Principal Type Inference under a Prefix

A Fresh Look at Static Overloading

DAAN LEIJEN, Microsoft Research, USA
WENJIA YE, National University of Singapore and The University of Hong Kong, Singapore and China

At the heart of the Damas-Hindley-Milner (HM) type system lies the abstraction rule which derives a function type for a lambda expression. In this rule, the type of the parameter can be “guessed”, and can be any type that fits the derivation. The beauty of the HM system is that there always exists a most general type that encompasses all possible derivations – Algorithm W is used to infer these most general types in practice.

Unfortunately, this property is also the bane of the HM type rules. Many languages extend HM typing with additional features which often require complex side conditions to the type rules to maintain principal types. For example, various type systems for impredicative type inference, like HMF, FreezeML, or Boxy types, require let-bindings to always assign most general types. Such a restriction is difficult to specify as a logical deduction rule though, as it ranges over all possible derivations. Despite these complications, the actual implementations of various type inference algorithms are usually straightforward extensions of algorithm W, and from an implementation perspective, much of the complexity of various type system extensions, like boxes or polymorphic weights, is in some sense artificial.

In this article we rephrase the HM type rules as *type inference under a prefix*, called HMQ. HMQ is sound and complete with respect to the HM type rules, but always derives principal types that correspond to the types inferred by algorithm W. The HMQ type rules are close to the clarity of the declarative HM type rules, but also specific enough to “read off” an inference algorithm, and can form an excellent basis to describe type system extensions in practice. We show in particular how to describe the FreezeML and HMF systems in terms of inference under a prefix, and how we no longer require complex side conditions. We also show a novel formalization of static overloading in HMQ as implemented in Koka language.

CCS Concepts: • **Software and its engineering** → *Functional languages*; • **Theory of computation** → **Type theory**.

Additional Key Words and Phrases: Type Inference, Damas-Hindley-Milner, Static Overloading

ACM Reference Format:
Daan Leijen and Wenjia Ye. 2025. Principal Type Inference under a Prefix: A Fresh Look at Static Overloading. *Proc. ACM Program. Lang.* 9, PLDI, Article 205 (June 2025), 24 pages. https://doi.org/10.1145/3729308

1 Introduction

At the heart of Damas-Hindley-Milner style type inference [Damas and Milner 1982; Hindley 1969; Milner 1978] lies the abstraction rule which infers a type $\tau_1 \to \tau_2$ for a lambda expression $\lambda x.e$ under a type environment Γ:

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}\,\text{FUN}$$

Authors’ Contact Information: Daan Leijen, Microsoft Research, Redmond, WA, USA, daan@microsoft.com; Wenjia Ye, National University of Singapore and The University of Hong Kong, Singapore and China, yewenjia@outlook.com.

ACM 2475-1421/2025/6-ART205
https://doi.org/10.1145/3729308

Interestingly, the type τ_1 of the parameter x occurs free and is "guessed" – it can be any type that fits the derivation. This encompasses both the *beauty*, but also the *bane*, of the Damas-Hindley-Milner (HM) type rules.

For example, for the identity function $\lambda x.\ x$ we can derive many types, like $int \rightarrow int$, or $bool \rightarrow bool$ etc. That seems a problem at first, but the beauty of the HM type rules is that there always exists a derivation with a most general type of which all other possible derivations are an instance – in the identity case the type $\forall\alpha.\alpha \rightarrow \alpha$. Moreover, there exist an algorithm W that always infers these most general types which is widely used in practice for HM style type inference.

Nevertheless, this rule is also the bane of HM type inference. In practice many languages extend HM typing with various extensions and it turns out that the inference rules need to be restricted in often complicated ways. For example, Leijen [2008] describes the HMF system that allows for impredicative higher-ranked types. He gives the following example:

$\mathsf{let}\ wrapl\ x\ y\ =\ [y]\ \mathsf{in}\ wrapl\ ids\ id$

where *ids* has the impredicative type $[\forall\alpha.\alpha \rightarrow \alpha]$ (i.e. a list of polymorphic identity functions). If *wrapl* is given its most general type, namely $\forall\alpha\beta.\ \alpha \rightarrow \beta \rightarrow [\beta]$, we can derive the type $\forall\alpha.[\alpha \rightarrow \alpha]$ for the body. However, if we use the [FUN] rule to "guess" a less general type for *wrapl*, namely $\forall\alpha.[\alpha] \rightarrow \alpha \rightarrow [\alpha]$, then we can derive (in HMF) the type $[\forall\alpha.\alpha \rightarrow \alpha]$ for the body (as the shared α now matches with the polymorphic identity type). Unfortunately, these types are incomparable – neither is an instance of the other – and we lose principal type derivations. To fix this issue, the HMF system includes a side-condition on the let-rule to always assign most general types:

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \sigma_1 \quad \Gamma, x : \sigma_1 \vdash e_2 : \sigma_2 \\ \forall\sigma.\ \Gamma \vdash e_1 : \sigma \Rightarrow \sigma_1 \sqsubseteq \sigma\end{array}}{\Gamma \vdash \mathsf{let}\ x\ =\ e_1\ \mathsf{in}\ e_2 : \sigma_2}\ \textsc{hmf-let}$$

From a logical perspective this condition is quite unsatisfactory. It is no a longer a natural deduction rule since the condition ranges negatively over *all* possible derivations, making it more difficult to reason about. In another more recent example, Emrich et al. [2022] describe the FreezeML system that also includes a side-condition on the let-rule that ranges over all possible derivations, and they write "*the reader may be concerned about whether the typing judgement is well-defined given that it appears in a negative position in the definition of* principal. *[...] the definition is nevertheless well founded by indexing by untyped terms or the height of the derivation tree*". Vytiniotis et al [2006,§6] also introduce a similar let rule in the context of boxy type inference, and similar ideas were also used by Leroy and Mauny [1993] for the typing of dynamics in ML, and by Garrigue and Rémy [1999,§5] in their extension of ML with semi-explicit first-class polymorphism.

An example of a novel extension that we describe in this article is *static overloading*. With static overloading, we allow a function f to be defined in different modules, say *modi* and *modb*, where we can give their fully qualified names as *modi/f* and *modb/f*. Suppose they have the types:

$modi/f\ :\ int \rightarrow int$
$modb/f\ :\ bool \rightarrow int$

The idea is now to use local type information to allow a programmer to write just f and have it be resolved to either definition. For example, f 1 would be elaborated to *modi/f* 1. Since the overloading is static, we reject expressions where the the definition cannot be resolved uniquely. For example a bare f is rejected, but we would also like to reject $\lambda x.\ f\ x$. Unfortunately, the [FUN] rule again allows us to "guess" the type *int* for x, in which case we could elaborate to $\lambda x : int.\ modi/f\ x$ – but also we could guess the type *bool* for x and derive $\lambda x : bool.\ modb/f\ x$. Again, the flexibility of the [FUN] rule causes non-principal derivations.

The interesting part of all the previous examples is that it is only difficult to extend the declarative HM type rules with the new extensions – but for all of the example systems, the changes to the actual type inference *implementation*, based on algorithm W, are usually quite straightforward! For example, all HM based type inference algorithms already infer most general types for let-bindings – as required by HMF, FreezeML, or Boxy type inference; and they will already use a general type α for the x binding in the static overloading example (and not some arbitrary type *int* or *bool*). As such, most of the complexity that we see in the type rules of these systems are in some sense artificial, and are only needed to constrain the high level declarative rules enough to match the inference algorithm more closely! This leads one to ask if we can perhaps create a more restricted set of inference rules that match the inference algorithm more closely while still being close to the clarity of the HM type rules.

- In this article we rephrase the HM type rules as *type inference under a prefix*, called HMQ. These new rules always derive principal types that correspond to the types inferred by algorithm W [Damas and Milner 1982] (and we can use algorithm W unchanged to infer types for HMQ as well). In particular, if we can derive a type σ_1 in HM, then we can also derive a type σ_2 in HMQ such that σ_2 can be instantiated to σ_1.
- Inspired by MLF [Le Botlan and Rémy 2003], we use a *prefix* Q to propagate type variable constraints in such a way that there is no need for complex side conditions and we retain natural deduction rules. As such, we believe the HMQ type rules are close to the clarity of the HM type rules and can form an excellent basis to describe type system extensions in practice.
- We provide evidence for this by rephrasing FreezeML and HMF in terms of HMQ where there is no need anymore for the elaborate side conditions of each system. The original type rules of each system differ significantly, but building from a common HMQ specification, we can now precisely characterize the relationship between the two systems.
- Finally, we show a novel formalization of static overloading (as implemented in the Koka language). The extension is surprisingly straightforward – requiring no changes to the base system and only adding additional rules to resolve identifiers based on their context.

All proofs and supplementary material can be found in the corresponding technical report [Leijen and Ye 2024].

2 Inference under a Prefix

The goal of HMQ is two-fold: First of all, we'd like the rules to be closer to algorithm W so we are able to "read off" the algorithm from the declarative type rules. At the same time though, we'd like to retain the clarity of the original HM rules as much as possible. HMQ can serve as foundation to specify practical type systems in a declarative way that serves both purposes: users can easily reason about what programs are accepted by the type checker, while compiler writers can derive sound implementations from those same rules.

2.1 Syntax

Figure 1 shows the syntax of our standard lambda calculus expressions e, mono-types τ, and polymorphic type schemes σ. The [INSTANCE] rule gives the general instantiation where we write $\sigma_1 \sqsubseteq \sigma_2$ when a type σ_1 can be instantiated to a type σ_2. For example, $\forall\alpha\beta.\ \alpha \to \beta \sqsubseteq \forall\beta.\ \mathit{int} \to \beta \sqsubseteq \mathit{int} \to \mathit{bool}$. Note that we can only instantiate *bound* type variables $\overline{\alpha}$, and not the *free* type variables in σ_1 (written as $\mathrm{ftv}(\forall\overline{\alpha}.\ \tau)$), and in particular, $\forall\alpha.\ \alpha \to \beta \sqsubseteq \mathit{int} \to \mathit{bool}$ does not hold. A prefix Q is a set of type variable bindings, and we describe this in detail later in this section. A type environment Γ gives the types of variables bound by a lambda or let binding. We write $\Gamma, x : \sigma$ to extend a type environment with a new binding $x : \sigma$ (replacing any previous binding for x in Γ).

2.2 Type Rules

$$
\begin{array}{llll}
e & ::= & x \mid f & \text{(variables)} \\
 & \mid & e\ e & \text{(application)} \\
 & \mid & \lambda x.\ e & \text{(function)} \\
 & \mid & \mathsf{let}\ x\ =\ e\ \mathsf{in}\ e & \text{(let binding)}
\end{array}
\qquad
\begin{array}{llll}
\tau & ::= & \alpha & \text{(type variable)} \\
 & \mid & \tau \rightarrow \tau & \text{(function arrow)} \\
 & \mid & \mathit{int} \mid \mathit{bool} \mid \ldots & \text{(type constants)} \\
\sigma & ::= & \forall\alpha.\sigma & \text{(universal quant.)} \\
 & \mid & \tau & \text{(monomorphic type)}
\end{array}
$$

$$
\begin{array}{llll}
\Gamma & ::= & x_1 : \sigma_1, \ldots, x_n : \sigma_n & \text{(type environment)} \\
Q & ::= & \{\alpha_1{=}\tau_1, \ldots, \alpha_n{=}\tau_n\} & \text{(prefix)}
\end{array}
\qquad
\frac{\overline{\beta} \notin \mathrm{ftv}(\forall\overline{\alpha}.\ \tau)}{\forall\overline{\alpha}.\ \tau \sqsubseteq \forall\overline{\beta}.\ [\overline{\alpha}{:=}\overline{\tau}]\tau}\,\textsc{instance}
$$

Fig. 1. Syntax of types and terms.

$$
\boxed{\begin{array}{cccc} Q & \mid \Gamma \vdash & e & : \sigma \\ \downarrow & \uparrow & \uparrow & \downarrow \\ \text{out} & \text{in} & \text{in} & \text{out} \end{array}} \quad \text{with } \models Q
$$

$$
\frac{x : \sigma \ \in \Gamma}{\varnothing \mid \Gamma \vdash x : \sigma}\,\textsc{var}
\qquad
\frac{Q \mid \Gamma \vdash e : \forall\alpha.\sigma \quad \text{fresh}\ \alpha}{Q \mid \Gamma \vdash e : \sigma}\,\textsc{inst}
\qquad
\frac{Q \mid \Gamma \vdash e : \sigma \quad \alpha \notin \mathrm{ftv}(Q, \Gamma)}{Q \mid \Gamma \vdash e : \forall\alpha.\sigma}\,\textsc{gen}
$$

$$
\frac{Q \mid \Gamma, x : \alpha \vdash e : \tau \quad \text{fresh}\ \alpha}{Q \mid \Gamma \vdash \lambda x.\ e : \alpha \rightarrow \tau}\,\textsc{fun}
\qquad
\frac{Q \cdot \alpha{=}\tau \mid \Gamma \vdash e : \sigma \quad \alpha \notin \mathrm{ftv}(Q, \Gamma)}{Q \mid \Gamma \vdash e : [\alpha{:=}\tau]\sigma}\,\textsc{gensub}
$$

$$
\frac{Q_1 \mid \Gamma \vdash e_1 : \tau_1 \quad Q_2 \mid \Gamma \vdash e_2 : \tau_2 \quad Q_3 \vdash \tau_1 \approx \tau_2 \rightarrow \alpha \quad \text{fresh}\ \alpha}{Q_1, Q_2, Q_3 \mid \Gamma \vdash e_1\ e_2 : \alpha}\,\textsc{app}
$$

$$
\frac{Q_1 \mid \Gamma \vdash e_1 : \sigma \quad Q_2 \mid \Gamma, x : \sigma \vdash e_2 : \tau \quad \mathrm{ftv}(\sigma) \subseteq \mathrm{ftv}(\Gamma)}{Q_1, Q_2 \mid \Gamma \vdash \mathsf{let}\ x\ =\ e_1\ \mathsf{in}\ e_2 : \tau}\,\textsc{let}
$$

Fig. 2. Type rules under a prefix

Figure 2 defines the HMQ type inference rules, where a judgment $Q \mid \Gamma \vdash e : \sigma$ states that under a prefix Q and type environment Γ, we can derive type σ for the expression e. The prefix Q and the type σ are synthesized (i.e. output) while Γ and e are inherited (i.e. input).

