Principal Type Inference under a Prefix

A Fresh Look at Static Overloading

Microsoft Research Technical Report, MSR-TR-2024-34, 2024-09-17 (v1)

DAAN LEIJEN, Microsoft Research, USA
WENJIA YE, University of Hong Kong, 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. This rule allows the type of the parameter to be "guessed", which allows for multiple possible types for functions like the identity function. 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 weakness 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.

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:

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

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 $int \to int$, or $bool \to bool$ 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 $\forall\alpha.\alpha \to \alpha$. 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:

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

where *ids* has the impredicative type $[\forall\alpha.\alpha \to \alpha]$ (i.e. a list of polymorphic identity functions). If *wrapl* is given its most general type, namely $\forall\alpha\beta.\ \alpha \to \beta \to [\beta]$, we can derive the type $\forall\alpha.[\alpha \to \alpha]$ for the body. However, if we use the [FUN] rule to "guess" a less general type for *wrapl*, namely $\forall\alpha.[\alpha] \to \alpha \to [\alpha]$, then we can derive (in HMF) the type $[\forall\alpha.\alpha \to \alpha]$ 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:

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

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. [2020] 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* → *int*
modb/f : *bool* → *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 (and we can use algorithm W [Damas and Milner 1982] 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

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

$$
\begin{array}{rcll}
\Gamma & ::= & x_1 : \sigma_1, \ldots, x_n : \sigma_n & \text{(type environment)} \\
Q & ::= & \{\alpha_1{=}\tau_1, \ldots, \alpha_n{=}\tau_n\} & \text{(prefix)}
\end{array}
\qquad
\frac{\overline{\beta} \not\cap \mathsf{ftv}(\forall\overline{\alpha}.\ \tau)}{\forall\overline{\alpha}.\ \tau \sqsubseteq \forall\overline{\beta}.\ [\overline{\alpha}{:=}\overline{\tau}]\tau}\ \textsc{instance}
$$

Fig. 1. Syntax of types and terms.

σ_2 can be instantiated to σ_1. 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 show for example how we can describe FreezeML and HMF type inference in this system, and use it to formalize static overloading as implemented in the Koka language.

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 describes the syntax of our standard lambda calculus expressions e, mono-types τ, and polymorphic type schemes σ. The [instance] rule gives the general instantiation rule where we write $\sigma_1 \sqsubseteq \sigma_2$ when a type σ_1 can be instantiated to a type σ_2. For example, $\forall\alpha\beta.\ \alpha \rightarrow \beta \sqsubseteq \forall\beta.\ \mathit{int} \rightarrow \beta \sqsubseteq \mathit{int} \rightarrow \mathit{bool}$. Note that we can only instantiate *bound* type variables $\overline{\alpha}$, and not the *free* type variables in σ_1 (written as $\mathsf{ftv}(\forall\overline{\alpha}.\ \tau)$), and in particular, $\forall\alpha.\ \alpha \rightarrow \beta \sqsubseteq \mathit{int} \rightarrow \mathit{bool}$ 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

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 and implications as we go. We start with the [var] and [gen] rules that match the corresponding HM type rules (see Figure 7 in Appendix A) quite closely. the [var] rule reads the type bound to a variable from the type environment, while [gen] generalizes over free type variables that no longer occur in Γ (and Q in our case).

$$\boxed{\underset{\text{out}}{\underset{\downarrow}{Q}} \mid \underset{\text{in}}{\underset{\uparrow}{\Gamma}} \vdash \underset{\text{in}}{\underset{\uparrow}{e}} : \underset{\text{out}}{\underset{\downarrow}{\sigma}}} \quad \text{with } \vDash Q$$

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

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

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

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

Fig. 2. Type rules under a prefix

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 9 in Appendix A.2). For example, we can derive the type of the polymorphic identity function as:

$$\cfrac{\cfrac{\cfrac{x : \alpha \in (\Gamma, x : \alpha)}{\varnothing \mid \Gamma, x : \alpha \vdash x : \alpha}\textsc{var}}{\varnothing \mid \Gamma \vdash \lambda x.\, x : \alpha \to \alpha}\textsc{fun}}{\varnothing \mid \Gamma \vdash \lambda x.\, x : \forall\alpha.\alpha \to \alpha}\textsc{gen}$$

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.

As an aside, we use fresh α notation to create fresh names α, not only such that $\alpha \notin \text{ftv}(Q, \Gamma)$ in the local rule, but also to ensure there is no other occurrence of α as a fresh name in the derivation. We see this as a convenient notation for a more explicit formalization where we pass a fresh name supply using disjoint union for multiple sub-derivations – see Figure 10 in Appendix B for the full rules. However, adding an explicit name supply clutters the rules somewhat while not adding any essential insight so 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

$$\boxed{\begin{array}{ccccc} Q & \vdash & \tau & \approx & \tau \\ \downarrow & & \uparrow & & \uparrow \\ \text{out} & & \text{in} & & \text{in} \end{array}}$$

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

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

Fig. 3. Type equivalence under a prefix.

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 $\mathsf{ftv}(Q)$ as all free type variables in Q, where $\mathsf{ftv}(Q) = \mathsf{dom}(Q) \cup \mathsf{codom}(Q)$. Note that a general prefix is just a collection of type variable bounds, and can for example have 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 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.

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 $Q \vdash \tau_1 \approx \tau_2$ 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

[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 as θ_2, which holds if there exists some substitution θ' such that $\theta_2 = \theta' \circ \theta_1$.

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 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 9 in Appendix A.2). We can now derive a type for the application *inc x* as:

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

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 is 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 $\alpha \notin \mathrm{ftv}(Q', \tau)$:

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

This allows us to split a prefix Q into a bound $\alpha{=}\tau$ and a remaining prefix Q' that does not depend 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], we can finally derive the type of $\lambda x.\ \mathit{inc}\ x$ as:

$$\dfrac{\dfrac{\dfrac{\dfrac{\varnothing \mid \Gamma, x : \alpha \vdash \mathit{inc} : \mathit{int} \to \mathit{int} \qquad \varnothing \mid \Gamma, x : \alpha \vdash x : \alpha \qquad \{\alpha{=}\mathit{int}, \beta{=}\mathit{int}\} \vdash \mathit{int} \to \mathit{int} \approx \alpha \to \beta}{\{\alpha{=}\mathit{int}, \beta{=}\mathit{int}\} \mid \Gamma, x : \alpha \vdash \mathit{inc}\ x : \beta}\ \textsc{app}}{\{\alpha{=}\mathit{int}\} \mid \Gamma, x : \alpha \vdash \mathit{inc}\ x : \mathit{int}}\ \textsc{gensub}}{\{\alpha{=}\mathit{int}\} \mid \Gamma \vdash \lambda x.\ \mathit{inc}\ x : \alpha \to \mathit{int}}\ \textsc{fun}}{\varnothing \mid \Gamma \vdash \lambda x.\ \mathit{inc}\ x : \mathit{int} \to \mathit{int}}\ \textsc{gensub}$$

2.5 Principal Derivations

In the [LET] rule we find a single side condition: $\mathrm{ftv}(\sigma) \subseteq \mathrm{ftv}(\Gamma)$. 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.

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 HMQ rules are always most general, and we can show the rules are sound and complete with respect to the standard HM type rules:

Theorem 2.3. (*Soundness*)
If $Q \mid \Gamma \vdash e : \sigma$ then we also have $Q[\Gamma] \vdash_{\mathsf{HM}} e : Q[\sigma]$.

Theorem 2.4. (*Completeness*)
If $\Gamma \vdash_{\mathsf{HM}} e : \sigma$, 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 also have that algorithm W is a valid type inference algorithm for HMQ (and since W is also complete it infers the same types 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 7 in Appendix A for the definition of $\vdash_{\mathsf{HM}}$). 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 $\Gamma \vdash_{\mathsf{HM}} e : \sigma$, 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, , but in the HM type rules, we can also derive:

$$\dfrac{\dfrac{\dfrac{x{:}\alpha \in (x{:}\alpha)}{\varnothing \mid x{:}\alpha \vdash x : \alpha}\,\text{VAR}}{\varnothing \mid \varnothing \vdash \lambda x.\, x : \alpha \to \alpha}\,\text{LAM}}{\varnothing \mid \varnothing \vdash \lambda x.\, x : \forall\alpha.\alpha{\to}\alpha}\,\text{GEN} \qquad \dfrac{\dfrac{x{:}\mathit{int} \in (x{:}\mathit{int})}{x{:}\mathit{int} \vdash_{\mathsf{HM}} x : \mathit{int}}\,\text{VAR}}{\varnothing \vdash_{\mathsf{HM}} \lambda x.\, x : \mathit{int} \to \mathit{int}}\,\text{LAM}$$

For an inductive proof, it means that for the [VAR] case, we would need to show that if we derive $x{:}\mathit{int} \vdash_{\mathsf{HM}} x : \mathit{int}$, 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 there exists some substitution θ with $\theta\Gamma' \sqsubseteq \Gamma$, then $Q \sqsubseteq \theta$ and $\theta\sigma' \sqsubseteq \sigma$. 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 $\mathit{const} = \lambda x.\lambda y.\, y$ 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 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 actually a bit more general than stated here (see Appendix C.9 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 $\alpha \notin \mathrm{ftv}(\Gamma)$. 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 $\alpha \in \mathrm{ftv}(Q')$). However, for any consistent prefix, we can always simplify multiple bindings:

Theorem 2.5. (*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 in 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, when we would like to extract α from $\{\beta{=}\alpha,\ \alpha{=}\mathit{int}{\to}\mathit{int}\}$ (where $\alpha \in \mathrm{ftv}(\{\beta{=}\alpha\})$). It turns out though that any consistent prefix is always equivalent to an *idempotent* mapping where $\mathrm{dom}(Q) \not\pitchfork \mathrm{codom}(Q)$, e.g. $\{\beta{=}\alpha,\ \alpha{=}\mathit{int}{\to}\mathit{int}\} \equiv \{\beta{=}\mathit{int}{\to}\mathit{int},\ \alpha{=}\mathit{int}{\to}\mathit{int}\}$.

Theorem 2.6. (*Any consistent prefix is equivalent to an idempotent mapping*)
If $\vDash Q$, then there exists an equivalent idempotent mapping Q' (where $Q \equiv Q'$, $|\mathrm{dom}(Q')| = |Q'|$

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

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

Fig. 4. Function matching.

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 $\alpha \notin \text{ftv}(\Gamma)$.

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:

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

We can then use a single generalization rule as:

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

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 to for example 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 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 11 in Appendix B.1 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 3.7. (*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:

$$\begin{array}{lcll} \mathit{solve}(\varnothing) & = & \mathit{id} & \\ \mathit{solve}(Q \cup \{\alpha{=}\tau\} & = & \mathit{solve}(Q) \circ [\alpha{:=}\tau] & \text{if } \alpha \notin \mathsf{ftv}(Q, \tau) \end{array}$$

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 binding (with $\alpha \in \mathsf{dom}(Q)$), and in such case we can use Theorem 2.5 to simplify the duplicate bindings[2]:

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

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

[2] 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 $\mathit{solve}(\{\beta{=}\alpha, \alpha{=}\beta, \alpha{=}\mathit{int}\})$.

derive an implementation, representing prefixes again as substitutions, as:

```
$equiv\ :\ (\tau_1, \tau_2) \longrightarrow \theta$
$equiv\ (\alpha, \alpha) \qquad\qquad\qquad\qquad = \ id$
$equiv\ (\alpha, \tau)$ or $(\tau, \alpha) \mid \alpha \notin \mathsf{ftv}(\tau) = \ [\alpha{:=}\tau]$
$equiv\ (\tau_1{\to}\tau_2,\ \tau_1'{\to}\tau_2')\ =$
  let $\theta_1\ =\ equiv(\tau_1, \tau_1')$
  let $\theta_2\ =\ equiv(\tau_2, \tau_2')$
  $solve(\theta_1 \cup \theta_2)$
```

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 [APP$_S$] application rule becomes:

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

(where we use substitutions θ for idempotent mapping prefixes Q).

