ is well-defined, yielding $\Delta \vdash \theta_3 \circ \theta_2 : \Theta \Rightarrow \Theta_3$ **(95)**.
We have shown that all steps of the algorithm succeed (I). 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 (92), this choice immediately satisfies (II) (95), (III)

We show (III):

$$
\begin{array}{lll}
 & \tilde{\theta} & \\
= & \theta_2'' \circ \theta_2' \circ \theta_1 & \text{(by (87))} \\
= & \theta_2'' \circ \theta_2 & \text{(by (89))} \\
= & \theta_3'' \circ \theta_3 \circ \theta_2 & \text{(by (93))} \\
= & \theta'' \circ \theta' & \text{(by } \theta' := \theta_3 \circ \theta_2, \theta'' := \theta_3'' \text{)}
\end{array}
$$

FE: One could nit-pick that we do not show that all of the compositions above are well-defined because we switch between $\Delta$ and $\Delta, \Delta'$

By $\theta'' = \theta_3''$, (94) yields (IV).

□