We will go through the rules one-by-one, explaining the design decisions as we go. We start with the [VAR] and [GEN] rules that closely match the corresponding HM type rules (see Figure 8 in App. A of the techreport): the [VAR] rule gets the type of a variable from the type environment, while [GEN] generalizes over free type variables that no longer occur in Γ (and Q in our case).

2.3 Do Not Guess Types

As argued in the introduction, the guessing of types in the lambda rule is both problematic for describing type system extensions, but also for implementing an inference algorithm – what type to guess? In HMQ we follow algorithm W and always infer an *abstract* type α for a lambda-bound parameter. In particular, in the [FUN] rule the type is now always a fresh variable α – just as in algorithm W (see Figure 12 in App. C.1 of the techreport). For example, we can derive the type of the polymorphic identity function as:

$$\dfrac{\dfrac{\dfrac{x : \alpha \in (\Gamma, x : \alpha)}{\varnothing \mid \Gamma, x : \alpha \vdash x : \alpha}\textsc{var} \quad \text{fresh } \alpha}{\varnothing \mid \Gamma \vdash \lambda x.\, x : \alpha \to \alpha}\textsc{fun} \quad \alpha \notin \mathrm{ftv}(\varnothing, \Gamma)}{\varnothing \mid \Gamma \vdash \lambda x.\, x : \forall \alpha. \alpha \to \alpha}\textsc{gen}$$

Unlike the HM type rules there is no choice here for the type of the binding and we can *only* derive the type of the polymorphic identity function and not for example $int \to int$.

The other rule where we prevent guessing a type is the [INST] rule where we always instantiate directly to a fresh type α (where we rely on α-renaming to match the quantifier). Again, this corresponds to how algorithm W always instantiates using fresh type variables.

Note that we use fresh α notation to create fresh names α, not only such that α is fresh in the local rule, but also to ensure there is no other occurrence of fresh α in the derivation. This is a convenient notation for a more explicit formalization where we pass a fresh name supply using disjoint union for multiple sub-derivations – see Figure 7 in Section 8 for the full rules of HMQ. However, adding an explicit name supply clutters the rules somewhat while not adding any essential insight, and we prefer the fresh notation when applicable (i.e. when not doing proofs).

2.4 The Prefix

Clearly, we cannot always keep a parameter type abstract. For example, we'd like to infer the type $int \to int$ for the expression $\lambda x.\ inc\ x$. This is where we need the *prefix* Q, which is a set of type variable bounds $\alpha{=}\tau$ (similar to the *rigid* bounds of MLF [Le Botlan and Rémy 2003]):

$Q ::= \{\alpha_1{=}\tau_1,\ \ldots,\ \alpha_n{=}\tau_n\}$

The binders α form the domain of Q, and the types τ form the range. The codomain of Q consists of the free type variables of the range, and we define $\mathrm{ftv}(Q)$ as all free type variables in Q, where $\mathrm{ftv}(Q) = \mathrm{dom}(Q) \cup \mathrm{codom}(Q)$. Note that a general prefix is just a collection of type variable bounds, and can for example have duplicate bindings, like $\{\alpha{=}\beta{\to}int, \alpha{=}int{\to}\gamma\}$ or $\{\alpha{=}bool, \alpha{=}int\}$.

We write $\theta \vDash Q$ if a substitution θ is a *solution* to Q that satisfies all the constraints in Q where $\forall(\alpha{=}\tau) \in Q.\ \ \theta\alpha = \theta\tau$. If there exists any solution to Q, we say that Q is *consistent* or *solvable*, and we denote this by writing just $\vDash Q$. For example, $\{\beta{=}int, \alpha{=}\beta{\to}int\}$ or $\{\alpha{=}\beta{\to}int, \alpha{=}int{\to}\gamma\}$ are consistent prefixes. Examples of inconsistent prefixes that do not have a solution, are prefixes with incompatible bindings, like $\{\alpha{=}int, \alpha{=}bool\}$, or with cyclic bindings, like $\{\alpha{=}\beta, \beta{=}\alpha{\to}\alpha\}$.

We call a least solution of a prefix Q a *prefix solution*, written as $\langle Q\rangle$, such that for any other solution $\theta \vDash Q$, the prefix solution is more general[1]: $\langle Q\rangle \sqsubseteq \theta$. We write $Q[\tau]$ as a shorthand for applying the prefix solution as $\langle Q\rangle(\tau)$. Also, we sometimes leave out the angled brackets when the prefix substitution is clear from the context, and for example write $Q \sqsubseteq \theta$ for $\langle Q\rangle \sqsubseteq \theta$.

Finally, we consider two prefixes *equivalent* whenever their solution substitutions are equivalent: $Q_1 \equiv Q_2 \iff \langle Q_1\rangle \equiv \langle Q_2\rangle$. For example, we have $\{\alpha{=}\beta{\to}int, \alpha{=}\gamma{\to}\gamma\} \equiv \{\gamma{=}int, \beta{=}\gamma, \alpha{=}\gamma{\to}\gamma\} \equiv \{\beta{=}int, \gamma{=}int, \alpha{=}int{\to}int\}$. Similar to α-renaming, we can always substitute equivalent prefixes in type derivations.

2.4.1 Type Equivalence Under a Prefix. A *consistent union* is written as Q_1, Q_2 and denotes the union $Q_1 \cup Q_2$ where $Q_1 \cup Q_2$ is solvable. We use this in the conclusion of most type rules to ensure that we can only derive consistent prefixes. The consistent union allows for a better declarative specification than using substitutions, since we can *compose* prefixes from different sub-derivations as Q_1, Q_2, and we do not need to thread substitutions statefully through the rules.

[1]Following Pierce [2002,§22.4.1] we write $\theta_1 \sqsubseteq \theta_2$ to denote that θ_1 is a more-general (or less-specific) substitution than θ_2, which holds if there exists some substitution θ' such that $\theta_2 = \theta' \circ \theta_1$.

$$\boxed{\underset{\text{out}}{\underset{\downarrow}{Q}} \vdash \underset{\text{in}}{\underset{\uparrow}{\tau}} \approx \underset{\text{in}}{\underset{\uparrow}{\tau}}}$$

$$\frac{}{\varnothing \vdash \tau \approx \tau}\;\textsc{eq-id} \qquad \frac{\alpha \notin \text{ftv}(\tau)}{\{\alpha{=}\tau\} \vdash \alpha \approx \tau}\;\textsc{eq-var}$$

$$\frac{Q_1 \vdash \tau_1 \approx \tau_1' \quad Q_2 \vdash \tau_2 \approx \tau_2'}{Q_1, Q_2 \vdash \tau_1 \to \tau_2 \;\approx\; \tau_1' \to \tau_2'}\;\textsc{eq-fun} \qquad \frac{Q \vdash \tau_2 \approx \tau_1}{Q \vdash \tau_1 \approx \tau_2}\;\textsc{eq-refl}$$

Fig. 3. Type equivalence under a prefix.

The elegance of composable prefixes is shown in the definition of equivalence between types under a prefix as shown in Figure 3 (corresponding closely to unification). A rule $Q \vdash \tau_1 \approx \tau_2$ states that type τ_1 is equal to τ_2 under a prefix Q. Note how in the [eq-fun] rule we can compose the prefixes Q_1 and Q_2 from each sub derivation without needing to thread a substitution linearly through the derivations. It is straightforward to show that our definition of type equivalence is sound and complete:

Theorem 2.1. (*Type equivalence under a prefix is sound*)
If $Q \vdash \tau_1 \approx \tau_2$ then $Q[\tau_1] = Q[\tau_2]$.

Theorem 2.2. (*Type equivalence under prefix is complete*)
If $\theta\tau_1 = \theta\tau_2$, then there exists a Q such that $Q \vdash \tau_1 \approx \tau_2$ and $Q \sqsubseteq \theta$.

Soundness states that if we can derive that τ_1 and τ_2 are equivalent under a prefix Q, then the types are syntactically equal under the prefix solution: $Q[\tau_1] = Q[\tau_2]$. Completeness shows that if there exists any substitution θ that makes two types equal, then we can also derive that these types are equivalent under a prefix Q, and that such a prefix is also the "best" (most-general) solution: $\langle Q \rangle \sqsubseteq \theta$.

2.4.2 Application. We use type equivalence in the HMQ application rule [app] in Figure 2 to match the function type τ_1 with the argument type τ_2 and a fresh result type α:

$Q_3 \vdash \tau_1 \approx \tau_2 \to \alpha$

Similar to parameter types, we use a fresh type variable α to represent the result type of the application. The application rule now corresponds directly to the usual implementation in algorithm W where one unifies with the function type (see Figure 12 in App. C.1 of the techreport). We can now derive a type for the application *inc x* as:

$$\frac{\varnothing \mid \Gamma, x{:}\alpha \vdash \mathit{inc} : \mathit{int}{\to}\mathit{int} \quad \varnothing \mid \Gamma, x{:}\alpha \vdash x : \alpha \quad \{\alpha{=}\mathit{int}, \beta{=}\mathit{int}\} \vdash \mathit{int}{\to}\mathit{int} \approx \alpha{\to}\beta}{\{\alpha{=}\mathit{int}, \beta{=}\mathit{int}\} \mid \Gamma, x{:}\alpha \vdash \mathit{inc}\ x : \beta}\;\textsc{app}$$

Note that in the judgement $Q \mid \Gamma \vdash e : \sigma$, both the inferred type σ and the prefix Q are synthesized (i.e. output). Moreover, the rules are carefully set up to ensure that the resulting Q only contains constraints that are "induced" by the structure of the program and types, where the set of constraints is minimal. In particular, the only possible leaf nodes of a derivation are [var] and the type equivalence rules [eq-id] and [eq-var]. Since both [var] and [eq-id] have an empty prefix $\varnothing$, the *only* way to create (or dismiss depending on your viewpoint!) prefix constraints is through the [eq-var] rule. This property is crucial as it ensures that, unlike the HM type rules, we can never "make up" type constraints: *all constraints are induced by the structure of the program and types.*

2.4.3 Extracting Bounds to Substitute. We write $Q \;=\; Q' \cdot \alpha{=}\tau$ to *extract* a non-dependent bound $\alpha{=}\tau$ from a prefix Q such that $Q = Q' \cup \{\alpha{=}\tau\}$ with $\alpha \notin \text{ftv}(Q', \tau)$:

$$\frac{Q = Q' \cup \{\alpha{=}\tau\} \quad \alpha \notin \text{ftv}(Q', \tau)}{Q \;=\; Q' \cdot \alpha{=}\tau}\;\textsc{extract}$$

This allows us to split a prefix Q into a bound $\alpha{=}\tau$ and a remaining prefix Q' that does not depend on α. Using extraction, we can now *discharge* prefix bounds with the [GENSUB] rule. This is similiar to generalization in the [GEN] rule, except that we substitute the inferred monomorphic type bound on α. With [GENSUB][2], we can finally derive the type of $\lambda x.\ inc\ x$ as:

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\varnothing \mid \Gamma, x:\alpha \vdash\ inc\ :\ int{\to}int \qquad \varnothing \mid \Gamma, x:\alpha \vdash\ x\ :\ \alpha \\ \{\alpha{=}int, \beta{=}int\} \vdash\ int{\to}int \approx \alpha{\to}\beta \quad \text{fresh } \beta\end{array}}{\{\alpha{=}int, \beta{=}int\} \mid \Gamma, x:\alpha \vdash\ inc\ x\ :\ \beta}\text{APP} \qquad \beta \notin \text{ftv}(\{\alpha{=}int\}, \Gamma, x:\alpha)}{\dfrac{\{\alpha{=}int\} \mid \Gamma, x:\alpha \vdash\ inc\ x\ :\ int \quad \text{fresh } \alpha}{\{\alpha{=}int\} \mid \Gamma \vdash\ \lambda x.\ inc\ x\ :\ \alpha \to int}\text{FUN} \qquad \alpha \notin \text{ftv}(\varnothing, \Gamma)}\text{GENSUB}}{\varnothing \mid \Gamma \vdash\ \lambda x.\ inc\ x\ :\ int \to int}\text{GENSUB}$$

2.5 Principal Derivations

In the [LET] rule we find a single side condition: $\text{ftv}(\sigma) \subseteq \text{ftv}(\Gamma)$. This is to ensure that any free type variables in σ that do not occur in Γ are generalized by [GEN] or [GENSUB]. Since there are no longer "guessed" types, this condition is enough to guarantee that all let-bindings get a most general type. Let's rephrase the [LET] rule to use a separate judgement ($\Vdash$) for the type scheme:

$$\dfrac{Q_1 \mid \Gamma \Vdash\ e_1\ :\ \sigma \qquad Q_2 \mid \Gamma, x:\sigma \vdash\ e_2\ :\ \tau}{Q_1, Q_2 \mid \Gamma \vdash\ \text{let } x\ =\ e_1 \text{ in } e_2\ :\ \tau}\text{LET} \qquad \dfrac{Q \mid \Gamma \vdash\ e\ :\ \sigma \qquad \text{ftv}(\sigma) \subseteq \text{ftv}(\Gamma)}{Q \mid \Gamma \Vdash\ e\ :\ \sigma}\text{MGEN}$$

Since the prefix bounds are minimal, and always induced by either the structure of the program (by [APP]), or by the structure of the types (by [EQ-VAR]), the types that can be derived by the [MGEN] rule are always unique:

Theorem 2.3. (*Principal type scheme derivations*)
If $Q_1 \mid \Gamma \Vdash\ e\ :\ \sigma_1$ then for any other derivation $Q_2 \mid \Gamma \Vdash\ e\ :\ \sigma_2$, we have $\sigma_1 = \sigma_2$.