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]:

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

where we see that the substitution θ_1 is applied to Γ when checking e_2 (see Figure 9 in Appendix A.2 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, we assume a pre-processing step where we annotate each λ-bound parameter x with the number n of occurrences in the body as x_n, and number all occurrences x in sequence as x_i. For example, our elaborated example expression becomes $\lambda x_2.\ (inc\ x_1,\ show\ x_2)$.

We then modify algorithm W to 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 $\alpha\ =\ \Gamma(x)$
  $(id, \alpha_i)$
```

and at the λ-binding we eventually unify all α_i types of all the occurrences:

```
$inferWQ(\Gamma, \lambda x_n.e)$ =
  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 12 in Appendix B.2. 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 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 such systems, and it is not meant 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:

$$\begin{array}{llll@{\qquad}llll}
\sigma & ::= & \forall\alpha.\sigma & \text{(quantification)} & \rho & ::= & \sigma \to \sigma & \text{(higher-rank function)} \\
 & \mid & \rho & \text{(no outer quantifier)} & & \mid & \tau & \text{(monomorphic types)} \\
 & & & & & \mid & [\sigma] & \text{((impredicative) list of } \sigma\text{)}
\end{array}$$

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

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

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 $\forall\alpha.\alpha - \alpha \sqsubseteq int{\to}int$, 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 in 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-QUANT] rule from escaping into Q, we need the side-condition $\alpha \notin \mathrm{ftv}(Q)$. The new equivalence rules again closely resemble the usual unification algorithm for impredicative types [Emrich et al. 2020,Fig.15; Leijen 2008,Fig.5]. To implement the [EQF-QUANT] rule one usually instantiates both outer quantifiers with a fresh constant (often called a “skolem” constant [Odersky

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

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

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

and Läufer 1996; Peyton Jones et al. 2007]) 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, which may also give new insights in how these systems relate to each other.

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 the FreezeML is fundamentally syntax directed and we need to base a "FreezeHMQ" version on the syntax directed rules of HMQ (see Figure 11 in Appendix B.1), where we instantiate at variable occurrences, and generalize ([GEN] and [GENSUB]) at let-bindings. We can add the freezing rule of FreezeML directly in HMQ:

$$\frac{x\,{:}\,\sigma\ \in \Gamma}{\varnothing \mid \Gamma \vdash_{\mathsf{s}} \lceil x \rceil\ :\ \sigma}\textsc{freeze}$$

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 $\mathsf{let}\ f = \lambda x.x\ \mathsf{in}\ \lceil f \rceil\ 42$. We would expect this to be rejected with the frozen type for f as $\forall\alpha.\alpha{\rightarrow}\alpha$ (wich cannot be directly applied). However, if we allow less-general types for let-bindings we could also derive the type $\mathit{int}{\rightarrow}\mathit{int}$ 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]:

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

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

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 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:

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \sigma_1 \quad \Gamma, x{:}\sigma_1 \vdash e_2 : \sigma_2 \\ \forall \sigma'_1.\ \Gamma \vdash e_1 : \sigma'_1 \Rightarrow \sigma_1 \sqsubseteq \sigma'_1\end{array}}{\Gamma \vdash \mathsf{let}\ x\ =\ e_1\ \mathsf{in}\ e_2}\ \textsc{hmf-let} \qquad \frac{\begin{array}{c}\Gamma \vdash e_1 : \sigma_2 \rightarrow \sigma \quad \Gamma \vdash e_2 : \sigma_2 \\ \forall \sigma'\ \sigma'_2.\ (\Gamma \vdash e_1 : \sigma'_2 \rightarrow \sigma' \ \wedge \Gamma \vdash e_2 : \sigma'_2) \\ \Rightarrow [\![\sigma_2 \rightarrow \sigma]\!] \leqslant [\![\sigma'_2 \rightarrow \sigma']\!]\end{array}}{\Gamma \vdash e_1\ e_2 : \sigma}\ \textsc{hmf-app}$$

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 $\forall\alpha.[\alpha{\rightarrow}\alpha]$ 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 $\alpha \rightarrow \ldots$ is applied to a polymorphic argument σ. In such 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:

$$\boxed{\underset{\substack{\downarrow\\ \text{out}}}{Q} \vdash \underset{\substack{\uparrow\\ \text{in}}}{\rho} \overset{\rightarrow}{\approx} \underset{\substack{\uparrow\\ \text{in}}}{\sigma\rightarrow} \underset{\substack{\downarrow\\ \text{out}}}{\sigma}}$$

$$\frac{Q_1 \mid \Gamma \vdash e_1 : \rho \qquad Q_2 \mid \Gamma \vdash e_2 : \sigma_2 \qquad Q_3 \vdash \rho \overset{\rightarrow}{\approx} \sigma_2 \rightarrow \sigma}{Q_1, Q_2, Q_3 \mid \Gamma \vdash e_1\ e_2 : \sigma}\text{APP-HMF-MATCH}$$

$$\frac{Q \vdash \rho_1 \approx \rho_2}{Q \vdash \rho_1 \rightarrow \sigma \overset{\rightarrow}{\approx} \rho_2 \rightarrow \sigma}\text{MFUN} \qquad \frac{Q \vdash \alpha \approx \rho \rightarrow \beta \qquad \text{fresh } \beta}{Q \vdash \alpha \overset{\rightarrow}{\approx} \rho \rightarrow \beta}\text{MVAR} \qquad \frac{Q \vdash \sigma_1 \approx \sigma_2 \qquad \sigma_1 \notin \rho}{Q \vdash \sigma_1 \rightarrow \sigma \overset{\rightarrow}{\approx} \sigma_2 \rightarrow \sigma}\text{MQUANT}$$

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 $\forall\alpha.[\alpha\rightarrow\alpha]$).

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 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 for lambda-bound parameters [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 $\beta\rightarrow\tau$. Here, we propagate just the information that e_1 must be a function with result type τ where we use β to

$$\boxed{\underset{\substack{\downarrow\\ \text{out}}}{Q} \mid \underset{\substack{\uparrow\\ \text{in}}}{\Gamma} \vdash \underset{\substack{\uparrow\\ \text{in}}}{e} \overset{\leftharpoonup}{:} \underset{\substack{\uparrow\\ \text{in}}}{\sigma}} \quad \text{with } \vDash Q \qquad \frac{Q \mid \Gamma \vdash e \overset{\leftharpoonup}{:} \sigma}{Q \mid \Gamma \vdash (e\,{:}\,\sigma) : \sigma}\,\textsc{ann} \qquad \frac{Q_1 \mid \Gamma \vdash e : \tau_1 \qquad Q_2 \vdash \tau_1 \approx \tau_2}{Q_1, Q_2 \mid \Gamma \vdash e \overset{\leftharpoonup}{:} \tau_2}\,\textsc{chk}$$

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

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

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

Fig. 6. Bidirectional type checking rules

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 $\tau_2{\to}\alpha$ 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 aritmethic operations to work over integers and floats. Another example is the C++ language which uses templates to provide rich static overloading for any user defined functions.

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:

$$\mathit{modi}/\mathit{show} \;:\; \mathit{int} \longrightarrow \mathit{string} \;=\; \ldots$$
$$\mathit{modb}/\mathit{show} \;:\; \mathit{bool} \longrightarrow \mathit{string} \;=\; \ldots$$

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[3].

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 $\lambda x.\ \mathit{show}\ x$ 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.

Trying to specify static overloading directly on top of HM would again require complex side-conditions and rules – and it is not obvious if such solution even exists since static overloading is in direct tension with the HM lambda rule.

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 $\lambda x.\ \mathit{show}\ x$ 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: we *only* need to extend the bidirectional rules of Figure 6 with a case for variables:

[3] This is of course a severe restriction as it prohibits abstraction over overloaded variables. However, Following Lewis et al. [2000], we believe such abstraction should be a separate and orthogonal concept though, where we use implicit parameters in combination with static overloading, as is done in the Koka language for example [Leijen 2021].

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

We use unique notation in the [varc] rule to mean: "for all $m/x : \sigma \in \Gamma$, there exists exactly one declaration that satisfies $Q \vdash \sigma \sqsubseteq \tau$". The bidirectional type rules provide the type information required to disambiguate the variable. For example, we can derive the type of *show* 1 as:

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

In essence, extending HMQ with static overloading as disambiguation over qualified names is as simple as shown here, and the [varc] and [instancec] rules are also straightforward to implement. However, there are still some implementation and design issues to consider which we discuss in the following sections.

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:

$$\begin{array}{l} modi/neg : int \to int \\ modf/neg : float \to float \end{array}$$

with $sqrt : float \to float$. If we now consider the expression $\lambda x.\ sqrt\ (neg\ x)$ we can only accept this if we use [app-argc] on the application $sqrt\ (neg\ x)$ to propagate the *float* result type into the argument expression $neg\ x$. 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 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 always prefer to 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 such 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:

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

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:

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

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 $\mathit{modi/add}$ 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.

6.4 Spooky Action at a Distance

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 $\mathit{int}{\to}\mathit{int}$. According to the type rules, this expression should be rejected since x will have an abstract type in each derivation of $\mathit{inc}\ x$ and $\mathit{show}\ x$ and thus show cannot be disambiguated. However, if we naively use algorithm W, the type of $x : \alpha$ is substituted after checking $\mathit{inc}\ x$ to $x : \mathit{int}$, and subsequently $\mathit{show}\ x$ 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).

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 a HM style system though where the type of λ-bound parameters is "guessed". We believe it should be possible though 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 polymorphic 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]:

$$\frac{\Theta_0 ⨾ \vdash\ e : \tau\ \dashv \Theta_1 ⨾ \Xi}{\Theta_0 \vdash\ e : \forall\Xi.\tau\ \dashv \Theta_1}\textsc{gen-ctx} \qquad \begin{array}{ll} \forall\varnothing.\tau & = \ \tau \\ \forall(\alpha : *, \Xi).\tau & = \ \forall\alpha.(\forall\Xi.\tau) \\ \forall(\alpha{:=}\tau' : *, \Xi).\tau & = \ [\alpha{:=}\tau'](\forall\Xi.\tau) \end{array}$$

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:

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

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 (and instantiation and generalization) 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:

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

Their [app-con] is very similar to our [app] rule except that the constraint $\{\tau_1 \equiv \tau_2 \to \alpha\}$ is directly included while in HMQ one derives a prefix Q_3 from the equivalence relation $Q_3 \vdash\ \tau_1 \approx \tau_2 \to \alpha$. 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]:

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

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

(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:

```
$inferW'(\Gamma, e_1\ e_2)\ =$
  let $(\theta_1, \tau_1)\ =\ inferW'(\Gamma, e_1)$
  let $(\theta_2, \tau_2)\ =\ inferW'(\Gamma, e_2)$
  let $\theta\ =\ U_s(\theta_1, \theta_2)$
  let $\alpha\ =$ fresh
  let $\theta'\ =\ unify(\theta\tau_1, \theta\tau_2 \rightarrow \alpha)$
  return $(\theta' \circ \theta \circ \theta_1,\ \theta'\alpha)$
```

The $\theta\ =\ U_s(\theta_1, \theta_2)$ operation unifies two substitutions such that $\theta \circ \theta_1\ =\ \theta \circ \theta_2$ (and θ is the most general substitution to do so). This is very similar how HMQ uses the notion of a consistent union of prefixes. In particular, if we keep all prefixes as an idempotent mapping, these are just substitutions and we can use McAdam's U_s algorithm to compute the consistent union of two prefixes (as shown in Section 3.1.2).

8 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 *Proceedings of the First ACM SIGPLAN Symposium 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 *ML Workshop*.

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 *Proceedings of the Eighth ACM SIGPLAN International Conference 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 *Proceedings of the 13th ACM Symposium of the International Conference 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`.

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 *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 108–118. POPL'00. ACM, Boston, MA, USA.

McAdam. 1999. On the Unification of Substitutions in Type Inference. In *Implementation of Functional Languages*, edited by Kevin Hammond, Tony Davie, and Chris Clack, 137–152. Springer, Berlin, Heidelberg.

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. arXiv:cs.PL/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 *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, 39–50. TLDI '10. ACM, New York, NY, USA.

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 *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 60–76. POPL'89. ACM, Austin, Texas, USA. doi:https://doi.org/10.1145/75277.75283.

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.

$$\boxed{\begin{array}{llll}\Gamma & \vdash_{\mathsf{HM}}\; e & : & \sigma\\ \uparrow & \;\;\;\;\uparrow & & \downarrow\\ \text{in} & \;\;\;\;\text{in} & & \text{out}\end{array}}$$

$$\frac{x : \sigma \;\in \Gamma}{\Gamma \vdash_{\mathsf{HM}} x \;:\; \sigma}\,\textsc{var}_{\textsc{hm}} \qquad \frac{\Gamma \vdash_{\mathsf{HM}} e_1 \;:\; \sigma \quad \Gamma, x : \sigma \vdash_{\mathsf{HM}} e_2 \;:\; \tau}{\Gamma \vdash_{\mathsf{HM}} \mathsf{let}\; x \;=\; e_1 \;\mathsf{in}\; e_2 \;:\; \tau}\,\textsc{let}_{\textsc{hm}}$$

$$\frac{\Gamma, x : \tau_1 \vdash_{\mathsf{HM}} e \;:\; \tau_2}{\Gamma \vdash_{\mathsf{HM}} \lambda x.\; e \;:\; \tau_1 \to \tau_2}\,\textsc{fun}_{\textsc{hm}} \qquad \frac{\Gamma \vdash_{\mathsf{HM}} e_1 \;:\; \tau_2 \to \tau \quad \Gamma \vdash_{\mathsf{HM}} e_2 \;:\; \tau_2}{\Gamma \vdash_{\mathsf{HM}} e_1\; e_2 \;:\; \tau}\,\textsc{app}_{\textsc{hm}}$$

$$\frac{\Gamma \vdash_{\mathsf{HM}} e \;:\; \forall\alpha.\sigma}{\Gamma \vdash_{\mathsf{HM}} e \;:\; [\alpha{:=}\tau]\sigma}\,\textsc{inst}_{\textsc{hm}} \qquad \frac{\Gamma \vdash_{\mathsf{HM}} e \;:\; \sigma \quad \alpha \notin \mathsf{ftv}(\Gamma, e)}{\Gamma \vdash_{\mathsf{HM}} e \;:\; \forall\alpha.\sigma}\,\textsc{gen}_{\textsc{hm}}$$

Fig. 7. HM type rules.

$$\boxed{\begin{array}{llll}\Gamma & \vdash_{\mathsf{HMS}}\; e & : & \tau\\ \uparrow & \;\;\;\;\uparrow & & \downarrow\\ \text{in} & \;\;\;\;\text{in} & & \text{out}\end{array}} \qquad \mathsf{gen}(\Gamma, \tau) = \forall\overline{\alpha}.\tau \;\;\text{ with } \overline{\alpha} = \mathsf{ftv}(\tau) - \mathsf{ftv}(\Gamma)$$

$$\frac{x : \forall\overline{\alpha}.\tau \;\in \Gamma}{\Gamma \vdash_{\mathsf{HMS}} x \;:\; [\overline{\alpha}{:=}\overline{\tau}]\tau}\,\textsc{var}_{\textsc{hms}} \qquad \frac{\Gamma \vdash_{\mathsf{HMS}} e_1 \;:\; \tau_1 \quad \Gamma, x : \sigma \vdash_{\mathsf{HMS}} e_2 \;:\; \tau_2 \quad \sigma = \mathsf{gen}(\Gamma, \tau_1)}{\Gamma \vdash_{\mathsf{HMS}} \mathsf{let}\; x \;=\; e_1 \;\mathsf{in}\; e_2 \;:\; \tau_2}\,\textsc{let}_{\textsc{hms}}$$

$$\frac{\Gamma, x : \tau_1 \vdash_{\mathsf{HMS}} e \;:\; \tau_2}{\Gamma \vdash_{\mathsf{HMS}} \lambda x.\; e \;:\; \tau_1 \to \tau_2}\,\textsc{fun}_{\textsc{hms}} \qquad \frac{\Gamma \vdash_{\mathsf{HMS}} e_1 \;:\; \tau_2 \to \tau \quad \Gamma \vdash_{\mathsf{HM}} e_2 \;:\; \tau_2}{\Gamma \vdash_{\mathsf{HMS}} e_1\; e_2 \;:\; \tau}\,\textsc{app}_{\textsc{hms}}$$

Fig. 8. Syntax directed HM type rules.

A THE DAMAS-HINDLEY-MILNER TYPE RULES

Figure 7 gives the standard HM type rules [Damas and Milner 1982]. A judgment $\Gamma \vdash_{\mathsf{HM}} e \;:\; \sigma$ states that an expression can be given type σ under a type environment Γ. Γ, and e are inherited (i.e. input) while σ is synthesized (i.e. output). We write $\Gamma, x : \sigma$ to extend a type environment Γ with a fresh binding $x : \sigma$ where $x \notin \mathsf{dom}(\Gamma)$ (which we can always ensure by appropriate renaming).

The [$\textsc{var}_{\textsc{hm}}$] rule derives the type of a variable that is bound in the environment. This will always be a monomorphic type τ for lambda-bound variables but can be polymorphic for let-bound variables as the [$\textsc{let}_{\textsc{hm}}$] rule allows a σ type for the binding. As discussed in the introduction, the [$\textsc{abs}_{\textsc{hm}}$] rule allows "guessing" any τ_1 type for the parameter. The [$\textsc{inst}_{\textsc{hm}}$] rule is another source of "guessing", as we can freely instantiate a polymorphic binder to any τ' that fits the derivation.

A.1 Syntax Directed Type Rules

As a step towards an inference algorithm, we can also give syntax directed rules for the HM rules as shown in Figure 8. Following Damas and Milner [1982], we always instantiate variables and generalize at let-bindings.

A.2 HM Type Inference: Algorithm W

Damas and Milner [1982] describe a type inference algorithm W (shown as *inferW* in Figure 9) which always infers a most-general type, and they show it is sound and complete with respect to the inference rules:

```
$unify\ :\ (\tau_1, \tau_2) \rightarrow \theta$
$unify\ (\alpha, \alpha)\ =$
  $id$

$unify\ (\alpha, \tau)$ or $(\tau, \alpha)\ |\ \alpha \notin \mathrm{ftv}(\tau)\ =$
  $[\alpha{:=}\tau]$

$unify\ (\tau_1{\rightarrow}\tau_2, \tau_1'{\rightarrow}\tau_2')\ =$
  let $\theta_1\ =\ unify(\tau_1, \tau_1')$
  let $\theta_2\ =\ unify(\theta_1\tau_2, \theta_1\tau_2')$
  $(\theta_2 \circ \theta_1)$

$gen\ :\ (\Gamma, \tau) \rightarrow \sigma$
$gen(\Gamma, \tau)\ =$
  let $\overline{\alpha}\ =\ \mathrm{ftv}(\tau)\ -\ \mathrm{ftv}(\Gamma)$
  $\forall\overline{\alpha}.\ \tau$
```

```
$inferW\ :\ (\Gamma, e) \rightarrow (\theta, \tau)$
$inferW(\Gamma, x)\ =$
  let $\forall\overline{\alpha}.\ \tau\ =\ \Gamma(x)$
  let $\overline{\beta}\ =$ fresh
  $(id, [\overline{\alpha}{:=}\overline{\beta}]\tau)$
$inferW(\Gamma, e_1\ e_2)\ =$
  let $(\theta_1, \tau_1)\ =\ inferW(\Gamma, e_1)$
  let $(\theta_2, \tau_2)\ =\ inferW(\theta_1\Gamma, e_2)$
  let $\alpha\ =$ fresh
  let $\theta_3\ =\ unify(\theta_2\tau_1,\ \tau_2 \rightarrow \alpha)$
  $(\theta_3 \circ \theta_2 \circ \theta_1,\ \theta_3\alpha)$
$inferW(\Gamma, \lambda x.e)\ =$
  let $\alpha\ =$ fresh
  let $(\theta, \tau)\ =\ inferW((\Gamma, x : \alpha), e)$
  $(\theta, \theta\alpha \rightarrow \tau)$
$inferW(\Gamma,$ let $x\ =\ e_1$ in $e_2)\ =$
  let $(\theta_1, \tau_1)\ =\ inferW(\Gamma, e_1)$
  let $\sigma\ =\ gen(\theta_1\Gamma, \tau_1)$
  let $(\theta_2, \tau_2)\ =\ inferW((\theta_1\Gamma, x : \sigma), e_2)$
  $(\theta_2 \circ \theta_1,\ \tau_2)$
```

Fig. 9. Algorithm W

Theorem A.8. (*Algorithm W is sound*)
If $(\theta, \tau)\ =\ infer(\Gamma, e)$, then $\theta\Gamma \vdash_{\mathsf{HM}} e\ :\ \tau$.

Theorem A.9. (*Algorithm W is complete*)
If $\Gamma \vdash_{\mathsf{HM}} e\ :\ \sigma$, then $(\theta, \tau)\ =\ \mathrm{infer}(\Gamma, e)$ and $gen(\theta\Gamma, \tau) \sqsubseteq \sigma$.

$$\boxed{\begin{array}{ccccc} F \mid & Q \mid & \Gamma \vdash & e \,:\, & \sigma \\ \uparrow & \downarrow & \uparrow & \uparrow & \downarrow \\ \text{in} & \text{out} & \text{in} & \text{in} & \text{out} \end{array}} \quad \text{with } \models Q, \text{ and } F \not\pitchfork \mathsf{ftv}(\Gamma)$$

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

$$\frac{F \mid Q \mid \Gamma, x : \alpha \vdash e : \tau}{F, \alpha \mid Q \mid \Gamma \vdash \lambda x.\, e : \alpha \to \tau}\textsc{fun} \qquad \frac{F \mid Q \cdot \alpha{=}\tau' \mid \Gamma \vdash e : \tau \quad \alpha \notin \mathsf{ftv}(Q, \Gamma)}{F \mid Q \mid \Gamma \vdash e : [\alpha{:=}\tau']\tau}\textsc{gensub}$$

$$\frac{F_1 \mid Q_1 \mid \Gamma \vdash e_1 : \tau_1 \quad F_2 \mid Q_2 \mid \Gamma \vdash e_2 : \tau_2 \quad Q_3 \vdash \tau_1 \approx \tau_2 \to \alpha}{F_1, F_2, \alpha \mid Q_1, Q_2, Q_3 \mid \Gamma \vdash e_1\ e_2 : \alpha}\textsc{app}$$

$$\frac{F_1 \mid Q_1 \mid \Gamma \vdash e_1 : \sigma \quad F_2 \mid Q_2 \mid \Gamma, x : \sigma \vdash e_2 : \tau \quad \mathsf{ftv}(\sigma) \subseteq \mathsf{ftv}(\Gamma)}{F_1, F_2 \mid Q_1, Q_2 \mid \Gamma \vdash \mathsf{let}\ x\ =\ e_1\ \mathsf{in}\ e_2 : \tau}\textsc{let}$$

Fig. 10. Type rules under a prefix using a fresh name supply F (where we write F_1, F_2 for the disjoint union $F_1 \uplus F_2$)

B HMQ WITH EXPLICIT FRESH NAMES

Figure 10 gives full inductive type rules for HMQ using an explicit fresh name supply F. The fresh α notation used in Figure 2 is essentially a convenient shorthand for the rules here with an explicit name supply. We write F_1, F_2 for the disjoint union of F_1 and F_2, where $F_1, F_2 \ \doteq\ F_1 \uplus F_2$. Every time we used fresh α in the rules in Figure 2, we now pick a fresh α from the name supply F (as F, 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. A well-formedness condition for the new rules is that F is disjoint from the free type variables in the environment, with $F \not\pitchfork \mathsf{ftv}(\Gamma)$. This ensures that for every derivation the fresh names are indeed fresh and do not contain type variable names occurring in the environment. Note that for the output prefix Q and type σ, we have $\mathsf{ftv}(Q, \sigma) \subseteq \mathsf{ftv}(F, \Gamma)$ – that is, all output type variables are either fresh or occur free in the environment.