We also have that any derived mono-type is unique up to prefix substitution (due to [GENSUB]):

Theorem 2.4. (*Principal type derivations*)
If $Q_1 \mid \Gamma \vdash\ e\ :\ \tau_1$ then for any other derivation $Q_2 \mid \Gamma \vdash\ e\ :\ \tau_2$, we have $Q_1[\tau_1] = Q_2[\tau_2]$.

The proofs are given in App. D.9 of the techreport. There we also show that we can also change any derivation to use a stronger condition on [MGEN] where we require $(\text{dom}(Q) \cup \text{ftv}(\sigma)) \subseteq \text{ftv}(\Gamma)$. This condition forces any trivial substitutions (for any $\alpha{=}\tau \in Q$ with $\alpha \notin \text{ftv}(\sigma)$), which leads a stronger principality Lemma D.35 where for any $Q_1 \mid \Gamma \Vdash\ e\ :\ \sigma_1$ and $Q_2 \mid \Gamma \Vdash\ e\ :\ \sigma_2$ we also have $Q_1 = Q_2$ (besides $\sigma_1 = \sigma_2$) (see App. D.9.4 of the techreport).

Moreover, we can also show that the HMQ rules are sound and complete with respect to the standard HM type rules:

Theorem 2.5. (*Soundness*)
If $Q \mid \Gamma \vdash\ e\ :\ \sigma$ then we also have $Q[\Gamma] \vdash_{\text{HM}} e\ :\ Q[\sigma]$.

Theorem 2.6. (*Completeness*)
If $\Gamma \vdash_{\text{HM}} e\ :\ \sigma$, then there exists a θ such that $\theta\Gamma' \sqsubseteq \Gamma$, with $Q \mid \Gamma' \vdash\ e\ :\ \sigma'$, $Q \sqsubseteq \theta$, and $\theta\sigma' \sqsubseteq \sigma$.

As a corrollary, we have that [MGEN] always derives most-general types, and also that algorithm W is a valid type inference algorithm for HMQ (and since W is also complete it infers the same types

[2]We could simplify the condition $\alpha \notin \text{ftv}(Q, \Gamma)$ in [GENSUB] to just $\alpha \notin \text{ftv}(\Gamma)$ since $\alpha \notin \text{ftv}(Q)$ is implied by $Q \cdot \alpha{=}\tau$.

as HMQ derives).

The soundness theorem states that if we can derive a type σ under a prefix Q in HMQ, then we can also derive the type $Q[\sigma]$ in HM (see Figure 8 in App. A of the techreport for the definition of $\vdash_{\text{HM}}$). We need to apply the prefix to σ since it can still contain bounds (that could be applied with [GENSUB]).

The completeness theorem is more involved. We may have expected to see a simpler statement like: if $\Gamma \vdash_{\text{HM}} e : \sigma$, then $Q \mid \Gamma' \vdash e : \sigma'$ with $Q[\sigma'] \sqsubseteq \sigma$. That does not hold though since derivations may contain abstract types in our system. In particular, any lambda bound parameter always has an "abstract" type α and there may be no bound yet.

For example, $$\dfrac{\dfrac{\dfrac{x : \alpha \in (x : \alpha)}{\varnothing \mid x : \alpha \vdash x : \alpha}\ \text{VAR}}{\varnothing \mid \varnothing \vdash \lambda x.\, x : \alpha \to \alpha}\ \text{FUN}}{\varnothing \mid \varnothing \vdash \lambda x.\, x : \forall\alpha.\alpha\to\alpha}\ \text{GEN}$$, but in the HM type rules, we can also derive: $$\dfrac{\dfrac{x : \mathit{int} \in (x : \mathit{int})}{x : \mathit{int} \vdash_{\text{HM}} x : \mathit{int}}\ \text{VAR}}{\varnothing \vdash_{\text{HM}} \lambda x.\, x : \mathit{int} \to \mathit{int}}\ \text{FUN}$$

For an inductive proof, it means that for the [VAR] case, we would need to show that if we derive $x : \mathit{int} \vdash_{\text{HM}} x : \mathit{int}$, we can also derive $\varnothing \mid x : \alpha \vdash x : \alpha$ with $\varnothing[\alpha] \sqsubseteq \mathit{int}$ which does not hold.

Therefore, the actual completeness theorem states that if there exists some substitution θ with $\theta\Gamma' \sqsubseteq \Gamma$, with $Q \sqsubseteq \theta$ with $\theta\sigma' \sqsubseteq \sigma$. In our example, when we use $\theta = [\alpha{:=}\mathit{int}]$, we indeed have $\varnothing \sqsubseteq [\alpha{:=}\mathit{int}]$ and $[\alpha{:=}\mathit{int}]\alpha \sqsubseteq \mathit{int}$. There is one more subtlety in that we need to use $\theta\sigma' \sqsubseteq \sigma$ and cannot use equality as $\theta\sigma' = \sigma$. In particular, in the HM type rules we can also introduce more sharing for let-bindings than we can in HMQ. For let $\mathit{const} = \lambda x.\lambda y.\, x$ we always infer $\mathit{const} : \forall\alpha\,\beta.\ \alpha\to\beta\to\alpha$ in HMQ but under the HM rules we can also derive the type $\forall\alpha.\alpha\to\alpha\to\alpha$. In such a case, for the [VAR] rule we still need to show $\forall\alpha\,\beta.\ \alpha\to\beta\to\alpha \sqsubseteq \forall\alpha.\alpha\to\alpha\to\alpha$. (and thus we need the instance relation $\sqsubseteq$). For the inductive proof, the full required completeness theorem is still a bit more general than stated here (see App. D.7 of the techreport for details).

2.6 Idempotent Mappings

The reader may have noticed that [GENSUB] may not always apply as we cannot always extract a binding α even if $\alpha \notin \text{ftv}(\Gamma)$. In particular, there might be multiple bounds for a type variable in the prefix, like $\{\alpha{=}\beta\to\mathit{int}, \alpha{=}\mathit{int}\to\gamma\}$, and in that case we cannot extract α directly for the [GENSUB] rule (since $\alpha \in \text{ftv}(Q')$). However, for any consistent prefix, we can always simplify multiple bindings:

Theorem 2.7. (*Simplify*)
If $Q' \vdash \tau_1 \approx \tau_2$, then $Q \cup \{\alpha{=}\tau_1, \alpha{=}\tau_2\} \equiv Q \cup Q' \cup \{\alpha{=}\tau_1\}$

For example, $\{\alpha{=}\beta\to\mathit{int}, \alpha{=}\mathit{int}\to\gamma\} \equiv \{\beta{=}\mathit{int}, \gamma{=}\mathit{int}, \alpha{=}\beta\to\mathit{int}\}$. By using repeated simplification, we can always bring a consistent prefix into a form where all bindings are distinct (called a *mapping*).

Even with a mapping, there are still cases where we may have a dependency that prevents extraction. For example, consider extracting α from $\{\beta{=}\alpha,\ \alpha{=}\mathit{int}\to\mathit{int}\}$ (where $\alpha \in \text{ftv}(\{\beta{=}\alpha\})$). It turns out though that any consistent prefix is always equivalent to an *idempotent* mapping where $\text{dom}(Q) \not\pitchfork \text{codom}(Q)$, e.g. $\{\beta{=}\alpha,\ \alpha{=}\mathit{int}\to\mathit{int}\} \equiv \{\beta{=}\mathit{int}\to\mathit{int},\ \alpha{=}\mathit{int}\to\mathit{int}\}$.

Theorem 2.8. (*Any consistent prefix is equivalent to an idempotent mapping*)
If $\vDash Q$, then there exists an equivalent idempotent mapping Q' (where $Q \equiv Q'$, $|\text{dom}(Q')| = |Q'|$ and $\text{dom}(Q') \not\pitchfork \text{codom}(Q')$).

This essentially allows us to always simplify a prefix enough to apply [GENSUB] for any binding α in Q where $\alpha \notin \text{ftv}(\Gamma)$.

$$\boxed{\underset{\substack{\downarrow\\ \text{out}}}{Q} \vdash \underset{\substack{\uparrow\\ \text{in}}}{\tau} \overset{\rightarrow}{\approx} \underset{\substack{\uparrow\\ \text{in}}}{\tau} \rightarrow \underset{\substack{\downarrow\\ \text{out}}}{\tau}} \qquad \frac{Q \vdash \tau_1 \approx \tau_2}{Q \vdash \tau_1 \rightarrow \tau \overset{\rightarrow}{\approx} \tau_2 \rightarrow \tau}\,\text{MFUN} \qquad \frac{Q \vdash \alpha \approx \tau \rightarrow \beta \quad \text{fresh } \beta}{Q \vdash \alpha \overset{\rightarrow}{\approx} \tau \rightarrow \beta}\,\text{MVAR}$$

$$\frac{Q_1 \mid \Gamma \vdash e_1 : \tau_1 \qquad Q_2 \mid \Gamma \vdash e_2 : \tau_2 \qquad Q_3 \vdash \tau_1 \overset{\rightarrow}{\approx} \tau_2 \rightarrow \tau}{Q_1, Q_2, Q_3 \mid \Gamma \vdash e_1\ e_2 : \tau}\,\text{APP-MATCH}$$

Fig. 4. Function matching.

2.7 Flexible Bounds

The idea of inference under a prefix is inspired by the use of a prefix in the MLF type system [Le Botlan 2004; Le Botlan and Rémy 2003]. In MLF, the prefix does not just contain *rigid* bounds of the form $\alpha{=}\tau$, but also *flexible* bounds of the form $\alpha{\geqslant}\sigma$, which allows α to be any instance of σ. The flexible bound $\alpha{\geqslant}\bot$ allows α to be instantiated to any type. Finally, MLF uses quantification over a prefix, as in $\forall Q.\ \tau$ where $\forall\alpha.\sigma$ is a shorthand for $\forall\alpha{\geqslant}\bot.\ \sigma$. Moreover, rigid monomorphic bounds can be inlined, and $\forall\alpha{=}\tau.\ \sigma$ is equivalent to $[\alpha{:=}\tau]\sigma$.

Using these richer bounds for a prefix, it is possible to use a single generalization rule instead of both [GEN] and [GENSUB]. Let's extend our bounds to include $\alpha{\geqslant}\bot$ bounds as:

$$\alpha{\diamond}\rho ::= \alpha{=}\tau \mid \alpha{\geqslant}\bot \qquad\qquad Q ::= \{\ \alpha_1{\diamond_1}\rho_1,\ \ldots,\ \alpha_n{\diamond_n}\rho_n\ \}$$

We can then use a single generalization rule as:

$$\frac{Q \cdot (\alpha{\diamond}\rho) \mid \Gamma \vdash e : \sigma \qquad \alpha \notin \text{ftv}(Q, \Gamma)}{Q \mid \Gamma \vdash e : \forall(\alpha{\diamond}\rho).\ \sigma}\,\text{GENX}$$

This rule now concisely subsumes both [GEN] and [GENSUB] (and corresponds exactly to the [GEN] rule of MLF [Le Botlan 2004,Fig. 5.2]). We would also extend simplification to merge flexible and rigid bounds where $Q \cup \{\alpha{\geqslant}\bot, \alpha{=}\tau\}$ simplifies to $Q \cup \{\alpha{=}\tau\}$.

We chose the current presentation in this paper for simplicity. Nevertheless, we believe that the use of an extended prefix with $\alpha{\geqslant}\bot$ bounds is perhaps more natural from a technical perspective and might also be better suited for example to extend HMQ to the MLF type rules.

2.8 Function Matching

The current [APP] in Figure 2 has a drawback that it always creates a fresh type variable α for the result type. In practice, most implementations instead first match on the inferred type for e_1 to see if it is a function type $\tau'{\rightarrow}\tau$ already – and in that case directly use τ for the result type.

We can express this technique declaratively in HMQ as well. Figure 4 shows an improved [APP-MATCH] rule that avoids creating a fresh result type variable by matching on the function type as $Q \vdash \tau_1 \overset{\rightarrow}{\approx} \tau_2 \rightarrow \tau$, where τ_1 and τ_2 are given, and Q and the result type τ are synthesized. The ($\overset{\rightarrow}{\approx}$) judgment has two rules. The [MFUN] rule handles the case where it is already a function type and directly matches the expected parameter type with the inferred argument type. The only other possible case is that the type of e_1 is still an abstract α (for example, in $\lambda f.\ f\ 1$). The [MVAR] rule in that case applies and does create a fresh result type variable as before.

3 Implementing Inference Under a Prefix

We believe the type rules in Figure 2 form a nice declarative specification of the type system where users can easily reason about what programs are accepted. At the same time though, it is possible to "read off" a type inference algorithm from the same rules. First we discuss how a *direct* implementation would look, and then consider a more standard implementation based on algorithm W.

3.1 Deriving a Direct Implementation

To directly derive an algorithm from the type rules, we first need to make the rules syntax-directed since the instantiation and generalization rules can be applied at any time. Following Damas and Milner [1982], we can make the rules syntax-directed by doing full instantiation at the leaves (in the [VAR] rule), and full generalization (with the [GEN] and [GENSUB] rules) at let-bindings. For example, the syntax directed rules for variables and let-bindings become:

$$\frac{x : \forall\overline{\alpha}.\tau \in \Gamma \quad \text{fresh } \overline{\alpha}}{\varnothing \mid \Gamma \vdash_s x \,:\, \tau} \qquad \frac{Q_0 \mid \Gamma \vdash_s e_1 \,:\, \tau_1 \quad Q_2 \mid \Gamma, x : \sigma \vdash_s e_2 \,:\, \tau_2 \quad (Q_1, \sigma) = \mathsf{gen}(Q_0, \Gamma, \tau_1)}{Q_1, Q_2 \mid \Gamma \vdash_s \mathsf{let}\ x \,=\, e_1\ \mathsf{in}\ e_2 \,:\, \tau_2}$$

where $\mathsf{gen}(Q_0, \Gamma, \tau_1)$ generalizes a type τ_1 with respect to a given environment Γ and prefix Q_0. See Figure 9 in App. B of the techreport for the full syntax-directed rules. We can now almost implement each rule directly, except that we need a way to compute the consistent union between prefixes.

3.1.1 Computing the Prefix Solution. Any initial prefix at the leaves of a derivation is always either empty or a singleton $\{\alpha{=}\tau\}$ (with $\alpha \notin \mathsf{ftv}(\tau)$). If we ensure that we always create an idempotent mapping from a consistent union Q_1, Q_2 then all our prefixes are always an idempotent mapping – and we can represent them in our implementation as regular substitutions; effectively representing Q as its minimal solution $\langle Q\rangle$. Using our notion of type equivalence, we can derive a straightforward algorithm to compute the prefix solution. In particular, we have that extraction corresponds to composition of prefix solutions:

Lemma 3.9. (*Extraction corresponds to composition of prefix solutions*)
If $\vDash Q$ and $Q = Q' \cdot \alpha{=}\tau$, then $\langle Q\rangle = \langle Q'\rangle \circ [\alpha{:=}\tau]$.

Using this lemma, we can write the initial cases of our algorithm as:

$$\mathit{solve}(\varnothing) \;=\; \mathit{id} \qquad \text{and,} \qquad \mathit{solve}(Q \cup \{\alpha{=}\tau\}) \;=\; \mathit{solve}(Q) \circ [\alpha{:=}\tau] \qquad \text{if } \alpha \notin \mathsf{ftv}(Q, \tau)$$

If we cannot find any α with $\alpha \notin \mathsf{ftv}(Q, \tau)$, that leaves two other cases to consider. If $\alpha \in \mathsf{ftv}(\tau)$ or $\alpha \in \mathsf{ftv}(\mathsf{rng}(Q))$ there must be cyclic dependency and there is no solution. Otherwise, there must be duplicate bindings (with $\alpha \in \mathsf{dom}(Q)$), and in such a case we can use Theorem 2.7 to simplify the duplicate bindings[3]:

$$\begin{array}{lll}
\mathit{solve} \,:\, Q \longrightarrow \theta & & \\
\mathit{solve}(\varnothing) & = \;\mathit{id} & \\
\mathit{solve}(Q \cup \{\alpha{=}\tau\} & = \;\mathit{solve}(Q) \circ [\alpha{:=}\tau] & \text{if } \alpha \notin \mathsf{ftv}(Q, \tau) \\
\mathit{solve}(Q \cup \{\alpha{=}\tau_1, \alpha{=}\tau_2\}) & = \;\mathit{solve}(Q \cup Q' \cup \{\alpha{=}\tau_1\}) & \text{if } Q' \vdash \tau_1 \approx \tau_2 \;\wedge \alpha \notin \mathsf{ftv}(\tau_1, \tau_2, \mathsf{rng}(Q))
\end{array}$$

Essentially this algorithm picks non-dependent bindings and composes them recursively, while simplifying duplicate bindings away by unifying their types using the equivalence relation. But how can we compute $Q' \vdash \tau_1 \approx \tau_2$? To derive an implementation for the equivalence relation we need to make these syntax-directed as well. Similar to instantiation we can always apply [EQ-REFL] at the leaves of the derivation at the [EQ-VAR] rule and make them syntax-directed. We can then derive an implementation, representing prefixes again as substitutions, as:

$$\begin{array}{ll}
\mathit{equiv}\ (\alpha, \alpha) & = \mathit{id} \\
\mathit{equiv}\ (\alpha, \tau) \text{ or } (\tau, \alpha) \mid \alpha \notin \mathsf{ftv}(\tau) & = [\alpha{:=}\tau] \\
\mathit{equiv}\ (\tau_1{\longrightarrow}\tau_2,\ \tau_1'{\longrightarrow}\tau_2') & = \mathsf{let}\ \theta_1 = \mathit{equiv}(\tau_1, \tau_1');\ \theta_2 = \mathit{equiv}(\tau_2, \tau_2'\ \mathsf{in}\ \mathit{solve}(\theta_1 \cup \theta_2)
\end{array}$$

[3]The extra side condition $\alpha \notin \mathsf{ftv}(\tau_1, \tau_2, \mathsf{rng}(Q)))$ is needed here to ensure that *solve* terminates for any inconsistent Q with cyclic bindings – consider for example $\mathit{solve}(\{\beta{=}\alpha, \alpha{=}\beta, \alpha{=}\mathit{int}\})$.

This is recursive with *solve* but we can show it is terminating since the number of free type variables is decreasing on each recursive invocation [Pierce [2002],§22.4.5].

Since we happen to represent the prefixes as a substitutions in our implementation, we can now compute prefix composition as $Q_1, Q_2 = solve(Q_1 \cup Q_2)$, and directly "read off" an inference algorithm from our type rules. For example, the inference case for the [$\textsc{app}_\textsc{s}$] application rule becomes (using substitutions θ for idempotent mapping prefixes Q):

```
$inferD\ (\Gamma, e_1\ e_2)\ :\ (\Gamma, e) \longrightarrow (\theta, \tau)\ =$
  let $(\theta_1, \tau_1)\ =\ inferD(\Gamma, e_1);\ (\theta_2, \tau_2)\ =\ inferD(\Gamma, e_2)$
  let $\alpha\ =$ fresh; $\theta_3\ =\ equiv(\tau_1, \tau_2 \rightarrow \alpha)$
  let $\theta\ =\ solve(solve(\theta_1 \cup \theta_2) \cup \theta_3)$
  $(\theta, \alpha)$
```

3.1.2 Robinson Unification and Substitution Unification. Of course, we can also readily use standard Robinson unification [Robinson 1965] to compute $\langle Q \rangle$ as well. In particular, if we view Q as a set of type constraints C with constraints of the form $\tau_1{=}\tau_2$, we can use the standard $\mathsf{unify}(C)$ algorithm from Pierce [2002,§22.4] to compute the most general unifier of the constraints in Q – which is $\langle Q \rangle$ by definition (and therefore $solve(Q) = \mathsf{unify}(Q)$). An approach that maps more directly to the idea of joining prefixes is the work by McAdam [1999] – describing an algorithm U_s for unifying substitutions which can be used directly to implement joining (idempotent mapping) prefixes where $Q_1, Q_2 \equiv U_s(Q_1, Q_2) \circ Q_1$ (and thus $solve(Q_1 \cup Q_2) = U_s(Q_1, Q_2) \circ Q_1$)

Nevertheless, we prefer *solve* as that is parameterized by our type equivalence rules, $Q \vdash\ \tau_1 \approx \tau_2$, to determine equivalent types and least solutions. In contrast, the type equivalence is "built-in" in the unify and U_s algorithms. We believe that our approach lends itself better to type system extensions, like record types or impredicative types, where the equality between types can go beyond syntactical equality. In such cases, it is straightforward to extend our type equivalence relation with further rules.

3.2 Algorithm W

Even though we can implement the syntax-directed rules directly with *inferD*, it may not be the most efficient way to do this. However, since HMQ is sound and complete with respect to the HM type rules, we can also directly use algorithm W as our inference algorithm *as-is*. That means also that any efficient implementation, for example using in-place updating substitutions [Peyton Jones et al. 2007] or level-based generalization [Kiselyov 2022; Kuan and MacQueen 2007; Rémy 1992], is correct for HMQ as well.

There is a catch though – even though algorithm W is correct for the basic type rules of HMQ, it may not be correct anymore for some extensions and we need to be a bit more careful. In particular, we can view algorithm W as an optimized version of the derived *inferD* direct implementation: in an application for example, the direct implementation unifies type constraints by joining prefixes of separate sub derivations, while in algorithm W we use a single substitution that is threaded linearly through each sub derivation, resolving unification constraints eagerly. This linear traversal is what allows for an efficient in-place updating implementation of substitutions. For example, the case for applications in algorithm W is [Damas and Milner 1982]:

```
$inferW(\Gamma, e_1\ e_2)\ :\ (\Gamma, e) \longrightarrow (\theta, \tau)\ =$
  let $(\theta_1, \tau_1)\ =\ inferW(\Gamma, e_1);\ (\theta_2, \tau_2)\ =\ inferW(\theta_1\Gamma, e_2)$
  let $\alpha\ =$ fresh; $\theta_3\ =\ unify(\theta_2\tau_1,\ \tau_2 \rightarrow \alpha)$
  $(\theta_3 \circ \theta_2 \circ \theta_1,\ \theta_3\alpha)$
```

where we see that the substitution θ_1 is applied to Γ when checking e_2 (see Figure 12 in App. C.1 of the techreport for the full algorithm).

For the basic type rules this makes no difference, but if we were to inspect the types of λ-bound parameters the implementations start to differ. In algorithm W, since the substitution is updated eagerly, type information from an early variable occurrence may "leak" into another sub derivation – we call this spooky action at a distance. Consider for example

$\lambda x.\ (inc\ x,\ \ show\ x)$

At the first occurrence of x the type will be some fresh type α, and after checking the $inc\ x$ expression, we'll have a substitution $[\alpha{:=}int]$. When this substitution is propagated into the second derivation, the second occurrence of x in $show\ x$ now has the substituted type int in algorithm W! For static overloading (described in Section 6) this would mean that $show\ x$ can be resolved while it should be rejected according to the HMQ type rules where x always has an abstract type (and worse, it leaks the left-to-right bias of algorithm W, where $\lambda x.\ (show\ x,\ \ inc\ x)$ would be rejected). The *inferD* direct implementation does not have this problem as it derives a prefix/substitution for each sub derivation separately (joining them later in *solve*) – no spooky action at a distance.

3.3 Algorithm WQ

It turns out that a small change to algorithm W can prevent spooky action at a distance, and prevent leaking type information between separate sub derivations even when using efficient stateful substitutions. The core issue is that in algorithm W the fresh type variable for a λ-bound parameter is shared between sub derivations. What we can do instead is to use separate fresh type variables for each occurrence of a λ-bound parameter and unify them all eventually. In particular, in an expression $\lambda x.e$ we could name all occurrences of a lambda bound parameter x in e sequentially to x_i (as e'), and rewrite to $\lambda x.\ (\lambda x_1 \ldots x_n.\ e')\ x \ldots x$. After such transformation, algorithm W will effectively create a fresh type variable for each x_i and only unify them all afterwards (with the type of x), thus preventing spooky action a distance.

We can do this more efficiently by integrating this strategy directly in a revised algorithm WQ, where we generate a fresh "template" type variable α for a parameter, but expand that to a unique type variable α_i at each occurrence of a λ-bound parameter:

```
$inferWQ(\Gamma, x_i)$ =
  let $\alpha$ = $\Gamma(x)$
  $(id, \alpha_i)$
```

```
$inferWQ(\Gamma, \lambda x.e)$ =
  let $\alpha$ = fresh
  let $(\theta_1, \tau)$ = $inferWQ((\Gamma, x : \alpha), e)$
  let $\theta_2$ = $unifies(\alpha, \theta_1\alpha_1, \ldots, \theta_1\alpha_n)$
  $(\theta_2, \theta_2(\alpha \to \tau))$
```

The full algorithm WQ can be found in Figure 10 in App. B.1 of the techreport. For our standard rules, this change to algorithm W only changes when certain type errors happen (which are now sometimes delayed). However, when adding type propagation (Section 5) and overloading (Section 6), the new algorithm WQ prevents spooky action at a distance, and type information in separate sub derivations can no longer be accidentally shared. Note also that in practice we can use a stateful counter at each λ-bound parameter to avoid doing a separate pre-processing step, while still being able to unify all freshly instantiated type variables for a parameter afterwards.

As such, algorithm WQ is a modest extension to algorithm W and we believe that it is straightforward to adapt for type system implementations in practice, while simultaneously benefitting from being able to use HMQ to specify the type rules and further extensions in a concise manner that matches the implementation closely.

4 Inference under a Prefix for FreezeML and HMF

We believe HMQ can be an excellent basis to describe common type system extensions in practice that are difficult to formalize directly in the HM type rules. In this section we look at some of the previous work on higher-rank and impredicative type inference, and consider how these systems

$$\frac{}{\varnothing \vdash \tau \approx \tau}\textsc{eqf-id} \qquad \frac{Q \vdash \sigma_2 \approx \sigma_1}{Q \vdash \sigma_1 \approx \sigma_2}\textsc{eqf-refl} \qquad \frac{Q \vdash \sigma_1 \approx \sigma_2 \quad \alpha \notin \mathsf{ftv}(Q)}{Q \vdash \forall\alpha.\sigma_1 \approx \forall\alpha.\sigma_2}\textsc{eqf-poly}$$

$$\frac{\alpha \notin \mathsf{ftv}(\sigma)}{\{\alpha{=}\sigma\} \vdash \alpha \approx \sigma}\textsc{eqf-var} \qquad \frac{Q \vdash \sigma_1 \approx \sigma_2}{Q \vdash [\sigma_1] \approx [\sigma_2]}\textsc{eqf-list} \qquad \frac{Q_1 \vdash \sigma_1 \approx \sigma_2 \quad Q_2 \vdash \sigma'_1 \approx \sigma'_2}{Q_1, Q_2 \vdash \sigma_1{\to}\sigma_2 \approx \sigma'_1{\to}\sigma'_2}\textsc{eqf-fun}$$

Fig. 5. Equivalence of System F types under a prefix.

could be viewed in terms of inference under a prefix. We consider in particular the recent FreezeML system [Emrich et al. 2020 2022] and HMF [Leijen 2007 2008]. Note that for the purposes of this article we restrict ourselves to highlight essential differences only – the goal of this section is to show how inference under a prefix may be a better way to formalize and compare such systems, and it is not meant as a general introduction to impredicative type inference.