The need for fresh names in HMQ is not ideal, and one might have hoped to see a local condition instead, for example:

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

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 $\mathit{const} : \forall\alpha\beta.\ \alpha \to \beta \to \alpha$. 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]), leading to:

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

and thus *const id* 1 gets type $\mathit{int}{\to}\mathit{int}$ instead of the expected $\forall\alpha.\alpha{\to}\alpha$. The formalization in Figure 10 ensures that separate sub-derivations all use unique names by using a disjoint union of

$$\boxed{\underset{\substack{\uparrow\\ \text{in}}}{F} \mid \underset{\substack{\downarrow\\ \text{out}}}{Q} \mid \underset{\substack{\uparrow\\ \text{in}}}{\Gamma} \vdash_{\mathrm{S}} \underset{\substack{\uparrow\\ \text{in}}}{e} \,:\, \underset{\substack{\downarrow\\ \text{out}}}{\tau}} \qquad \text{with } \vDash Q, \text{ and } F \not\pitchfork \mathrm{ftv}(\Gamma)$$

$$\frac{x : \forall \overline{\alpha}.\ \tau \ \in \Gamma}{\overline{\alpha} \mid \varnothing \mid \Gamma \vdash_{\mathrm{S}} x \,:\, \tau}\,\textsc{var}_{\mathrm{S}} \qquad\qquad \frac{F \mid Q \mid \Gamma, x : \alpha \vdash_{\mathrm{S}} e \,:\, \tau}{F, \alpha \mid Q \mid \Gamma \vdash_{\mathrm{S}} \lambda x.\ e \,:\, \alpha \to \tau}\,\textsc{fun}_{\mathrm{S}}$$

$$\frac{F_1 \mid Q_1 \mid \Gamma \vdash_{\mathrm{S}} e_1 \,:\, \tau_1 \qquad F_2 \mid Q_2 \mid \Gamma \vdash_{\mathrm{S}} e_2 \,:\, \tau_2 \qquad Q_3 \vdash \tau_1 \approx \tau_2 \to \alpha}{F_1, F_2, \alpha \mid Q_1, Q_2, Q_3 \mid \Gamma \vdash_{\mathrm{S}} e_1\ e_2 \,:\, \alpha}\,\textsc{app}_{\mathrm{S}}$$

$$\frac{F_1 \mid Q_0 \mid \Gamma \vdash_{\mathrm{S}} e_1 \,:\, \tau_1 \qquad F_2 \mid Q_2 \mid \Gamma, x : \sigma \vdash_{\mathrm{S}} e_2 \,:\, \tau_2 \qquad (Q_1, \sigma) = \mathrm{gen}(Q_0, \Gamma, \tau_1)}{F_1, F_2 \mid Q_1, Q_2 \mid \Gamma \vdash_{\mathrm{S}} \mathsf{let}\ x \ =\ e_1\ \mathsf{in}\ e_2 \,:\, \tau_2}\,\textsc{let}_{\mathrm{S}}$$

$$\begin{array}{lcll}
\multicolumn{4}{l}{\mathrm{gen} \,:\, (Q, \Gamma, \tau) \to (Q, \sigma)} \\
\mathrm{gen}(Q \cdot \alpha{=}\tau',\ \Gamma,\ \tau) & = & \mathrm{gen}(Q,\ \Gamma,\ [\alpha{:=}\tau']\tau) & \text{if } \alpha \notin \mathrm{ftv}(Q, \Gamma) \\
\mathrm{gen}(Q,\ \Gamma,\ \tau) & = & (Q, \forall \overline{\alpha}.\ \tau) & \text{if } \mathrm{dom}(Q) \subseteq \mathrm{ftv}(\Gamma), \text{ with } \overline{\alpha} = \mathrm{ftv}(\tau) - \mathrm{ftv}(\Gamma)
\end{array}$$

Fig. 11. Syntax directed type rules under a prefix

the names used in each sub-derivation (and thus, we cannot both instantiate *const* and *id* with a shared β as in our example).

Lemma B.10. (*Free type variables are either fresh or occur free in the environment*)
If $F \mid Q \mid \Gamma \vdash e \,:\, \sigma$, then $\mathrm{ftv}(Q, \sigma) \subseteq \mathrm{ftv}(F, \Gamma)$.

Proof. (*Of Lemma B.10*) By induction over the type rules. □

B.1 Syntax Directed Type Rules for HMQ

Figure 11 gives the syntax directed type rules for HMQ. We can make the declarative type rules (Figure 10) syntax-directed in the usual way by applying full instantiation at the leaves of the derivation at a variable occurrence, and applying full generalization at let bindings. The [$\textsc{var}_{\mathrm{S}}$] rule now instantiates the type of variable fully with fresh variables $\overline{\alpha}$ for the quantifiers – just like in the [$\textsc{inst}$] rule this may require α-renaming of the quantifiers.

Generalization now takes place in the [$\textsc{let}_{\mathrm{S}}$] rule using the gen function that takes the prefix Q, the environment Γ, and a monotype τ, and returns a new prefix Q' and generalized type σ. The new prefix Q' only has bindings that still occur in Γ with $\mathrm{dom}(Q') \subseteq \mathrm{ftv}(\Gamma)$. The first case of generalization essentially applies [$\textsc{gensub}$] for all binders α in Q that do not occur free in Γ. The second case corresponds the [$\textsc{gen}$] rule and to generalization in HM type rules (see Figure 8) where we quantify over all free variables in τ that do not occur free in Γ.

Theorem B.11. (*The syntax directed HMQ rules are sound*)
If $F \mid Q \mid \Gamma \vdash_{\mathrm{S}} e \,:\, \tau$, then also $F \mid Q \mid \Gamma \vdash e \,:\, \tau$.

Theorem B.12. (*The syntax directed HMQ rules are complete*)
If $F \mid Q \mid \Gamma \vdash e \,:\, \sigma$, then also $F \mid Q' \mid \Gamma \vdash_{\mathrm{S}} e \,:\, \tau$ with $(Q, \sigma) = \mathrm{gen}(Q', \Gamma, \tau)$.

B.2 Algorithm WQ

```
$\mathit{unify} : (\tau_1, \tau_2) \to \theta$
$\mathit{unify}\ (\alpha, \alpha) =$
  $\mathit{id}$

$\mathit{unify}\ (\alpha, \tau)$ or $(\tau, \alpha) \mid \alpha \notin \mathsf{ftv}(\tau) =$
  $[\alpha{:=}\tau]$

$\mathit{unify}\ (\tau_1{\to}\tau_2, \tau'_1{\to}\tau'_2) =$
  let $\theta_1 = \mathit{unify}(\tau_1, \tau'_1)$
  let $\theta_2 = \mathit{unify}(\theta_1\tau_2, \theta_1\tau'_2)$
  $(\theta_2 \circ \theta_1)$

$\mathit{unifies} : [\tau] \to \theta$
$\mathit{unifies}\ []$ or $[\tau] = \mathit{id}$
$\mathit{unifies}\ (\tau_1 : \tau_2 : \overline{\tau}) =$
  let $\theta_1 = \mathit{unify}(\tau_1, \tau_2)$
  let $\theta_2 = \mathit{unifies}\ (\theta_1\tau_2\ :\ \theta_1\overline{\tau})$
  $(\theta_2 \circ \theta_1)$

$\mathit{gen} : (\Gamma, \tau) \to \sigma$
$\mathit{gen}(\Gamma, \tau) =$
  let $\overline{\alpha} = \mathsf{ftv}(\tau) - \mathsf{ftv}(\Gamma)$
  $(\forall\overline{\alpha}.\ \tau)$
```

```
$\mathit{inferWQ} : (\Gamma, e) \to (\theta, \tau)$
$\mathit{inferWQ}(\Gamma, x_i) =$
  let $\alpha = \Gamma(x)$
  $(\mathit{id}, \alpha_i)$
$\mathit{inferWQ}(\Gamma, x) =$
  let $\forall\overline{\alpha}.\ \tau = \Gamma(x)$
  let $\overline{\beta}$ = fresh
  $(\mathit{id}, [\overline{\alpha}{:=}\overline{\beta}]\tau)$
$\mathit{inferWQ}(\Gamma, e_1\ e_2) =$
  let $(\theta_1, \tau_1) = \mathit{inferWQ}(\Gamma, e_1)$
  let $(\theta_2, \tau_2) = \mathit{inferWQ}(\theta_1\Gamma, e_2)$
  let $\alpha$ = fresh
  let $\theta_3 = \mathit{unify}(\theta_2\tau_1,\ \tau_2 \to \alpha)$
  $(\theta_3 \circ \theta_2 \circ \theta_1,\ \theta_3\alpha)$
$\mathit{inferWQ}(\Gamma, \lambda x_n.e) =$
  let $\alpha$ = fresh
  let $(\theta_1, \tau) = \mathit{inferWQ}((\Gamma, x : \alpha), e)$
  let $\theta_2 = \mathit{unifies}(\alpha, \theta_1\alpha_1, \ldots, \theta_1\alpha_n)$
  $(\theta_2, \theta_2\alpha \to \theta_2\tau)$
$\mathit{inferWQ}(\Gamma,$ let $x = e_1$ in $e_2) =$
  let $(\theta_1, \tau_1) = \mathit{inferWQ}(\Gamma, e_1)$
  let $\sigma = \mathit{gen}(\theta_1\Gamma, \tau_1)$
  let $(\theta_2, \tau_2) = \mathit{inferWQ}((\theta_1\Gamma, x : \sigma), e_2)$
  $(\theta_2 \circ \theta_1,\ \tau_2)$
```

Fig. 12. Algorithm WQ

Figure 12 gives a type inference algorithm WQ for HMQ which is closely based on algorithm W. The algorithm assumes a pre-processing step where every lambda binding x is annotated (as x_n) with the total number of occurrences n in the body of the lambda, and where every lambda bound variable occurrence is annotated with its (unique) occurrence i (as x_i). This way, we can generate an initial fresh variable α for a lambda bound variable, and use that to generate a unique fresh type variable α_i per occurrence. Eventually, we unify all α_i occurrences again. This removes any accidental propagation of type inference unifications between different derivations which happens in plain algorithm W.

This use of a unique type variable per occurrence may delay unification errors until the moment all occurrences are unified. However, at the same time it may improve the precision of the type error since we can for example pick the most probable error instead of the the one that occurs first [Heeren et al. 2003], i.e. for $\lambda x.\ (\mathit{inc}\ x,\ \mathit{sqr}\ x,\ \mathit{not}\ x)$ we can give an error for the single *bool* occurrence since the two *int* occurrences are more common.

Theorem B.13. (*Algorithm WQ is sound*)
If $(Q, \tau) = \mathit{inferWQ}(\Gamma, e)$ with $\mathsf{ftv}(\Gamma) = \varnothing$, then $F \mid Q \mid \Gamma \vdash_{\mathsf{s}} e : \tau$ (for some F).

Theorem B.14. (*Algorithm WQ is complete*)
If $F \mid Q \mid \Gamma \vdash_{\mathsf{s}} e : \tau$, then also $(Q, \tau) = \mathit{inferWQ}(\Gamma, e)$.

The soundness theorem requires initially an environment without free type variables, such that it contains only let bindings. This is needed since unification errors may be delayed in algorithm WQ until the lambda-binding, whereas the type rules always require $\vDash Q$ at every step. For the inductive proof we need to use a more general theorem to account for this.

C PROOFS

C.1 Substitutions

A *substitution* θ is an idempotent function from type variables to types. The (finite) domain of of θ is the set of type variables such that $\theta(\alpha) \neq \alpha$ for any $\alpha \in \mathsf{dom}(\theta)$, while the codomain consists of the free type variables of its range.

We use the notation $[\alpha{:=}\tau]$ for a singleton substitution θ with domain $\{\alpha\}$ and $\theta(\alpha) = \tau$. We usually write substitution application with explicit parenthesis as $\theta(\tau)$ but sometimes shorten to just $\theta\tau$ when appropriate[5].

We write $\theta \sqsubseteq \theta'$ if θ is a *more general* (or less specific) substitution than θ', such that $\theta' \;=\; \theta'' \circ \theta$ for some θ''. We say that two substitutions are equivalent if each is as-general as the other, i.e. $\theta_1 \sqsubseteq \theta_2 \wedge \theta_2 \sqsubseteq \theta_1 \Leftrightarrow \theta_1 \equiv \theta_2$ (such substitutions are not always exactly equal since they can potentially differ on a renaming between type variables $\alpha_1{:=}\alpha_2$ where either direction is possible).