Generally, these systems allow for higher-rank (i.e. nested quantifiers) and impredicative types (i.e. polymorphic types in a data structure), and extend the syntax of types essentially with:

σ	$::=$	$\forall\alpha.\sigma$	(quantification)	ρ	$::=$	$\sigma \to \sigma$	(higher-rank function)
	$\mid$	ρ	(no outer quantifier)		$\mid$	τ	(monomorphic types)
					$\mid$	$[\sigma]$	((impredicative) list of σ)

where we restrict ourselves to impredicative lists for example purposes. The instance relation can now instantiate polymorphic types as well:

$$\frac{\overline{\beta} \notin \mathsf{ftv}(\forall\overline{\alpha}.\ \sigma_1)}{\forall\overline{\alpha}.\sigma_1 \sqsubseteq \forall\overline{\beta}.[\overline{\alpha}{:=}\overline{\sigma}]\sigma_1}\textsc{instancef}$$

Generally, impredicative systems are *invariant* where we can only instantiate the outer quantifiers (as in Damas-Hindley-Milner), but not any inner quantifiers. For example, we can instantiate the identity function as $\forall\alpha.\alpha{\to}\alpha \sqsubseteq int{\to}int$, but we cannot instantiate a list of polymorphic identity functions as $[\forall\alpha.\alpha{\to}\alpha] \not\sqsubseteq [int{\to}int]$. In HMQ this shows up clearly when defining new type equivalence rules that take impredicative types into consideration as shown in Figure 5.

Note that we extended the prefix to include (rigid) polymorphic bounds $\alpha{=}\sigma$. Also, to prevent the bound α in the [eqf-poly] rule from escaping into Q, we need the side-condition $\alpha \notin \mathsf{ftv}(Q)$. The new equivalence rules again closely resemble common unification algorithms for impredicative types [Emrich et al. 2020,Fig.15; Leijen 2008,Fig.5]. To implement the [eqf-poly] rule one usually instantiates both outer quantifiers with a fresh constant (often called a “skolem” constant) and afterwards check that the constant does not escape into Q.

There are two troublesome cases to consider with impredicative type inference. Generally, we cannot infer polymorphic types for lambda-bound parameters. Consider for example:

$poly\ =\ \lambda f.\ (f\ 1,\ f\ True)$

This would be rejected in HM systems since there is no monomorphic type for f that can be applied to both an *int* and a *bool*. We could assign a polymorphic type to f though – like $\forall\alpha.\alpha{\to}\alpha$. Unfortunately, there is no principal type for f and there are many other incomparable types possible, like $\forall\alpha.\alpha{\to}[\alpha]$ etc. The systems we discuss therefore never infer a polymorphic type for a lambda-bound parameter and require a type annotation for polymorphic parameters which can express directly in HMQ as well:

$$\frac{Q \mid \Gamma, x{:}\sigma \vdash e : \tau}{Q \mid \Gamma \vdash \lambda(x{:}\sigma).\ e : \sigma \to \tau}\textsc{fun-ann}$$

The second issue occurs at applications where there is sometimes a choice between instantiations. Consider the application *single id* where *single* has type $\forall\alpha.\alpha\to[\alpha]$. If we instantiate *id* first, the result type is $\forall\alpha.[\alpha\to\alpha]$ after generalization – but if we keep *id* polymorphic, the result type is a list of polymorphic identity functions $[\forall\alpha.\alpha\to\alpha]$ instead. Unfortunately, neither type is an instance of the other.

Generally, different proposed systems handle this case in very different ways. Here, we take a closer look at the FreezeML and HMF systems specifically, and consider how they could be simplified by using prefix based inference.

4.1 FreezeML

FreezeML [Emrich et al. 2020 2022] is an impredicative type inference system based on the idea of *freezing* the polymorphic type of a variable occurrence, written as $\lceil x \rceil$, and only allowing instantiation at regular variable occurrences x. Alas, that also means FreezeML is fundamentally syntax directed and we need to base a “FreezeHMQ” version on the syntax directed rules of HMQ (see Fig. 9 in App. B of the techreport), where we instantiate at variable occurrences, and generalize ([GEN] and [GENSUB]) at let-bindings. The freezing rule of FreezeML becomes:

$$\frac{x : \sigma \ \in \Gamma}{\varnothing \mid \Gamma \vdash_s \lceil x \rceil \ : \ \sigma}\ \text{FREEZE}$$

Since we no longer can instantiate freely, a frozen type σ stays polymorphic while regular variable occurrences have instantiated ρ types. This resolves the *single id* ambiguity: *single id* has the HM type $\forall\alpha.[\alpha\to\alpha]$ since *id* is fully instantiated. If we wish to create a list of polymorphic identity functions we would write *single* $\lceil id \rceil$ instead (which has type $[\forall\alpha.\alpha\to\alpha]$).

Unfortunately, even though FreezeML is syntax directed, it still requires let-bindings to have most-general types. Consider for example let $f = \lambda x.x$ in $\lceil f \rceil$ 42. We would expect this to be rejected with the frozen type for f as $\forall\alpha.\alpha\to\alpha$ (wich cannot be directly applied). However, if we allow less-general types for let-bindings we could also derive the type *int*$\to$*int* for f and in that case the example *can* be typed. To resolve this, the [LET] rule in FreezeML adds the principal condition [Emrich et al. 2020,Fig. 7&8]:

$$\frac{\begin{array}{l}(\_, \Delta') = \mathsf{gen}(\Delta, \sigma_1, e) \quad (\Delta, \Delta', e, \sigma_1) \Updownarrow \sigma \\ \Delta, \Delta' \mid \Gamma \vdash e_1 \ : \ \sigma_1 \quad \Delta \mid \Gamma, x : \sigma \vdash e_2 \ : \ \sigma_2 \\ \mathsf{principal}(\Delta, \Gamma, e, \Delta', \sigma_1)\end{array}}{\Delta \mid \Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \ : \ \sigma}\ \text{LET-FML}$$

$$\begin{array}{l}\mathsf{principal}(\Delta, \Gamma, e, \Delta', \sigma') \ = \\ \quad \Delta' = \mathsf{ftv}(\sigma) - \Delta \text{ and } \Delta, \Delta' \mid \Gamma \vdash e \ : \ \sigma \text{ and} \\ \quad (\forall\Delta'', \sigma''. \text{ if } \Delta'' = \mathsf{ftv}(\sigma'') - \Delta \\ \qquad\qquad \text{and } \Delta, \Delta'' \mid \Gamma \vdash e \ : \ \sigma'' \\ \qquad\quad \text{then } \exists\delta.\ \Delta \vdash \delta :\ \Delta' \Rightarrow_\star \Delta'' \\ \qquad\qquad\qquad \text{and } \delta(\sigma') = \sigma'')\end{array}$$

We can disregard the $\Updownarrow$ rule as that is related to the value-restriction, and we can similarly ignore the Δ environment that tracks the free variables. The principal condition enforces that all let bindings are assigned most general types. Since it ranges over all possible derivations where the type inference judgment occurs negatively, it is not a natural deduction rule which makes it hard to reason about. Emrich et al [2020,§3.2] show though that it is still possible to stratify the relation to allow inductive reasoning.

However, in “FreezeHMQ” none of this complexity is required as we already always derive principal types, and we can keep the regular (syntax-directed) [LET] rule as is. We only need to extend the types and type equivalence as shown in the previous section together with [FUN-ANN] and [FREEZE] to model FreezeML. This also shows that an implementation of type inference for FreezeML only requires a modest extension of algorithm W – essentially just extending unification according to the rules in Figure 5 (as in [Emrich et al. 2020,Fig. 15]).

4.2 HMF

As another example, we take a close look at the HMF system [Leijen 2008], which is used for impredicative type inference in the Koka language [Leijen 2014 2021]. Unlike FreezeML, the HMF rules are not required to be syntax-directed and one can freely instantiate and generalize. The HMF system, however, contains two inference rules with complex side-conditions:

$$\frac{\Gamma \vdash e_1 : \sigma_1 \quad \Gamma, x{:}\sigma_1 \vdash e_2 : \sigma_2 \qquad \forall \sigma_1'.\ \Gamma \vdash e_1 : \sigma_1' \Rightarrow \sigma_1 \sqsubseteq \sigma_1'}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2}\ \textsc{hmf-let} \qquad \frac{\Gamma \vdash e_1 : \sigma_2 \to \sigma \quad \Gamma \vdash e_2 : \sigma_2 \qquad \forall \sigma'\ \sigma_2'.\ (\Gamma \vdash e_1 : \sigma_2' \to \sigma' \wedge \Gamma \vdash e_2 : \sigma_2') \Rightarrow [\![\sigma_2 \to \sigma]\!] \leqslant [\![\sigma_2' \to \sigma']\!]}{\Gamma \vdash e_1\ e_2 : \sigma}\ \textsc{hmf-app}$$

As before, the [hmf-let] rule requires that we only assign most general types to let-bindings – but this comes for free in HMQ and we can again can use our regular [let] rule as is. In the [hmf-app] rule, the condition requires that the inferred type must be the one with a minimal *polymorphic weight* (denoted as $[\![\sigma]\!]$), where the polymorphic weight of a type is defined as the number of nested quantifiers. This is how HMF disambiguates the *single id* example, which has the type $\forall\alpha.[\alpha{\to}\alpha]$ since that is the one with minimal nested polymorphism (and if a list of polymorphic identity functions is required one needs to use a type signature).

Just as in the [let-hmf] rule, the side condition is stated over all derivations again – in this case this is needed as a polymorphic instantation can be further up in the derivation. This issue is already avoided though in HMQ since the [inst] rule never guesses types and always instantiates with an abstract type variable. As a consequence, it is possible to locally extend the function matching in the application rule to explicitly disambiguate.

Leijen [2008] observes that the only ambiguity can arise when a function of the form $\alpha \to \ldots$ is applied to a polymorphic argument σ. In such a case we need to instantiate the σ and not unify directly with α. We can extend the function match relation in Figure 4 to do this disambiguation:

$$\boxed{\begin{array}{cccccc} Q & \vdash & \rho & \overset{\rightarrow}{\approx} & \sigma \to & \sigma \\ \downarrow & & \uparrow & & \uparrow & \downarrow \\ \text{out} & & \text{in} & & \text{in} & \text{out} \end{array}} \qquad \frac{Q_1 \mid \Gamma \vdash e_1 : \rho \quad Q_2 \mid \Gamma \vdash e_2 : \sigma_2 \quad Q_3 \vdash \rho \overset{\rightarrow}{\approx} \sigma_2 \to \sigma}{Q_1, Q_2, Q_3 \mid \Gamma \vdash e_1\ e_2 : \sigma}\ \textsc{app-hmf-match}$$

$$\frac{Q \vdash \rho_1 \approx \rho_2}{Q \vdash \rho_1 \to \sigma \overset{\rightarrow}{\approx} \rho_2 \to \sigma}\ \textsc{mfun} \qquad \frac{Q \vdash \alpha \approx \rho \to \beta \quad \text{fresh } \beta}{Q \vdash \alpha \overset{\rightarrow}{\approx} \rho \to \beta}\ \textsc{mvar} \qquad \frac{Q \vdash \sigma_1 \approx \sigma_2 \quad \sigma_1 \notin \rho}{Q \vdash \sigma_1 \to \sigma \overset{\rightarrow}{\approx} \sigma_2 \to \sigma}\ \textsc{mquant}$$

The [mfun] and [mvar] rules are as before but extended to apply to impredicative ρ types. The [mquant] rule is added and matches an actual polymorphic parameter type σ_1 (where σ_1 cannot be an unquantified ρ-type). This ensures that in the *single id* case, we must use the regular [mfun] rule which forces the argument type to be instantiated (as ρ_2) (and thus *single id* has type $\forall\alpha.[\alpha{\to}\alpha]$).

The new function match, together with the new type equivalence as shown in the previous section are the only changes needed to phrase HMF as inference under a prefix! This again also implies that only a modest extension to algorithm W is required to implement HMF under a prefix: indeed, the subsume and funmatch implementations as shown in the original HMF paper [Leijen 2008,Fig.6&8] closely match the function match rules that we show here. As argued in the introduction, if we consider the complex polymorphic weight condition in the [app-hmf] rule, it can be considered somewhat artificial and quite far removed from the relatively straightforward implementation based on local matching.

We believe that stating both FreezeML and HMF using common prefix inference rules also makes the relation between the two more clear – HMF disambiguates instantiation at applications by inspecting the expected parameter type, while FreezeML disambiguates syntactically at variable

$$\boxed{\underset{\underset{\text{out}}{\downarrow}}{Q} \mid \underset{\underset{\text{in}}{\uparrow}}{\Gamma} \vdash \underset{\underset{\text{in}}{\uparrow}}{e} \overset{\leftarrow}{:} \underset{\underset{\text{in}}{\uparrow}}{\sigma}} \quad \text{with } \vDash Q \qquad \frac{Q \mid \Gamma \vdash e \overset{\leftarrow}{:} \sigma}{Q \mid \Gamma \vdash (e : \sigma) : \sigma}\,\textsc{ann} \qquad \frac{Q_1 \mid \Gamma \vdash e : \tau_1 \quad Q_2 \vdash \tau_1 \approx \tau}{Q_1, Q_2 \mid \Gamma \vdash e \overset{\leftarrow}{:} \tau}\,\textsc{chk}$$

$$\frac{Q \mid \Gamma, x : \tau_1 \vdash e \overset{\leftarrow}{:} \tau_2}{Q \mid \Gamma \vdash \lambda x.e \overset{\leftarrow}{:} \tau_1 \to \tau_2}\,\textsc{func} \qquad \frac{Q \mid \Gamma \vdash e \overset{\leftarrow}{:} \sigma \quad \alpha \notin \mathrm{ftv}(Q, \Gamma, e)}{Q \mid \Gamma \vdash e \overset{\leftarrow}{:} \forall \alpha.\sigma}\,\textsc{genc}$$

$$\frac{Q_1 \mid \Gamma \vdash e_1 \overset{\leftarrow}{:} Q_2[\tau_2] \to \tau \quad Q_2 \mid \Gamma \vdash e_2 : \tau_2}{Q_1, Q_2 \mid \Gamma \vdash e_1\ e_2 \overset{\leftarrow}{:} \tau}\,\textsc{app-func} \qquad \frac{Q \mid \Gamma \vdash e_1\ e_2 \overset{\leftarrow}{:} \alpha \quad \text{fresh } \alpha}{Q \mid \Gamma \vdash e_1\ e_2 : \alpha}\,\textsc{app-chk}$$

$$\frac{Q_1 \mid \Gamma \vdash e_1 \overset{\leftarrow}{:} \beta \to \tau \quad Q_2 \mid \Gamma \vdash e_2 \overset{\leftarrow}{:} Q_1[\beta] \quad \text{fresh } \beta}{Q_1, Q_2 \mid \Gamma \vdash e_1\ e_2 \overset{\leftarrow}{:} \tau}\,\textsc{app-argc}$$

Fig. 6. Bidirectional type checking rules

occurrences relying on syntax directed rules.

5 Bidirectional Inference under a Prefix

Almost all type inference systems in practice use a form of bidirectional type inference [Odersky et al. 2001; Pierce and Turner 2000] where type information is not only inferred, but also propagated up to the leaves of a derivation. One advantage is to improve type error messages, but often it is used to enable type system extensions. For example, this technique can be used to check higher-ranked types [Odersky and Läufer 1996; Peyton Jones et al. 2007]. It is straightforward to add bidirectional type rules to inference under a prefix as well as shown in Figure 6.

The checking judgement $Q \mid \Gamma \vdash e \overset{\leftarrow}{:} \sigma$ states that an expresion e can be *checked* to have (the input) type σ under a given environment Γ and (output) prefix Q. The [ann] rule switches from inference mode to checking mode with a given type annotation σ. Dually, we can always apply the [chk] rule to switch from checking mode to inference mode where we use the type equivalence relation to ensure the inferred type τ_1 matches the checked type τ_2. The [func] rule splits a checked function type to bind the parameter type directly and propagate the result type to the body. The rule [genc] instantiates progagated polymorphic types.

For checking applications $e_1\ e_2$ there is a choice: we can either first infer the type of the argument and use that to check the function type ([app-func]), or we can first infer the type of the function and use that to check the type of the argument ([app-argc]). The rule [app-func] is straightforward and just propagates the inferred type of the argument τ_2 into the function type. For [app-argc] we use a fresh type β as a place holder for the argument type, and check if e_1 is a function $\beta \to \tau$. Here, we propagate just the information that e_1 must be a function with result type τ where we use β to be able to refer to the (inferred!) expected type of the argument. We propagate this type to check the argument as $Q_1[\beta]$. This is somewhat similar to boxy type inference [Vytiniotis et al. 2006] where one would check the function type as $\boxed{\tau_2} \to \tau$ where the boxed τ_2 represents inferred type information that cannot be used for checking – in our prefix based system such boxes are handled by abstract (fresh) type variables.

The new checking rules for applications can now be used to replace the inference rule for application with [app-chk] where we just propagate a fresh result type α. We have now neatly separated out different parts of the original [APP] rule: the creation of a fresh result type α in [APP-CHK], the inference of the argument in [APP-FUNC], and finally the equivalence of the function type τ_1 to $\tau_2 \rightarrow \alpha$ using [CHK] (combined with the use of the checking judgment in [APP-FUNC]).

The application checking rules are not syntax-directed though – which rule should we apply in practice? This choice is not so clear cut [Dunfield and Krishnaswami 2021]; usually it is considered best to use [APP-ARGC] to propagate type information into the argument expression [Peyton Jones et al. 2007; Pierce and Turner 2000] but this is not always the case, and it depends on intended usage (and we discuss this in more detail in Section 6.3). In particular, at the moment our checked type rules do not *do* anything, and just propagate known type information. At this point, these rules can only improve type error messages in practice. In Section 6 though we look at a checking rule for variables that actually takes the propagated type information into account.

6 Static Overloading

After reconsidering existing systems like FreezeML and HMF in Section 4 in terms of inference under a prefix, we now take a look at a novel application where we rely on prefixes to disambiguate variables for *static overloading*. For example, we would like to write $\lambda x\ y.(x + 1, y + 1.0)$ and have the (+) operations resolve to integer- and floating point addition respectively. One elegant solution to overloading is the use of type classes [Wadler and Blott 1989]. Even though type classes are very expressive and highly succesful in languages like Haskell and Lean, they are also a complex extension that changes the semantics of types, and require sophisticated constraint solving of type instance relations [Selsam et al. 2020; Vytiniotis et al. 2010] with many possible design choices from a language perspective [Jones and Diatchki 2008; Peyton Jones et al. 1997].

6.1 Overloading as Disambiguation

Instead, we consider a much simpler alternative here, and look at the most basic form of overloading where we only disambiguate statically between different known versions of an overloaded function f based on the local type context. This form of static overloading is quite common and for example used in the C language to overload various arithmetic operations to work over integers and floats.

For our purposes, we allow a function f to be defined with a qualified name, like *modi/f*, which allows multiple definitions for f in different modules or namespaces. For example, we could have:

$$\begin{array}{l} \mathit{modi/show} \ :\ \mathit{int} \longrightarrow \mathit{string} \ =\ \ldots \\ \mathit{modb/show} \ :\ \mathit{bool} \longrightarrow \mathit{string} \ =\ \ldots \end{array}$$

Generally, such qualified names can come from definitions in different imported modules, but we may also directly allow programmers to use qualified names when defining functions (as if the function is defined inside a mini-module). Note that these kinds of qualified names already occur naturally in any language with namespaces or modules, and languages already need some mechanism to deal with ambiguity: if one imports module *modi* and *modb* that both export the *show* definition, to which definition should an unqualified *show* refer to? In Haskell for example, one needs to use a fully qualified name to disambiguate.

Another advantage of using qualified names is that it does not require an upfront declaration of the variables which can be overloaded, and we can always refer directly to each definition by explicitly using their unique fully qualified name. As such we can view static overloading as a source-to-source translation that only disambiguates identifiers to their fully qualified name.

The idea is now to use static type information at a call site to allow a programmer to write an unqualified name, like *show*, and have it be disambiguated automatically to the full qualified name depending on the type context. For example, *show* 1 is disambiguated to *modi/show* 1 since it is used with an argument of type *int*. In contrast to type classes, static overloading rejects programs where a variable cannot be disambiguated uniquely, like $\lambda x.\ \mathit{show}\ x$ for example. This is of course a severe restriction as it prohibits abstraction over overloaded variables. However, Following Lewis et al. [2000], we believe such an abstraction should be a separate and orthogonal concept, where we use implicit parameters in combination with static overloading. We come back to this at the end of this Section where we discuss the implementation of static overloading in Koka language.

Even in this restricted form, static overloading can be quite useful in practice as it handles many common cases of first-order overloading. However, even though the idea is simple, it clearly does not work well with standard HM inference. If we consider $\lambda x.\ show\ x$ again, we can "guess" the type *int* for the lambda-bound x parameter, and in that case we can accept the expression and disambiguate to *modi/show* – or guess type *bool* and elaborate to *modb/show* instead. Similarly, if we allow non-principal types for let-bindings we can also derive different disambiguations.

6.2 Bidirectional Disambiguation

If we use inference under a prefix though, we avoid all these problems since parameter types are no longer guessed, and let-bindings have a principal type by construction. For example, the expression $\lambda x.\ show\ x$ is always rejected now since we cannot disambiguate on the abstract type variable that is assigned to x. It turns out that extending HMQ with static overloading is quite straightforward where we mainly need to extend the bidirectional rules of Figure 6 with a case for variables:

$$\dfrac{\text{unique } m/x : \sigma \in \Gamma \ \text{ with } \ Q \vdash \sigma \sqsubseteq \tau}{Q \mid \Gamma \vdash x \overset{\leftarrow}{:} \tau \rightsquigarrow m/x}\,\textsc{varc} \qquad \dfrac{Q \vdash \tau_1 \approx \tau_2 \quad \text{fresh } \overline{\alpha}}{Q \vdash \forall\overline{\alpha}.\tau_1 \sqsubseteq \tau_2}\,\textsc{instancec}$$

We use unique notation in the [varc] rule to mean: "there exists a unique $m/x : \sigma \in \Gamma$, where σ can be instantiated to τ (with $Q \vdash \sigma \sqsubseteq \tau$), and for all other $m'/x : \sigma' \in \Gamma$ with $m \neq m'$, $Q' \vdash \sigma' \sqsubseteq \tau$ does not hold". The [instancec] rule checks if a type $\forall\overline{\alpha}.\tau_1$ can be instantiated to a given type τ_2. This can be done directly by using the equivalence relation with fresh types for $\overline{\alpha}$ (as in the [inst] rule). The bidirectional type rules provide the type information τ required to disambiguate overloaded variables. For example, we can derive the type of *show* 1 as:

$$\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{l}\text{unique } modi/show : int \to string \in \Gamma \\ \text{with } \{\alpha = string\} \vdash int \to string \sqsubseteq int \to \alpha\end{array}}{\{\alpha = string\} \mid \Gamma \vdash show \overset{\leftarrow}{:} int \to \alpha \rightsquigarrow modi/show}\,\textsc{varc} \qquad \dfrac{}{\varnothing \mid \Gamma \vdash 1 : int}\,\textsc{int}}{\{\alpha = string\} \mid \Gamma \vdash show\ 1 \overset{\leftarrow}{:} \alpha \qquad \text{fresh } \alpha}\,\textsc{app-func}}{\{\alpha = string\} \mid \Gamma \vdash show\ 1 : \alpha}\,\textsc{appc}}{\varnothing \mid \Gamma \vdash show\ 1 : string}\,\textsc{gensub}$$

Extending HMQ with static overloading, essentially as just disambiguation over qualified names, is as simple as shown here – and the [varc] and [instancec] rules are also straightforward to implement. However, to ensure that all overloaded variables are always resolved uniquely we need to use syntax-directed bidirectional type rules (as shown in Figure 13 in App. D.10 of the techreport), where we in particular disallow [chk] (or otherwise we could always choose to switch to inference mode for a variable which cannot resolve overloading). Moreover, we need to reconsider how to check applications since both [app-func] and [app-argc] could apply.

6.3 Arguments First versus Functions First

In the previous example *show* 1 we used the [app-func] rule to push the type of the argument into the function derivation in order to resolve *show* using [varc]. However, sometimes we need to do the opposite and push the function type into the argument in order to disambiguate. Suppose we have an overloaded definition of *neg* as:

$$\begin{array}{l} \mathit{modi/neg} \;:\; \mathit{int} \rightarrow \mathit{int} \\ \mathit{modf/neg} \;:\; \mathit{float} \rightarrow \mathit{float} \end{array}$$

with $\mathit{sqrt} : \mathit{float} \rightarrow \mathit{float}$. If we now consider the expression $\lambda x.\ \mathit{sqrt}\ (\mathit{neg}\ x)$ we can only accept this if we use [APP-ARGC] on the application $\mathit{sqrt}\ (\mathit{neg}\ x)$ to propagate the *float* result type into the argument expression $\mathit{neg}\ x$. Otherwise, if we use [APP-FUNC] we cannot disambiguate the *neg* variable (since x has an abstract type at that point).

The optimal choice between using [APP-ARGC] or [APP-FUNC] cannot be made locally at an application node and depends on the sub-expressions. A straightforward implementation that tries all combinations would be exponential in the number of nested application nodes. Instead, following the "Pfenning recipe" [Dunfield and Krishnaswami 2021], we propose a syntax-directed approach that can be decided locally and can be easily understood by the programmer. This is also the approach used in the Koka language [Leijen 2021].