Properties C.15. (*Substitution*)
1. For any τ and well-formed θ, $\mathsf{dom}(\theta) \notin \mathsf{ftv}(\theta(\tau))$.
2. For any τ with $\mathsf{dom}(\theta) \notin \mathsf{ftv}(\tau)$, $\theta(\tau) \;=\; \tau$.
3. For a well-formed θ with $\alpha \notin \mathsf{ftv}(\theta, \tau')$, $\theta([\alpha{:=}\tau'](\tau)) \;=\; [\alpha{:=}\theta(\tau')](\theta(\tau))$.

Proof. (*of Property C.15.3*) We have $\alpha \notin \mathsf{ftv}(\theta, \tau')$ (**1**), and thus $\theta(\alpha) = \alpha$ (**2**). Induction on τ.

Case $[\tau = \alpha]$:

$$\begin{array}{lll} & \theta([\alpha{:=}\tau'](\alpha)) & \\ = & \theta(\tau') & \{\ \mathit{def.}\ \} \\ = & [\alpha{:=}\theta(\tau')](\alpha) & \{\ \mathit{def.}\ \} \\ = & [\alpha{:=}\theta(\tau')](\theta(\alpha)) & \{\ (2)\ \} \end{array}$$

Case $[\tau = \tau_1]$: with $\alpha \notin \mathsf{ftv}(\tau_1)$ (**3**):

$$\begin{array}{lll} & \theta([\alpha{:=}\tau'](\tau_1)) & \\ = & \theta(\tau_1) & \{\ (3).\ \} \\ = & [\alpha{:=}\theta(\tau')](\theta(\tau_1)) & \{\ (3),\ (1),\ \mathit{prop}\ \mathrm{C}.15.2\ \} \end{array}$$

Case $[\tau = \tau_1{\rightarrow}\tau_2]$:

$$\begin{array}{lll} & \theta([\alpha{:=}\tau'](\tau_1{\rightarrow}\tau_2)) & \\ = & \theta([\alpha{:=}\tau']\tau_1 \rightarrow \theta([\alpha{:=}\tau']\tau_2) & \{\ \mathit{def.}\ \} \\ = & [\alpha{:=}\theta(\tau')](\theta(\tau_1)) \rightarrow [\alpha{:=}\theta(\tau')](\theta(\tau_2)) & \{\ \mathit{ind.\ hyp.}\ \} \\ = & [\alpha{:=}\theta(\tau')](\theta(\tau_1{\rightarrow}\tau_2)) & \{\ \mathit{def.}\ \} \end{array}$$

□

Lemma C.16. (*Substitution cancelation*)
If $\alpha \notin \mathsf{ftv}(\tau)$ and θ is well-formed with $\alpha \in \mathsf{dom}(\theta)$, then $\theta \;=\; [\alpha{:=}\tau'] \circ \theta$ (for any τ').

Proof. (*of Lemma C.16*) With $\alpha \in \mathsf{dom}(\theta)$, we have for any τ, $\alpha \notin \mathsf{ftv}(\theta(\tau))$ (**1**) (due to prop C.15.1). Therefore,

$$\begin{array}{lll} & ([\alpha{:=}\tau'] \circ \theta)(\tau) & \\ = & [\alpha{:=}\tau'](\theta(\tau)) & \{\ \mathit{def}\ \} \\ = & \theta(\tau) & \{\ (1),\ \mathit{prop}\ \mathrm{C}.15.2\ \} \end{array}$$

□

[5] Note that we do not use the common notation $\tau[\alpha{:=}\tau']$ but always write this as $[\alpha{:=}\tau'](\tau)$ (or $[\alpha{:=}\tau']\tau$).

Lemma C.17. (*Substitution commutation*)
If $\alpha \notin \mathrm{ftv}(\theta)$ (**1**), then $\theta \circ [\alpha{:=}\tau] \;=\; [\alpha{:=}\theta(\tau)] \circ \theta$

Proof. (*of Lemma C.17*) For any τ', we have:

$$\begin{array}{lll}
 & (\theta \circ [\alpha{:=}\tau])(\tau') & \\
= & \theta([\alpha{:=}\tau](\tau')) & \{\ \mathit{def.}\ \} \\
= & [\alpha{:=}\theta(\tau)](\theta(\tau')) & \{\ \mathit{prop}\ \mathrm{C}.15.3\ \} \\
= & ([\alpha{:=}\theta(\tau)] \circ \theta)(\tau') & \{\ \mathit{def.}\ \}
\end{array}$$

□

Lemma C.18. (*Equivalence of Composed Substitutions*)
If $\theta_1 \circ [\alpha{:=}\tau_1] \;=\; \theta_2 \circ [\alpha{:=}\tau_2]$ with $\alpha \notin \mathrm{ftv}(\theta_1, \tau_1, \theta_2, \tau_2)$, then $\theta_1 = \theta_2$ with $\theta_1(\tau_1) = \theta_1(\tau_2)$.

Proof. (*of Lemma C.18*) We have $\theta \;=\; \theta_1 \circ [\alpha{:=}\tau_1] \;=\; \theta_2 \circ [\alpha{:=}\tau_2]$ (**1**), with $\alpha \notin \mathrm{ftv}(\theta_1, \tau_1, \theta_2, \tau_2)$ (**2**). For any τ with $\alpha \notin \mathrm{ftv}(\tau)$, we have $\theta_1(\tau) \;=\; \theta_2(\tau)$ (**3**):

$$\begin{array}{lll}
 & \theta_1(\tau) & \\
= & (\theta_1 \circ [\alpha{:=}\tau_1])(\tau) & \{\ \alpha \notin \mathrm{ftv}(\tau),\ (2)\ \} \\
= & \theta(\tau) & \{\ (1)\ \} \\
= & (\theta_2 \circ [\alpha{:=}\tau_2])(\tau) & \{\ (1)\ \} \\
= & \theta_2(\tau) & \{\ \alpha \notin \mathrm{ftv}(\tau),\ (2)\ \}
\end{array}$$

and thus $\theta_1(\tau_1) = \theta_1(\tau_2)$ (**4**):

$$\begin{array}{lll}
 & \theta_1(\tau_1) & \\
= & (\theta_1 \circ [\alpha{:=}\tau_1])(\alpha) & \{\ (2)\ \} \\
= & (\theta_2 \circ [\alpha{:=}\tau_2])(\alpha) & \{\ (1)\ \} \\
= & \theta_2(\tau_2) & \{\ (2)\ \} \\
= & \theta_1(\tau_2) & \{\ (2),\ (3)\ \}
\end{array}$$

Therefore:

$$\begin{array}{lll}
 & \theta_1 \circ [\alpha{:=}\tau_2] & \\
= & [\alpha{:=}\theta_1(\tau_2)] \circ \theta_1 & \{\ \mathit{Lemma}\ \mathrm{C}.17,\ (2)\ \} \\
= & [\alpha{:=}\theta_1(\tau_1)] \circ \theta_1 & \{\ (4)\ \} \\
= & \theta_1 \circ [\alpha{:=}\tau_1] & \{\ \mathit{Lemma}\ \mathrm{C}.17,\ (2)\ \} \\
= & \theta_2 \circ [\alpha{:=}\tau_2] & \{\ (1)\ \}
\end{array}$$

and thus $\theta_1 \;=\; \theta_2$. □

C.2 More General Substitutions

Properties C.19.
1. $\mathit{id} \sqsubseteq \theta$ (for any θ)
2. If $\theta_1 \sqsubseteq \theta_2$ then $\theta_1 \circ \theta \sqsubseteq \theta_2 \circ \theta$.
3. If $\theta(\alpha) = \theta(\tau)$ with $\alpha \notin \mathrm{ftv}(\tau)$, then $[\alpha{:=}\tau] \sqsubseteq \theta$.
4. If $\theta_1 \circ \theta_2 \sqsubseteq \theta$ then $\theta \;=\; \theta' \circ \theta_2$ with $\theta_1 \sqsubseteq \theta'$.
5. If $\theta \sqsubseteq \theta_1 \circ \theta_2$ and $\mathrm{dom}(\theta_1) \not\pitchfork \mathrm{dom}(\theta)$, then $\theta \sqsubseteq \theta_2$.

Proof. (*of Property C.19.1*) For any θ,

$$\begin{array}{lll}
 & \theta & \\
= & \theta \circ \mathit{id} & \{\ \mathit{def.}\ \}
\end{array}$$

□

Proof. (*of Property C.19.2*) With $\theta_1 \sqsubseteq \theta_2$, we have $\theta_2 \ = \ \theta' \circ \theta_1$ (**1**), and thus:

$$\begin{array}{ll} & \theta_2 \circ \theta \\ = & (\theta' \circ \theta_1) \circ \theta \ \ \{ \ (1) \ \} \\ = & \theta' \circ (\theta_1 \circ \theta) \ \ \{ \ \mathit{assoc.} \ \} \end{array}$$

□

Proof. (*of Property C.19.3*) We have $\theta(\alpha) = \theta(\tau)$ (**1**).

$$\begin{array}{ll} & \theta(\alpha) = \theta(\tau) \\ \Rightarrow & \theta \ = \ \theta' \circ [\alpha{:=}\tau] \ \ \{ \ ? \ (\mathit{add\ extra\ substitution\ property}?) \ \} \end{array}$$

□

Proof. (*of Property C.19.4*) We have $\theta_1 \circ \theta_2 \sqsubseteq \theta$ (**1**).

$$\begin{array}{lll} & \theta & \\ = & \theta'' \circ \theta_1 \circ \theta_2 & \{ \ (1), \ \mathit{some}\ \theta'' \ \} \\ = & \theta' \circ \theta_2 & \{ \ \mathit{assume}\ \theta' \ = \ \theta'' \circ \theta_1 \ (\mathbf{2}) \ \} \end{array}$$

and

$$\begin{array}{ll} & \theta' \\ = & \theta'' \circ \theta_1 \ \ \{ \ (2) \ \} \\ \Rightarrow & \theta_1 \sqsubseteq \theta' \end{array}$$

□

Proof. (*of Property C.19.5*) We have $\theta \sqsubseteq \theta_1 \circ \theta_2$ (**1**) and $\mathrm{dom}(\theta_1) \not\cap \mathrm{dom}(\theta)$ (**2**)

$$\begin{array}{lll} & \theta_1 \circ \theta_2 & \\ = & \theta' \circ \theta & \{ \ (1) \ \} \\ = & \theta_3 \circ \theta_4 \circ \theta & \{ \ \mathit{for\ some}\ \theta_3, \theta_4\ \mathit{with}\ \mathrm{dom}(\theta_4) \not\cap \mathrm{dom}(\theta_1) \ \} \end{array}$$

since $\mathrm{dom}(\theta_4 \circ \theta) \not\cap \mathrm{dom}(\theta_1)$, it must be that $\theta_2 \ = \ \theta_4 \circ \theta$, and thus $\theta \sqsubseteq \theta_2$. (TODO: more formal proof?) □

C.3 Prefixes

The solution to a prefix is also a solution to any subset:

Lemma C.20. (*Consistent weakening*)
If $\theta \vDash Q_1 \cup Q_2$, then also $\theta \vDash Q_1$.

Proof. (*of Lemma C.20*) We have $\theta \vDash Q_1 \cup Q_2$ (**1**) and need to show $\theta \vDash Q_1$. From (1) and [SOLUTION], we have $\forall(\alpha{=}\tau) \in Q_1 \cup Q_2.\ \theta(\alpha) = \theta(\tau)$, and thus $\forall(\alpha{=}\tau) \in Q_1.\ \theta(\alpha) = \theta(\tau)$ with $\theta \vDash Q_1$. □

The least solution of a subset of a prefix is less specific than the prefix solution:

Lemma C.21. (*Consistent union*)
If $\vDash Q_1 \cup Q_2$, then $\langle Q_1 \rangle \sqsubseteq \langle Q_1 \cup Q_2 \rangle$.