In particular, we will disallow [APP-FUNC], and always use [APP-ARGC] where we propagate the expected argument types into the arguments. The only exception is for a direct n-ary application to a variable of the form $f\ e_1 \ldots e_n$. In that case, we infer the least amount of arguments i such that we can disambiguate f, and then propagate the remaining argument types into the remaining argument expressions:

$$\begin{array}{l} \text{least } i \text{ with } 0 \leqslant i \leqslant n \text{ and fresh } \alpha_{i+1}, \ldots, \alpha_n \\ \text{unique } m/f : \sigma \in \Gamma \text{ with } Q \vdash \sigma \sqsubseteq Q_1[\tau_1] \rightarrow \ldots \rightarrow Q_i[\tau_i] \rightarrow \alpha_{i+1} \rightarrow \ldots \rightarrow \alpha_n \rightarrow \tau \end{array}$$
$$\frac{Q_1 \mid \Gamma \vdash e_1 : \tau_1 \;\ldots\; Q_i \mid \Gamma \vdash e_i : \tau_i \quad Q_{i+1} \mid \Gamma \vdash e_{i+1} \overset{\leftarrow}{:} Q[\alpha_{i+1}] \;\ldots\; Q_n \mid \Gamma \vdash e_n \overset{\leftarrow}{:} Q[\alpha_n]}{Q, Q_1, \ldots, Q_n \mid \Gamma \vdash f\ e_1 \ldots e_i \ldots e_n \overset{\leftarrow}{:} \tau}\ \text{APPN}$$

This strategy is straightforward to implement: first try to disambiguate f (without any inference of the arguments) and keep inferring one argument at a time until f can be disambiguated, and then use checking rules for the remaining arguments. There are two drawbacks to this approach: a left-to-right bias, and argument types are never propagated into a lambda expression. As an example of the left-to-right bias, consider the following definitions:

$$\begin{array}{l} \mathit{modi/add} \;:\; \mathit{int} \rightarrow \mathit{int} \rightarrow \mathit{int} \\ \mathit{modf/add} \;:\; \mathit{float} \rightarrow \mathit{float} \rightarrow \mathit{float} \end{array}$$

The expression $\lambda x.\ \mathit{add}\ 1\ (\mathit{neg}\ x)$ can be accepted by [APPN] since after inferring the type of 1, *add* is resolved to *modi/add* and the *int* type is propagated into the $\mathit{neg}\ x$ argument which can subsequently be disambiguated to $\mathit{modi/neg}\ x$. However, the expression $\lambda x.\ \mathit{add}\ (\mathit{neg}\ x)\ 1$ is not accepted since the type of $\mathit{neg}\ x$ cannot be inferred (as *neg* cannot be uniquely disambiguated). Secondly, since we otherwise always prefer to propagate types into arguments, no argument type is progagated into a lambda expression. For example $(\lambda x.\ \mathit{show}\ x)\ 1$ is rejected.

We believe though that having an easy rule for type propagation is preferable to trying to maximise the accepted programs, and the current rule seems to work out well in practice within the Koka language. Nevertheless, further experience may be warranted and other design approaches may be valid as well. For example, following Serrano et al. [2020], instead of strictly inferring from left-to-right we may first take a "quick look" at all expressions and infer "easy" expressions first. Or following Xie and Oliveira [2018], if the function expression in an application is syntactically a lambda expression, we could choose to propagate the argument types into the function.

As discussed in Section 3.3, in an implementation of HMQ we need to be careful to not leak type information between separate sub derivations. The example given was $\lambda x.\ (\mathit{inc}\ x,\ \mathit{show}\ x)$, where *inc* has type $int \to int$. This expression should be rejected since x will have an abstract type in each derivation of *inc* x and *show* x and thus *show* cannot be disambiguated. However, if we naively use algorithm W, the type of $x : \alpha$ is substituted after checking *inc* x to $x : int$, and subsequently *show* x can be disambiguated! When using HMQ extended with static overloading, it is important to use algorithm WQ (or the direct algorithm *inferD*) which uses fresh type variables for each occurrence of a λ-bound parameter (and correctly rejects the example expression).

The full syntax-directed bidirectional rules, including [APPN], are given in App. D.10 of the techreport. There we also show that for any possible derivation, each overloaded identifier is always resolved uniquely. These are also the rules used to implement static overloading in the Koka language [Leijen 2021]. Koka allows *locally qualified* names where one can directly define both `fun int/show` and `fun bool/show` within a module and use a plain `show` to resolve to either one. This works especially well in combination with the *syntactic implicits* feature of Koka that can be used for abstraction. This can express many examples that are typically addressed by type classes but in an arguably simpler way. For example, one can define a `list/show` function as:

```
fun list/show( xs : list<a>, ?show : a -> string ) : string
  match xs
    Cons(x,Nil) -> show(x)                    // ~> ?show(x)
    Cons(x,xx)  -> show(x) ++ "," ++ show(xx)  // ~> ?show(x) ++ "," ++ list/show(xx)
    Nil         -> ""
```

An implicit parameter `?show` in Koka is just a parameter whose syntactic name is resolved at each lexical call site, where regular static overloading is used to disambiguate based on the type context. For example, `show([1,2])` elaborates to `list/show([1,2],int/show)`.

7 Related Work

Damas and Milner [1982] introduce the now common HM type rules and show that type inference with algorithm W is sound and complete. This work builds on earlier work by Hindley [1969], who shows principal types exist for objects in combinatory logic, and Milner [1978] who gives the first description of algorithm W.

Prefixes. As discussed in Section 2.7, the main idea of inference under a prefix comes from the work on impredicative type inference in MLF as described by Le Botlan and Rémy [2003]. The prefix in MLF is much richer though and contains both polymorphic rigid bounds, $\alpha{=}\sigma$, and polymorphic flexible bounds $\alpha \geqslant \sigma$, where α can be any instance of σ. We can also quantify over bounds as $\forall Q.\sigma$ which, as shown in Section 2.7, could be useful for HMQ as well – as it allows us to unify both generalization rules into a single one. MLF is still an HM style system though where the type of λ-bound parameters is "guessed". We believe it should be possible to extend HMQ naturally to MLFQ by extending the prefix to contain rich MLF bounds, and the equivalence relation to MLF equivalence. Leijen [2009] describes a restriction of MLF to only use flexible bounds which would lend itself well to a HMQ extension as it simplifies unification between polymorphic types.

Gundry, McBride, and McKinna [2010] describe type inference under a *context* Θ defined as:

$\Theta ::= \varnothing \mid \Theta, \alpha : * \mid \Theta, \alpha{:=}\tau : * \mid \Theta, x : \sigma \mid \Theta ⨾$

where $\alpha : *$ can be viewed as an MLF instance constraint $\alpha {\geqslant} \bot$, and where $\alpha{:=}\tau : *$ corresponds to our $\alpha{=}\tau$ bindings. The other forms are environment bindings $x : \sigma$ and ordering constraints ⨾. A context restricted to just $\alpha : *$ and $\alpha{:=}\tau : *$ bindings is written as Ξ, which can be viewed as a dependency ordered prefix. Indeed, the generalization rule is defined as [Gundry 2013,Fig. 2.9]:

$$\frac{\Theta_0 ⨾ \vdash\; e : \tau \;\dashv \Theta_1 ⨾ \Xi}{\Theta_0 \vdash\; e : \forall \Xi.\tau \;\dashv \Theta_1}\text{GEN-CTX} \qquad \begin{array}{lcl} \forall \varnothing.\tau & = & \tau \\ \forall(\alpha : *, \Xi).\tau & = & \forall \alpha.(\forall \Xi.\tau) \\ \forall(\alpha{:=}\tau' : *, \Xi).\tau & = & [\alpha{:=}\tau'](\forall \Xi.\tau) \end{array}$$

which corresponds closely to the [GENX] rule of Section 2.7 (and the corresponding MLF generalization rule) where we quantify over a prefix, written here as $\forall\Xi.\tau$, where all monomorphic bounds are substituted. The main idea of having dependency ordered contexts is to simplify generalization where there is no need in the [GEN-CTX] rule to compute the free type variables in the environment (similar to using level-based generalization [Kiselyov 2022; Kuan and MacQueen 2007; Rémy 1992]).

The (algorithmic) application rule also closely matches our [APP] rule:

$$\frac{\Theta_0 \vdash e_1 : \tau_1 \dashv \Theta_1 \quad \Theta_1 \vdash e_2 : \tau_2 \dashv \Theta_2 \quad \Theta_2 \vdash \tau_1 \equiv \tau_2 \rightarrow \alpha \dashv \Theta_3 \quad \text{fresh } \alpha}{\Theta_0 \vdash e_1\ e_2 : \alpha \dashv \Theta_3}\textsc{app-ctx}$$

Here the context is statefully threaded through the rules but we believe it should be possible to define consistent context composition similar to our prefix composition such that sub derivations can be composed independently.

Constraint Based Inference. Type inference based on constraint generation has many similarities to the prefix based approach. These systems generate sets of unification constraints of the form $\tau_1 \equiv \tau_2$. Pierce [2002,§22.3] describes constraint based inference for a monomophic calculus with essentially the following abstraction and application rules:

$$\frac{C \mid \Gamma, x:\tau_1 \vdash e : \tau_2}{C \mid \Gamma \vdash \lambda x.e : \tau_1 \rightarrow \tau_2}\textsc{fun-con} \qquad \frac{C_1 \mid \Gamma \vdash e_1 : \tau_1 \quad C_2 \mid \Gamma \vdash e_2 : \tau_2 \quad \text{fresh } \alpha}{C_1 \cup C_2 \cup \{\tau_1 \equiv \tau_2 \rightarrow \alpha\} \mid \Gamma \vdash e_1\ e_2 : \alpha}\textsc{app-con}$$

Their [APP-CON] is quite similar to our [APP] rule, except that the constraint $\{\tau_1 \equiv \tau_2 \rightarrow \alpha\}$ is directly included while in HMQ one derives a prefix Q_3 from the equivalence relation $Q_3 \vdash \tau_1 \approx \tau_2 \rightarrow \alpha$. In that sense, a prefix is a restricted form of a general constraint set which makes it closer to an implementation based on (in-place) substitutions. Just like the standard HM type rules, the constraint based system of Pierce still "guesses" types for lambda bound parameters.

In contrast, Heeren, Hage, and Swierstra [2002] describe a bottom-up constraint based system which uses abstract fresh variables for lambda-bound parameters. As part of the bottom-up inference, there is no top-down Γ environment, but instead a bottom-up *assumption* environment A. The abstraction, variable, and application rules are [Heeren 2005, Fig. 4.5]:

$$\frac{M \cup \{\alpha\} \mid C \mid A \vdash e : \tau \quad \text{fresh } \alpha}{M \mid C \cup \{\alpha \equiv \tau' \mid x:\tau' \in A\} \mid A/x \vdash \lambda x.e : \alpha \rightarrow \tau}\textsc{fun-bu} \qquad \frac{\text{fresh } \alpha}{M \mid \varnothing \mid \{x:\alpha\} \vdash x : \alpha}\textsc{var-bu}$$

$$\frac{M \mid C_1 \mid A_1 \vdash e_1 : \tau_1 \quad M \mid C_2 \mid A_2 \vdash e_2 : \tau_2 \quad \text{fresh } \alpha}{M \mid C_1 \cup C_2 \cup \{\tau_1 \equiv \tau_2 \rightarrow \alpha\} \mid A_1 \cup A_2 \vdash e_1\ e_2 : \alpha}\textsc{app-bu}$$

(where M is the set of monomorphic type variables used for the generation of generalization constraints in the let-rule). These bottom-up algorithmic type rules are closer to HMQ as the types of the lambda-bound parameters are abstract. Moreover, the use of an assumption together with the [VAR-BU] and [FUN-BU] rules is also close to algorithm WQ (Section 3.3) where we use fresh type variables for each occurrence and unify them all eventually at the λ expression, corresponding to the $\{\alpha \equiv \tau' \mid x:\tau' \in A\}$ constraint set in [FUN-BU]. However, depending on how type constraints $\tau_1 \equiv \tau_2$ are resolved, further restrictions may be needed to ensure all let-bindings have a principal type. With the addition of those restrictions, we believe that Heeren's bottom-up algorithm can be a valid implementation for HMQ.

Unifying Substitutions. McAdam [1999] describes a new inference algorithm W' which does not have a right-to-left bias by computing substitutions for each subderivation independently and afterwards unifying the substitutions.

$$\boxed{\begin{array}{ccccc} \Delta \mid & Q \mid & \Gamma \vdash & e \; : & \sigma \\ \downarrow & \downarrow & \uparrow & \uparrow & \downarrow \\ \text{out} & \text{out} & \text{in} & \text{in} & \text{out} \end{array}} \quad \text{with } \vDash Q \text{ and } \Delta \not\pitchfork \mathrm{ftv}(\Gamma)$$

$$\frac{x : \sigma \in \Gamma}{\varnothing \mid \varnothing \mid \Gamma \vdash x : \sigma}\textsc{var} \qquad \frac{\Delta \mid Q \mid \Gamma \vdash e : \forall \alpha. \sigma}{\Delta, \alpha \mid Q \mid \Gamma \vdash e : \sigma}\textsc{inst} \qquad \frac{\Delta, \alpha \mid Q \mid \Gamma \vdash e : \sigma \quad \alpha \notin \mathrm{ftv}(Q)}{\Delta \mid Q \mid \Gamma \vdash e : \forall \alpha. \sigma}\textsc{gen}$$

$$\frac{\Delta \mid Q \mid \Gamma, x : \alpha \vdash e : \tau}{\Delta, \alpha \mid Q \mid \Gamma \vdash \lambda x.\, e : \alpha \to \tau}\textsc{fun} \qquad \frac{\Delta, \alpha \mid Q \cdot \alpha{=}\tau \mid \Gamma \vdash e : \sigma}{\Delta \mid Q \mid \Gamma \vdash e : [\alpha{:=}\tau]\sigma}\textsc{gensub}$$

$$\frac{\Delta_1 \mid Q_1 \mid \Gamma \Vdash e_1 : \sigma \quad \Delta_2 \mid Q_2 \mid \Gamma, x : \sigma \vdash e_2 : \tau}{\Delta_1, \Delta_2 \mid Q_1, Q_2 \mid \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau}\textsc{let} \qquad \frac{\Delta \mid Q \mid \Gamma \vdash e : \sigma \quad \mathrm{ftv}(\sigma) \subseteq \mathrm{ftv}(\Gamma)}{\Delta \mid Q \mid \Gamma \Vdash e : \sigma}\textsc{mgen}$$

$$\frac{\Delta_1 \mid Q_1 \mid \Gamma \vdash e_1 : \tau_1 \quad \Delta_2 \mid Q_2 \mid \Gamma \vdash e_2 : \tau_2 \quad Q_3 \vdash \tau_1 \approx \tau_2 \to \alpha}{\Delta_1, \Delta_2, \alpha \mid Q_1, Q_2, Q_3 \mid \Gamma \vdash e_1\; e_2 : \alpha}\textsc{app}$$

Fig. 7. Full HMQ type rules under a prefix, using explicit fresh names Δ, where we write Δ_1, Δ_2 for the union $\Delta_1 \cup \Delta_2$ where the elements are disjoint ($\Delta_1 \not\pitchfork \Delta_2$). By construction, we also have $\mathrm{ftv}(Q, \sigma) \subseteq \mathrm{ftv}(\Delta, \Gamma)$.