Proof. (*of Lemma C.21*) By definition $\langle Q_1 \cup Q_2 \rangle \vDash Q_1 \cup Q_2$, and by lemma C.20 $\langle Q_1 \cup Q_2 \rangle$ is a solution of Q_1, $\langle Q_1 \cup Q_2 \rangle \vDash Q_1$. By definition of prefix solution we now have $\langle Q_1 \rangle \sqsubseteq \langle Q_1 \cup Q_2 \rangle$.

An important property is that a prefix solution of a subset is a right identity:

Lemma C.22. (*Prefix extension*)
If $\vDash Q_1 \cup Q_2$, then $\langle Q_1 \cup Q_2 \rangle \ = \ \langle Q_1 \cup Q_2 \rangle \circ \langle Q_1 \rangle$.

Proof. (*of Lemma C.22*) We have $\vDash Q_1 \cup Q_2$ (**1**) and need to show $\langle Q_1 \cup Q_2 \rangle \;=\; \langle Q_1 \cup Q_2 \rangle \circ \langle Q_1 \rangle$. By (1) and Lemma C.21, $\langle Q_1 \rangle \sqsubseteq \langle Q_1 \cup Q_2 \rangle$, which implies $\langle Q_1 \cup Q_2 \rangle \;=\; \theta \circ \langle Q_1 \rangle$ for some θ. Therefore

$$\begin{array}{lll} & \langle Q_1 \cup Q_2 \rangle & \\ = & \theta \circ \langle Q_1 \rangle & \{\ (2)\ \} \\ = & \theta \circ \langle Q_1 \rangle \circ \langle Q_1 \rangle & \{\ \textit{subst. idem.}\ \} \\ = & \langle Q_1 \cup Q_2 \rangle \circ \langle Q_1 \rangle & \{\ (2)\ \} \end{array}$$

□

A nice property of an extracted bound is that we get a stronger form of Lemma C.22 where we can write the prefix solution as a composition of each sub-solution:

Lemma 3.7. (*Extraction corresponds to composition of prefix solutions*)
If $\vDash Q$ and $Q = Q' \cdot \alpha{=}\tau$, then $\langle Q \rangle = \langle Q' \rangle \circ [\alpha{:=}\tau]$.

Proof. (*of Lemma 3.7*) We have $\vDash Q$ (**1**), and $Q \;=\; Q' \cdot \alpha{=}\tau$, and thus $Q \;=\; Q' \uplus \{\alpha{=}\tau\}$ (**2**) with $\alpha \notin \mathrm{ftv}(Q', \tau)$ (**3**) (and need to show $\langle Q \rangle \;=\; \langle Q' \rangle \circ [\alpha{:=}\tau]$). From (1), we have $\forall \beta{=}\tau' \in Q.\ Q[\beta] = Q[\tau']$, and thus from (2,3), $\forall(\beta{=}\tau') \in Q'.\ Q[\beta] = Q[\tau']$ and $Q[\alpha] = Q[\tau]$ (**4**). From (3,4) and Lemma C.19.3, $[\alpha{:=}\tau] \sqsubseteq \langle Q \rangle$ and thus $\langle Q \rangle \;=\; \theta \circ [\alpha{:=}\tau]$ for some θ (**5**), and from (3,5) $\forall(\beta{=}\tau') \in Q'.\ \theta(\beta) = \theta(\tau')$. A minimal solution for θ is $\langle Q' \rangle$, and therefore $\langle Q \rangle \;=\; \langle Q' \rangle \circ [\alpha{:=}\tau]$. □

Lemma C.24. (*Prefixes with a common solution are consistent*)
For any consistent Q_1 and Q_2, if $Q_1 \sqsubseteq \theta$ and $Q_2 \sqsubseteq \theta$, then Q_1, Q_2 is consistent with $(Q_1, Q_2) \sqsubseteq \theta$.

Proof. (*Of Lemma C.24*) We have $\vDash Q_1$ (**1a**) with $Q_1 \sqsubseteq \theta$ (**1b**), and $\vDash Q_2$ with $Q_2 \sqsubseteq \theta$. By definition, from (1a), $\forall \alpha{=}\tau \in Q_1.\ \langle Q_1 \rangle\alpha = \langle Q_1 \rangle\tau$, and thus by (1b) we also have $\forall \alpha{=}\tau \in Q_1.\ \theta\alpha = \theta\tau$ with $\theta \vDash Q_1$ (**2**) and similarly, $\theta \vDash Q_2$ (**3**). It follows by definition that we also have $\theta \vDash (Q_1 \cup Q_2)$ and thus Q_1, Q_2 is consistent. Since $\langle Q_1, Q_2 \rangle$ is minimal by definition, we also have $(Q_1, Q_2) \sqsubseteq \theta$. □

C.4 Consistent Prefixes are Substitutions

Proof. (*Of Theorem 2.6*) For any consistent Q we have a minimal solution $\langle Q \rangle$ $(= \theta)$ which is a well-formed idempotent substitution of the form $[\alpha_1{:=}\tau_1, \ldots, \alpha_n{:=}\tau_n]$ (with α_i pairwise distinct and $\mathrm{dom}(\theta) \not\pitchfork \mathrm{codom}(\theta)$ (**1**)). Define Q' as $\{\alpha_1{=}\tau_1, \ldots, \alpha_n{=}\tau_n\}$. The minimal solution for Q' is also θ and thus $Q \;\equiv\; Q'$ where Q' is an idempotent mapping (by (1)). □

Theorem C.25. (*For a consistent prefix, any binder can be extracted*)
If $\vDash Q$ and $(\alpha{=}\tau) \in Q$, then $Q \;\equiv\; Q' \cdot \alpha{=}\tau'$ (with $Q[\tau] = \tau'$).

Proof. (*Of Lemma C.25*) Since $\vDash Q$, by Theorem 2.6, we have a idempotent mapping Q_0 (**1**) with $Q_0 \;\equiv\; Q$ (**2**). Since $\alpha{=}\tau \in Q$, we must have $Q_0 \;=\; Q' \cup \{\alpha{=}\tau'\}$ for some τ'. Since Q_0 is an idempotent mapping (1), we have $\alpha \notin \mathrm{ftv}(Q', \tau')$, and we can use [EXTRACT] to conclude $Q_0 \;=\; Q' \cdot \alpha{=}\tau'$ and by (2), $Q_0 \;\equiv\; Q' \cdot \alpha{=}\tau'$. Moreover, by (2), $Q[\tau] = Q_0[\tau']$, and since Q_0 is idempotent (1), $Q_0[\tau'] = \tau'$, and therefore $Q[\tau] = \tau'$. □

C.5 Type Equivalence

Type equivalence is sound.

Proof. (*of Theorem C.34*) We have $Q \vdash\ \tau \approx \tau'$ (**1**) (and need to show $Q[\tau] = Q[\tau']$). We proceed by induction over the rules of ($\approx$).

Case [eq-id]: $\tau = \tau'$, and we have $\varnothing \vdash \tau \approx \tau$. It follows directly that $Q[\tau] = \varnothing[\tau] = \tau = \tau' = Q[\tau']$.

Case [eq-var] $\tau = \alpha$, and we have $\{\alpha{=}\tau'\} \vdash \alpha \approx \tau'$ with $\alpha \notin \mathrm{ftv}(\tau')$ (**2**).

$$\begin{array}{lll}
 & Q[\alpha] & \\
= & \langle\{\alpha{=}\tau'\}\rangle(\alpha) & \{\ \textit{assumption}\ \} \\
= & [\alpha{:=}\tau'](\alpha) & \{\ \textit{def.}\ \} \\
= & \tau' & \{\ \textit{def.}\ \} \\
= & [\alpha{:=}\tau'](\tau') & \{\ (2)\ \} \\
= & Q[\tau'] & \{\ \textit{def.}\ \}
\end{array}$$

Case [mvarr]: similar to the previous case.

Case [eq-fun]: We have $Q_1 \vdash \tau_1 \approx \tau_1'$, $Q_2 \vdash \tau_2 \approx \tau_2'$, and $\vDash Q_1 \cup Q_2$. By induction $Q_1[\tau_1] = Q_1[\tau_1']$ and $Q_2[\tau_2] = Q_2[\tau_2']$ (**2**). We can now derive:

$$\begin{array}{lll}
 & (Q_1 \cup Q_2)[\tau_1 {\to} \tau_2] & \\
= & (Q_1 \cup Q_2)[\tau_1] \to (Q_1 \cup Q_2)[\tau_2] & \{\ \textit{def.}\ \} \\
= & (Q_1 \cup Q_2)[Q_1[\tau_1]] \to (Q_1 \cup Q_2)[Q_2[\tau_2]] & \{\ \textit{lemma C.22}\ \} \\
= & (Q_1 \cup Q_2)[Q_1[\tau_1']] \to (Q_1 \cup Q_2)[Q_2[\tau_2']] & \{\ (2)\ \} \\
= & (Q_1 \cup Q_2)[\tau_1'] \to (Q_1 \cup Q_2)[\tau_2'] & \{\ \textit{lemma C.22}\ \} \\
= & (Q_1 \cup Q_2)[\tau_1' {\to} \tau_2'] & \{\ \textit{def.}\ \}
\end{array}$$

□

Lemma C.26. (*Consistent solutions*)
If $Q \sqsubseteq \theta$, then $\theta \vDash Q$.

Proof. (*of Lemma C.26*) Since $\langle Q\rangle \sqsubseteq \theta, \theta = \theta' \circ \langle Q\rangle$ for some θ'. Since $\langle Q\rangle \vDash Q$, we have $\forall(\alpha{=}\tau) \in Q.\ \langle Q\rangle(\alpha) = \langle$
Therefore, $\forall(\alpha{=}\tau) \in Q.\ (\theta' \circ \langle Q\rangle)(\alpha) = (\theta' \circ \langle Q\rangle)(\tau)$, and $\theta \vDash Q$. □

Lemma C.27. (*Consistent strengthen*)
If $Q_1 \sqsubseteq \theta$ and $Q_2 \sqsubseteq \theta$, then $\vDash Q_1 \cup Q_2$ and $Q_1 \cup Q_2 \sqsubseteq \theta$

Proof. (*of Lemma C.27*) We have $Q_1 \sqsubseteq \theta$ (**1**) and $Q_2 \sqsubseteq \theta$ (**2**). From (1) and Lemma C.26, we have $\theta \vDash Q_1$, and thus $\forall(\alpha{=}\tau) \in Q_1.\ \theta(\alpha) = \theta(\tau)$ (**3**). Similarly, from (2) we have $\forall(\alpha{=}\tau) \in Q_2.\ \theta(\alpha) = \theta(\tau)$ (**4**). Therefore, by (3,4) $\forall(\alpha{=}\tau) \in (Q_1 \cup Q_2).\ \theta(\alpha) = \theta(\tau)$, and thus $\theta \vDash Q_1 \cup Q_2$ (**5**) (and $\vDash Q_1 \cup Q_2$)). By definition, (5) implies $\langle Q_1 \cup Q_2\rangle \sqsubseteq \theta$. □

Type equivalence is complete.

Proof. (*of Theorem C.35*) We have $\theta(\tau_1) = \theta(\tau_2)$ (**1**). We proceed by induction over the shape of τ_1, τ_2.

Case $[\theta(\tau) = \theta(\tau)]$: We have $\tau = \tau_1 = \tau_2$, and thus by [eq-id], $\varnothing \vdash \tau_1 \approx \tau_2$, and $\varnothing \sqsubseteq \theta$ (since $\theta = \theta \circ \mathit{id}$).

Case $[\theta(\alpha) = \theta(\tau_2)]$: If $\alpha = \tau_2$, we have the previous case $\theta(\tau) = \theta(\tau)$. Since θ is idempotent, we otherwise have $\alpha \notin \mathrm{ftv}(\tau_2)$ (**2**). By [eq-var], $\{\alpha{=}\tau_2\} \vdash \alpha \approx \tau_2$. Moreover, by (2) and Property C.19.3), we also have $[\alpha{:=}\tau_2] \sqsubseteq \theta$.

Case $[\theta(\tau_1) = \theta(\alpha)]$: as the previous case with [mvarr].

Case $[\theta(\tau_1 {\to} \tau_2) = \theta(\tau_1' {\to} \tau_2')]$: We have $\theta(\tau_1) = \theta(\tau_1')$ and $\theta(\tau_2) = \theta(\tau_2')$. By ind. hyp. $Q_1 \vdash \tau_1 \approx \tau_1'$ (**2**) with $Q_1 \sqsubseteq \theta$ and $Q_2 \vdash \tau_2 \approx \tau_2'$ (**3**) with $Q_2 \sqsubseteq \theta$. By lemma C.27, we have $\vDash Q_1 \cup Q_2$ (**3**) and $Q_1 \cup Q_2 \sqsubseteq \theta$, and thus by (2,3,4), $Q_1, Q_2 \vdash \tau_1 {\to} \tau_2 \approx \tau_1' {\to} \tau_2'$.

□

C.6 Computing Prefixes

We can replace sub-prefixes with equivalent constraints:

Lemma C.28. (*Prefix replacement*)
If $Q_1 \equiv Q_2$, then $Q \cup Q_1 \;\equiv\; Q \cup Q_2$.

Proof. (*of Lemma C.28*) We have $Q_1 \;\equiv\; Q_2$ and by definition $\langle Q_1\rangle = \langle Q_2\rangle$ (**1**) (and need to show $Q \cup Q_1 \;\equiv\; Q \cup Q_2$). Since $Q_1 \sqsubseteq Q \cup Q_1$, we have from Lemma C.21, $\langle Q \cup Q_1\rangle \;=\; \theta_1 \circ \langle Q_1\rangle$ (**2**) and similarly, $\langle Q \cup Q_2\rangle \;=\; \theta_2 \circ \langle Q_2\rangle$ (**3**) for some $\theta_1,\ \theta_2$.

By definition, θ_1 is a least substitution such that $\forall(\alpha{=}\tau) \in Q.\ (\theta_1 \circ \langle Q_1\rangle)(\alpha) \;=\; (\theta_1 \circ \langle Q_1\rangle)(\tau)$ (**4**) $\wedge\ \forall(\alpha{=}\tau) \in Q_1.\ (\theta_1 \circ \langle Q_1\rangle)(\alpha) \;=\; (\theta_1 \circ \langle Q_1\rangle)(\tau)$. The second part induces no constraints on θ_1 since it holds for any substitution, so θ_1 only needs to be the least substitution such that (4) holds. Similarly, θ_2 is the least substitution such that $\forall(\alpha{=}\tau) \in Q.\ (\theta_2 \circ \langle Q_2\rangle)(\alpha) \;=\; (\theta_2 \circ \langle Q_2\rangle)(\tau)$ (**5**) holds. With (1,5), θ_2 is also the least substitution for $\forall(\alpha{=}\tau) \in Q.\ (\theta_2 \circ \langle Q_1\rangle)(\alpha) \;=\; (\theta_2 \circ \langle Q_1\rangle)(\tau)$ and with (4) it follows that $\theta_1 \equiv \theta_2$ (**6**). We can now derive:

$$\begin{array}{ll} & \langle Q \cup Q_1\rangle \\ = & \theta_1 \circ \langle Q_1\rangle \;\;\{\ (2)\ \} \\ = & \theta_1 \circ \langle Q_2\rangle \;\;\{\ (1)\ \} \\ \equiv & \theta_2 \circ \langle Q_2\rangle \;\;\{\ (6)\ \} \\ = & \langle Q \cup Q_2\rangle \;\;\{\ (3)\ \} \end{array}$$

□

Using the replacement Lemma, we can show that we can simplify duplicate bounds:

Proof. (*of Theorem 2.5*) We have $Q' \vdash\ \tau_1 \approx \tau_2$ (**1**) By (1) and Lemma C.34 and C.35, we have that $\langle Q'\rangle$ is the least substitution such that $Q'[\tau_1] = Q'[\tau_2]$. Since $\alpha \notin \mathrm{ftv}(Q', \tau_1, \tau_2)$ (Lemma ??) and Lemma 3.7, we have that $\theta_1 \;=\; \langle Q' \cup \{\alpha{=}\tau_1\}\rangle = \;\langle Q' \cdot \alpha{=}\tau_1\rangle = \;\langle Q'\rangle \circ [\alpha{:=}\tau_1]$, and thus θ_1 is a least solution such that $\theta_1(\alpha) = \theta_1(\tau_1) = \theta_1(\tau_2)$. Writing $Q_0 \;=\; \{\alpha{=}\tau_1, \alpha{=}\tau_2\}$, $\langle Q_0\rangle$ is also a least solution such that $Q_0[\alpha] = Q_0[\tau_2] = Q_0[\tau_2]$, and we have $\langle Q_0\rangle \equiv \theta_1$, and thus $Q_0 \;\equiv\; Q' \cup \{\alpha{=}\tau_1\}$. It follows from Lemma C.28 that $Q \cup Q_0 \;\equiv\; Q \cup Q' \cup \{\alpha{=}\tau_1\}$. □

Lemma C.29.
If $\vDash Q$, we can simplify Q to an equivalent mapping $\lfloor Q\rfloor$.

Proof. (*of Lemma C.29*) Since $\vDash Q$, for any duplicate binding $\{\alpha{=}\tau_1, \alpha{=}\tau_2\} \;\subseteq Q$, we have $\langle Q\rangle[\tau_1] = \langle Q\rangle[\tau_2]$ and by Lemma C.35, $Q' \vdash\ \tau_1 \approx \tau_2$ (for some $Q' \sqsubseteq Q$), and by Theorem 2.5 we can simplify the duplicate binding to $Q' \cup \{\alpha{=}\tau_1\}$ (with $\alpha \notin \mathrm{ftv}(Q')$). Repeated application reduces Q to a mapping $\lfloor Q\rfloor$.

Lemma C.30.
If $\vDash Q$, we can order all bindings in $\lfloor Q\rfloor$ as $\alpha_1{=}\tau_1 \cdot \ldots \cdot \alpha_n{=}\tau_n$.

Proof. (*of Lemma C.30*) If there is no possible order in the bounds of a mapping $\lfloor Q\rfloor$, there must a subset $Q' \subseteq \lfloor Q\rfloor$ where for all bounds $\alpha{=}\tau \in Q'$, $\alpha \in \mathrm{ftv}(Q', \tau)$. Since this is a cyclic dependency, such Q' has no solution (with an idempotent substitution). This contradicts our assumption that Q is consistent.

C.7 Solving Prefixes

We can show that *solve* (see Section 3.1.1) is sound, complete, and terminating:

Theorem C.31. (*Solve is sound*)
If $\mathit{solve}(Q) = \theta$ then $\theta \vDash Q$.

Theorem C.32. (*Solve is complete*)
If $\theta \vDash Q$, then $\mathit{solve}(Q) = \theta'$ with $\theta' \sqsubseteq \theta$ (and thus, $\theta' = \langle Q \rangle$).

Theorem C.33. (*Solve terminates*)
For any Q, $\mathit{solve}(Q)$ terminates.

Theorem C.34. (*Equiv is sound*)
If $\mathit{equiv}(\tau_1, \tau_2) = \theta$ then $\theta\tau_1 = \theta\tau_2$.

Theorem C.35. (*Equiv is complete*)
If $\theta\tau_1 = \theta\tau_2$, then $\mathit{equiv}(\tau_1, \tau_2) = \theta'$ with $\theta' \sqsubseteq \theta$.

C.8 Stable Derivations

Lemma C.36. (*Derivations are stable under more general environments*)
If $F \mid Q \mid \Gamma, x : \sigma \vdash e : \tau$ with $Q \sqsubseteq \theta$ and $\theta\sigma' \sqsubseteq \sigma$, we also have $F, F' \mid Q \mid \Gamma, x : \sigma' \vdash e : \tau'$ with some fresh F' and a θ' such that $(\theta' \circ \theta)\tau' = \tau$ with $\mathrm{dom}(\theta') \subseteq F'$.

Proof. (*Of Lemma C.36*) We have a derivation $F \mid Q \mid \Gamma, x : \sigma \vdash e : \tau$. If we bind x with a more general σ' (with $\theta\sigma' \sqsubseteq \sigma$), we get at a leaf $\varnothing \mid \varnothing \mid \Gamma, x : \sigma' \vdash x : \sigma'$ (instead of σ). We can now use [INST] to instantiate any outer quantifiers $\overline{\alpha}$ (as $\forall\overline{\alpha}.\sigma''$) (using the fresh F'). Therefore, there might be extra prefix constraints for $\overline{\alpha}$ (due to $\approx$ in [APP]). However, eventually we can use [GENSUB] to remove those (since $\overline{\alpha} \not\Cap \mathrm{ftv}(F, \Gamma)$). This may still leave some unconstrained $\alpha \in \overline{\alpha}$ in $\mathrm{ftv}(\tau')$ and we may need to apply a substitution θ' eventually (with $\mathrm{dom}(\theta') \subseteq F'$) such that $(\theta' \circ \theta)\tau' = \tau$. (todo: formalize this proof?)

C.9 Completeness of the Type Rules

We use x_λ for lambda bound variables, and x_{let} for let-bound variables. We can decompose a HM type environment into a substitution and HMQ environment where all lambda-bound variables are bound to a (fresh) variable:

$$\begin{array}{lll}
\varnothing & \cong id(\varnothing) & \\
\Gamma, x_\lambda : \tau & \cong (\theta \circ [\alpha{:=}\tau])(\Gamma', x_\lambda : \alpha) \;\; \text{where } \Gamma \cong \theta\Gamma',\ \text{fresh } \alpha & [\textsc{split-mono}] \\
\Gamma, x_{\text{let}} : \sigma & \cong \theta(\Gamma', x_{\text{let}} : \sigma) \;\; \text{where } \Gamma \cong \theta\Gamma' & [\textsc{split-poly}]
\end{array}$$

We can now state completeness as:

Theorem C.37. (*Completeness & most general typings*)
If $\Gamma \vdash_{\text{HM}} e : \sigma$ with $\Gamma \cong \theta_0\Gamma'$, then we also have $F \mid Q \mid \Gamma' \vdash e : \sigma'$ for some $F \not\pitchfork \text{ftv}(\Gamma')$, and $\theta = \theta' \circ \theta_0$ with $\text{dom}(\theta') \subseteq F$ and $\text{codom}(\theta') \subseteq \text{ftv}(Q, \sigma')$, such that $Q \sqsubseteq \theta$ and $\theta\sigma' \sqsubseteq \sigma$.

For the inductive proof we have strengthened the earlier Theorem 2.4 to decompose Γ into θ_0 and Γ', where θ' only substitutes fresh variables ($\text{dom}(\theta') \subseteq F$).

Proof. (*of Theorem C.37*) We have $\Gamma \vdash_{\text{HM}} e : \sigma$ (**1**) with $\Gamma \cong \theta_0\Gamma'$ (**1a**), and we need to show $F \mid Q \mid \Gamma' \vdash e : \sigma'$ (I) for some $\theta = \theta' \circ \theta_0$ with $Q \sqsubseteq \theta$ (II), and $\theta\sigma' \sqsubseteq \sigma$ (III) (and $\text{dom}(\theta') \subseteq F$ and $\text{codom}(\theta') \subseteq \text{ftv}(Q, \sigma')$ (IV)).

We proceed by induction on the typing rule $\Gamma \vdash_{\text{HM}} e : \sigma$:

Case [$\textsc{var}_{\textsc{hm}}$]: We have $x : \sigma \in \Gamma$ (**2**). With $\theta = \theta_0$ (**3**) (and $\theta' = id$ (**4**)) , we consider two cases: $x_\lambda : \tau \in \Gamma$ and $x_{\text{let}} : \sigma \in \Gamma$. For a let-bound x_{let}, we have:

$$\begin{array}{lll}
 & x : \sigma \in \Gamma' & \{\ [\textsc{split-poly}]\ \} \\
\Rightarrow & \varnothing \mid \varnothing \mid \Gamma' \vdash x : \sigma & \{\ [\textsc{var}],\ (\textbf{I})\ \}
\end{array}$$

where $\varnothing \sqsubseteq \theta$ (**II**) holds by definition, and $\theta\sigma = \sigma$ (3,[$\textsc{split-poly}$]) and thus $\theta\sigma \sqsubseteq \sigma$ (**III**). Otherwise, a lambda-bound $x_\lambda : \tau \in \Gamma$, and thus $x_\lambda : \alpha \in \Gamma'$ with $\theta\alpha = \tau$ (**5**) [$\textsc{split-mono}$], and we can derive:

$$\begin{array}{lll}
 & x : \alpha \in \Gamma' & \{\ [\textsc{split-mono}]\ \} \\
\Rightarrow & \varnothing \mid \Gamma' \vdash x : \alpha & \{\ [\textsc{var}],\ (\textbf{I})\ \}
\end{array}$$

where $\varnothing \sqsubseteq \theta$ (**II**) always holds, and by (5) $\theta\alpha \sqsubseteq \tau$ (**III**). For both cases, by (4), $\text{dom}(\theta') = \varnothing \subseteq F$ and $\text{codom}(\theta') \subseteq \text{ftv}(Q, \sigma)$ (**IV**).