This is very similar how HMQ uses the notion of a consistent union of prefixes. If we keep all prefixes as an idempotent mapping, we can use McAdam's U_s algorithm directly to compute the consistent union of two prefixes (as shown in Section 3.1.2).

8 HMQ Type Rules with Explicit Fresh Names

Figure 7 gives full inductive type rules for HMQ using explicit fresh names Δ. The fresh α notation used in Figure 2 is essentially a convenient shorthand for the rules here with explicit names. We write Δ_1, Δ_2 for the disjoint union of Δ_1 and Δ_2, where $\Delta_1, \Delta_2 \doteq \Delta_1 \cup \Delta_2$ with $\Delta_1 \not\pitchfork \Delta_2$. Every time we used fresh α in the rules in Figure 2, we now pick a fresh α disjoint from existing fresh names Δ (as Δ, α) in [inst], [fun], and [app]. As in the original rules, we rely on α-renaming in the [inst] rule such that the quantifier matches the fresh name. Such introduced fresh names are all eventually consumed by the generalization rules [gen] and [gensub] that have Δ, α in their premise. A well-formedness condition for the new rules is that Δ is disjoint from the free type variables in the environment, with $\Delta \not\pitchfork \mathrm{ftv}(\Gamma)$. This ensures that for every derivation the fresh names are indeed fresh and do not contain type variable names occurring in the environment.

Lemma 8.10. (*Output type variables are either fresh or occur free in the environment*)
If $\Delta \mid Q \mid \Gamma \vdash e : \sigma$, then $\mathrm{ftv}(Q, \sigma) \subseteq \mathrm{ftv}(\Delta, \Gamma)$.

The rules with explicit fresh names no longer need the full conditions on [gen] and [gensub] as in Figure 2. For both rules, the $\alpha \notin \mathrm{ftv}(\Gamma)$ requirement is now implied by the Δ, α premise (and the well-formedness condition $\Delta, \alpha \not\pitchfork \Gamma$), while for [gensub], $\alpha \notin \mathrm{ftv}(Q)$ is implied by the $Q \cdot \alpha{=}\tau$ premise. The need for fresh names in HMQ is not ideal, and one might have hoped to see a local condition instead, for example:

$$\frac{Q \mid \Gamma \vdash e : \forall \alpha. \sigma \quad \alpha \notin \mathrm{ftv}(Q, \Gamma)}{Q \mid \Gamma \vdash e : \sigma}\textsc{inst-wrong}$$

Unfortunately, such local constraints still allow the introduction of artifical sharing by using the same variable name in separate sub-derivations, which eventually leads to non-principal derivations again. Consider for example *const id* 1 with $const : \forall\alpha\beta.\ \alpha \to \beta \to \alpha$. If we instantiate *const* to $\alpha \to \beta \to \alpha$, we could instantiate the quantifier for *id* to also be β (if we can use [INST-WRONG]):

$$\dfrac{\dfrac{\varnothing \mid \Gamma \vdash const : \alpha{\to}\beta{\to}\alpha \quad \varnothing \mid \Gamma \vdash id : \beta{\to}\beta \quad \ldots \vdash \alpha{\to}\beta{\to}\alpha \approx (\beta{\to}\beta) \to \gamma}{\{\alpha{=}\beta{\to}\beta, \gamma{=}\beta{\to}\alpha\} \mid \Gamma \vdash const\ id : \gamma}\textsc{app}}{\varnothing \mid \Gamma \vdash const\ id : \beta{\to}(\beta{\to}\beta)}\textsc{gensub}$$

and thus *const id* 1 gets type $int{\to}int$ instead of the expected $\forall\alpha.\alpha{\to}\alpha$. The rules in Figure 7 ensure that separate sub-derivations all use unique names by using a disjoint union of names used in each sub-derivation (and thus, we cannot instantiate *const* and *id* with a shared β as in our example).

9 Conclusion

Type inference under a prefix gives us declarative type rules that we believe are close to the clarity of the original HM rules. At the same time, we are able to "read off" the algorithm from the declarative type rules. HMQ can serve as foundation to specify practical type systems in a declarative way that serves both purposes: users can easily reason about what programs are accepted by the type checker, while compiler writers can derive sound implementations from those same rules.

Acknowledgements

We would like to thank Tim Whiting for his feedback and many discussions about the design of static overloading with implicit parameters in Koka.

References

Damas, and Milner. 1982. Principal Type-Schemes for Functional Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 207–212. POPL'82. ACM, Albuquerque, New Mexico. doi:https://doi.org/10.1145/582153.582176.

Dunfield, and Krishnaswami. May 2021. Bidirectional Typing. *ACM Comput. Surv.* 54 (5). ACM. doi:https://doi.org/10.1145/3450952.

Emrich, Lindley, Stolarek, Cheney, and Coates. 2020. FreezeML: Complete and Easy Type Inference for First-Class Polymorphism. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 423–437. PLDI 2020. ACM, London, UK. doi:https://doi.org/10.1145/3385412.3386003.

Emrich, Stolarek, Cheney, and Lindley. Aug. 2022. Constraint-Based Type Inference for FreezeML. *Proc. ACM Program. Lang.* 6 (ICFP). ACM press. doi:https://doi.org/10.1145/3547642.

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

Gundry. 2013. Type Inference, Haskell and Dependent Types. Phdthesis, University of Strathclyde, Department of Computer and Information Sciences.

Gundry, McBride, and McKinna. 2010. Type Inference in Context. In *Proceedings of the Third ACM SIGPLAN Workshop on Mathematically Structured Functional Programming*, 43–54. MSFP'10. ACM, Baltimore, Maryland, USA. doi:https://doi.org/10.1145/1863597.1863608.

Heeren. Sep. 2005. Top Quality Type Error Messages. Phdthesis, Institute of Information and Computing Sciences, Utrecht University. `https://dspace.library.uu.nl/bitstream/handle/1874/7297/full.pdf`.

Heeren, Hage, and Swierstra. 2002. *Generalizing Hindley-Milner Type Inference Algorithms*. UU-CS-2002-031. Institute of Information and Computing Sciences, Utrecht University. `https://ics-archive.science.uu.nl/research/techreps/repo/CS-2002/2002-031.pdf`.

Heeren, Leijen, and IJzendoorn. 2003. Helium, for Learning Haskell. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, 62–71. Haskell'03. ACM, Uppsala, Sweden. doi:https://doi.org/10.1145/871895.871902.

Hindley. 1969. The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society* 146. American Mathematical Society: 29–60.

Jones, and Diatchki. 2008. Language and Program Design for Functional Dependencies. In *Proc. of the First ACM SIGPLAN Symp. on Haskell*, 87–98. Haskell'08. ACM, Victoria, BC, Canada. doi:https://doi.org/10.1145/1411286.1411298.

Kiselyov. 2022. How OCaml Type Checker Works – or What Polymorphism and Garbage Collection Have in Common. `https://okmij.org/ftp/ML/generalization.html`. Blog post.

Kuan, and MacQueen. 2007. Efficient Type Inference Using Ranked Type Variables. In *Proc. of the ML Workshop*, 3–14. Freiburg, Germany. doi:https://doi.org/10.1145/1292535.1292538.

Le Botlan. 2004. MLF: Une Extension de ML Avec Polymorphisme de Second Ordre et Instanciation Implicite. Phdthesis, l'école polytechnique, Paris.

Le Botlan, and Rémy. 2003. MLF: Raising ML to the Power of System F. In *Proc. of the 8th ACM SIGPLAN Int. Conf. on Functional Programming*, 27–38. ICFP'03. ACM press, Uppsala, Sweden. doi:https://doi.org/10.1145/944705.944709.

Leijen. Sep. 2007. *HMF: Simple Type Inference for First-Class Polymorphism.* MSR-TR-2007-118. Microsoft Research.

Leijen. Sep. 2008. HMF: Simple Type Inference for First-Class Polymorphism. In *Proc. of the 13th ACM Symp. of the Int. Conf. on Functional Programming*. ICFP'08. Victoria, Canada. doi:https://doi.org/10.1145/1411204.1411245.

Leijen. 2009. Flexible Types: Robust Type Inference for First-Class Polymorphism. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 66–77. POPL'09. ACM, Savannah, GA, USA. doi:https://doi.org/10.1145/1480881.1480891.

Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. In *MSFP'14, 5th Workshop on Mathematically Structured Functional Programming*. doi:https://doi.org/10.4204/EPTCS.153.8.

Leijen. 2021. The Koka Language. `https://koka-lang.github.io`.

Leijen, and Ye. Sep. 2024. *Principal Type Inference under a Prefix – A Fresh Look at Static Overloading*. MSR-TR-2024-34. Microsoft Research.

Leroy, and Mauny. 1993. Dynamics in ML. *Journal of Functional Programming* 3 (4): 431–463. doi:https://doi.org/10.1017/S0956796800000848.

Lewis, Launchbury, Meijer, and Shields. 2000. Implicit Parameters: Dynamic Scoping with Static Types. In *Proc. of the 27th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, 108–118. POPL'00. ACM, Boston, MA, USA. doi:https://doi.org/10.1145/325694.325708.

McAdam. 1999. On the Unification of Substitutions in Type Inference. In *Impl. of Functional Languages*, edited by K. Hammond, T. Davie, and C. Clack, 137–152. Springer, Berlin, Heidelberg. doi:https://doi.org/10.1007/3-540-48515-5_9.

Milner. 1978. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences* 17 (3): 348–375. doi:https://doi.org/10.1016/0022-0000(78)90014-4.

Odersky, and Läufer. 1996. Putting Type Annotations to Work. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 54–57. POPL '96. ACM, St. Petersburg Beach, Florida, USA. doi:https://doi.org/10.1145/237721.237729.

Odersky, Zenger, and Zenger. 2001. Colored Local Type Inference. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 41–53. POPL '01. ACM, London, United Kingdom. doi:https://doi.org/10.1145/360204.360207.

Peyton Jones, Jones, and Meijer. Jan. 1997. Type Classes: An Exploration of the Design Space. In *Haskell Workshop*. `https://www.microsoft.com/en-us/research/publication/type-classes-an-exploration-of-the-design-space/`.

Peyton Jones, Vytiniotis, Weirich, and Shields. 2007. Practical Type Inference for Arbitrary-Rank Types. *Journal of Functional Programming* 17 (1): 1–82. doi:https://doi.org/10.1017/S0956796806006034.

Pierce. Feb. 2002. *Types and Programming Languages (TAPL)*. 1st edition. The MIT Press, Cambridge, Massachusetts 02142.

Pierce, and Turner. Jan. 2000. Local Type Inference. *ACM Trans. Program. Lang. Syst.* 22 (1). ACM: 1–44. doi:https://doi.org/10.1145/345099.345100.

Rémy. 1992. *Extending ML Type System with a Sorted Equational Theory*. Research Report 1766. Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France.

Robinson. Jan. 1965. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM* 12 (1). ACM: 23–41. doi:https://doi.org/10.1145/321250.321253.

Selsam, Ullrich, and Moura. 2020. Tabled Typeclass Resolution. doi:https://doi.org/10.48550/arXiv.2001.04301.

Serrano, Hage, Peyton Jones, and Vytiniotis. Aug. 2020. A Quick Look at Impredicativity. *Proc. ACM Program. Lang.* 4 (ICFP). ACM. doi:https://doi.org/10.1145/3408971.

Vytiniotis, Peyton Jones, and Schrijvers. 2010. Let Should Not Be Generalized. In *Proc. of the 5th ACM SIGPLAN Workshop on Types in Language Design and Impl.*, 39–50. TLDI '10. ACM, Madrid, Spain. doi:https://doi.org/10.1145/1708016.1708023.

Vytiniotis, Weirich, and Peyton Jones. 2006. Boxy Types: Inference for Higher-Rank Types and Impredicativity. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, 251–262. ICFP '06. ACM press, Portland, Oregon, USA. doi:https://doi.org/10.1145/1159803.1159838.

Wadler, and Blott. 1989. How to Make Ad-Hoc Polymorphism Less Ad Hoc. In *Proc. of the 16th ACM SIGPLAN-SIGACT Symp. on Principles of Prog. Lang.*, 60–76. POPL'89. ACM, Austin, Texas, USA. doi:https://doi.org/10.1145/75277.75283.

Wright, and Felleisen. Nov. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115 (1): 38–94. doi:https://doi.org/10.1006/inco.1994.1093.

Xie, and Oliveira. 2018. Let Arguments Go First. Edited by Amal Ahmed. *Programming Languages and Systems*, LNCS, 10801. Springer International Publishing: 272–299. doi:https://doi.org/10.1007/978-3-319-89884-1_10. ESOP'18.