Case [$\textsc{fun}_{\textsc{hm}}$]: We have $\Gamma, x : \tau_1 \vdash_{\text{HM}} e : \tau_2$ with $\Gamma, x : \tau_1 \cong \theta_0(\Gamma', x : \alpha)$ (**2**) (with a fresh $\alpha \notin (F \cup \text{ftv}(\Gamma, \Gamma', \tau_2, \tau_2'))$ (**2a**)), $\theta_0\alpha = \tau_1$ (**2b**), and by induction $F \mid Q \mid \Gamma', x : \alpha \vdash e : \tau_2'$ (**3**) with $\theta = \theta' \circ \theta_0$ (**3a**), $\text{dom}(\theta') \subseteq F$ and $\text{codom}(\theta') \subseteq \text{ftv}(Q, \tau_2')$ (**3b**), such that $Q \sqsubseteq \theta$ (**3c**) and $\theta\tau_2' \sqsubseteq \tau_2$ (**3d**). We can now derive:

$$\begin{array}{lll}
 & F \mid Q \mid \Gamma', x : \alpha \vdash e : \tau_2' & \{\ (3)\ \} \\
\Rightarrow & F, \alpha \mid Q \mid \Gamma' \vdash e : \alpha \rightarrow \tau_2' & \{\ [\textsc{fun}],\ (2a),\ (\textbf{I})\ \}
\end{array}$$

From (3c), we have directly $Q \sqsubseteq \theta$ (**II**).

$$\begin{array}{lll}
 & \theta(\alpha \rightarrow \tau_2') & \\
= & \theta\alpha \rightarrow \theta\tau_2' & \{\ \textit{subst.}\ \} \\
\sqsubseteq & \theta\alpha \rightarrow \tau_2 & \{\ (3d)\ \} \\
= & \theta'(\theta_0\alpha) \rightarrow \tau_2 & \{\ (3a)\ \} \\
= & \theta'\tau_1 \rightarrow \tau_2 & \{\ (2b)\ \} \\
= & \tau_1 \rightarrow \tau_2 & \{\ (3b),\ (\textbf{III})\ \}
\end{array}$$

From (3b), it follows $\text{dom}(\theta') \subseteq F$ and $\text{codom}(\theta') \subseteq \text{ftv}(Q, \alpha{\rightarrow}\tau_2')$ (**IV**)

Case [$\textsc{app}_{\textsc{hm}}$]: By (1), we have $\Gamma \vdash_{\text{HM}} e_1 : \tau_2 \rightarrow \tau$ (**2a**) and $\Gamma \vdash e_2 : \tau_2$ (**2b**), and by induction $F_1 \mid Q_1 \mid \Gamma' \vdash e_1 : \tau_3$ (**3a**) and $F_2 \mid Q_2 \mid \Gamma' \vdash e_2 : \tau_4$ (**4a**), where we pick $F_1 \not\pitchfork F_2$ (**2c**). For the first derivation, we have $\theta_1 = \theta_1' \circ \theta_0$, $Q_1 \sqsubseteq \theta_1$, $\theta_1\tau_3 \sqsubseteq \tau_2 \rightarrow \tau$, $\text{dom}(\theta_1') \subseteq F_1$, and $\text{codom}(\theta_1') \subseteq \text{ftv}(Q_1, \tau_3)$ (**3b**). For the second derivation, $\theta_2 = \theta_2' \circ \theta_0$, $Q_2 \sqsubseteq \theta_2$, $\theta_2\tau_4 \sqsubseteq \tau_2$, $\text{dom}(\theta_2') \subseteq F_2$, and $\text{codom}(\theta_2') \subseteq \text{ftv}(Q_2, \tau_4)$

(**4b**).

We can now derive:

$$\begin{array}{lll} & \mathsf{codom}(\theta'_1) \subseteq \mathsf{ftv}(Q_1, \tau_3) & \{\ (3b)\ \} \\ \Rightarrow & \mathsf{codom}(\theta'_1) \subseteq \mathsf{ftv}(F_1, \Gamma') & \{\ \textit{Lemma B.10}\ \} \\ \Rightarrow & \mathsf{codom}(\theta'_1) \not\pitchfork F_2 & \{\ (2c),\ F_2 \not\pitchfork \mathsf{ftv}(\Gamma').\ \} \end{array}$$

and similarly $\mathsf{codom}(\theta'_2) \not\pitchfork F_1$. Moreover $\mathsf{dom}(\theta'_1) \subseteq F_1$ (3b) implies with (2c) that $\mathsf{dom}(\theta'_1) \not\pitchfork F_2$. A similar argument holds for θ'_2 and therefore the (co)domains of θ'_1 and θ'_2 are disjoint, and thus $\theta'_1 \circ \theta'_2 \ = \ \theta'_2 \circ \theta'_1$ (**5**). As an aside, this property is why we need fresh names as otherwise sharing between the two sub-derivations can occur and (5) would not hold.

We can now define $\theta' \ = \ \theta'_1 \circ \theta'_2 \circ \theta_0$ (**2d**), and derive:

$$\begin{array}{lll} & Q_2 \sqsubseteq \theta_2 & \{\ (4b)\ \} \\ = & Q_2 \sqsubseteq \theta'_2 \circ \theta_0 & \{\ (4b)\ \} \\ \Rightarrow & Q_2 \sqsubseteq \theta'_1 \circ \theta'_2 \circ \theta_0 & \{\ \textit{def.}\ \} \\ = & Q_2 \sqsubseteq \theta' & \{\ (2d),\ (\mathbf{4c})\ \} \end{array}$$

Since $\mathsf{dom}(\theta'_1) \not\pitchfork \mathsf{codom}(\theta_2)$ (4b), we have $\theta'\tau_4 \ = \ \theta_2\tau_4$ (4b) and thus $\theta'\tau_4 \sqsubseteq \tau_2$ (**4d**). By (5), we also have $\theta' \ = \ \theta'_2 \circ \theta'_1 \circ \theta_0$, and we can use the same reasoning for the left derivation to conclude $Q_1 \sqsubseteq \theta'$ (**3c**) and $\theta'\tau_3 \sqsubseteq \tau_2 \to \tau$ (**3d**).

From (3d,4d) it follows that $\theta'\tau_3 = \tau_2 \to \tau$ and $\theta'\tau_4 = \tau_2$ since these are monotypes. Therefore, $\theta'\tau_3 \ = \ \theta'\tau_4 \to \tau$. We now define $\theta \ = \ [\alpha{:=}\tau] \circ \theta'$ for some fresh $\alpha \notin F_1, F_2$ (**6**). Since α is fresh and Lemma B.10, we have $\theta\tau_3 \ = \ \theta(\tau_4 \to \alpha)$, and from Theorem C.35 it follows $Q_3 \vdash \tau_3 \approx \tau_4 \to \alpha$ (**7**), with $Q_3 \sqsubseteq \theta$ (**7a**).

From (3c,6), we have $Q_1 \sqsubseteq \theta$ and by (4c,6) $Q_2 \sqsubseteq \theta$. With (7a), and Lemma C.24 we have Q_1, Q_2, Q_3 is consistent with $Q_1, Q_2, Q_3 \sqsubseteq \theta$ (**II**). Together with (3a,4a,7), we can now use [APP] to conclude $F_1, F_2, \alpha \mid Q_1, Q_2, Q_3 \mid \Gamma' \vdash e_1\ e_2 \ : \ \alpha$ (**I**) and by (6), $\theta\alpha \sqsubseteq \tau$ (**III**). Finally, by (3b,4b,6), we also have $\mathsf{dom}(\theta) \subseteq F_1, F_2, \alpha$ with $\mathsf{codom}(\theta) \subseteq \mathsf{ftv}((Q_1, Q_2, \alpha), \alpha)$ (**IV**).

Case [LET$_{\text{HM}}$]: We have $\Gamma \vdash_{\text{HM}} e_1 \ : \ \sigma$ with $\Gamma \cong \theta_0\Gamma'$ (**2**), and $\Gamma, x{:}\sigma \vdash_{\text{HM}} e_2 \ : \ \tau$ (**3**). By induction, we have $F_1 \mid Q_1 \mid \Gamma' \vdash e_1 \ : \ \sigma_1$ (**4a**) and $F_2 \mid Q_2 \mid \Gamma', x{:}\sigma \vdash e_2 \ : \ \tau_2$ (**5a**), with $F_1 \not\pitchfork F_2$ (**6**). For the left derivation, we have $\theta_1 = \theta'_1 \circ \theta_0$, $Q_1 \sqsubseteq \theta_1$, $\theta_1\sigma_1 \sqsubseteq \sigma$, $\mathsf{dom}(\theta'_1) \subseteq F_1$, and $\mathsf{codom}(\theta'_1) \subseteq \mathsf{ftv}(Q_1, \sigma_1)$ (**4b**). For the right derivation, we similarly have $\theta_2 = \theta'_2 \circ \theta_0$, $Q_2 \sqsubseteq \theta_2$, $\theta_2\tau_2 \sqsubseteq \tau$, $\mathsf{dom}(\theta'_2) \subseteq F_2$, and $\mathsf{codom}(\theta'_2) \subseteq \mathsf{ftv}(Q_2, \tau_2)$ (**5b**).

We now proceed as in the [APP$_{\text{HM}}$] case, and define $\theta_12 \ = \ \theta'_1 \circ \theta'_2 \circ \theta_0$ (**7**) with $\theta'_1 \circ \theta'_2 \ = \ \theta'_2 \circ \theta'_1$ (**7a**). Just as in the [APP$_{\text{HM}}$] case, it follows that $Q_1 \sqsubseteq \theta_12$ (**4c**), $Q_2 \sqsubseteq \theta_12$ (**5c**), and $\theta_12\sigma_1 \sqsubseteq \sigma$ (**4d**), and $\theta_12\tau_2 \sqsubseteq \tau$ (**5d**). From (4c,5c) it follows from Lemma C.24 that Q_1, Q_2 is consistent with $Q_1, Q_2 \sqsubseteq \theta_12$ (**8**).

We cannot yet apply [LET] as we need to satisfy the side condition $\mathsf{ftv}(\sigma_1) \subseteq \mathsf{ftv}(\Gamma')$. Suppose there is a $\alpha \in \mathsf{ftv}(\sigma_1)$ with $\alpha \notin \mathsf{ftv}(\Gamma')$. In that case we can apply either [GEN] or [GENSUB] depending on whether $\alpha \in \mathsf{ftv}(Q_1)$. If $\alpha \in \mathsf{ftv}(Q_1)$, we have by Lemma 2.6, $Q_1 \equiv Q_3 \cdot \alpha{=}\tau_3$ (**9**), and we can apply [GENSUB] to derive $F_1 \mid Q_3 \mid \Gamma' \vdash e_1 \ : \ \sigma_3$ with $\sigma_3 = [\alpha{:=}\tau_3]\sigma_1$. From (9) and Lemma 3.7, $\langle Q_1 \rangle \ = \ \langle Q_3 \rangle \circ [\alpha{:=}\tau_3]$. From (4c), this implies $\theta_12 \ = \ \theta_12 \circ [\alpha{:=}\tau_3]$ (**9a**). We can now derive:

$$\begin{array}{lll} & \theta_12\sigma_1 \sqsubseteq \sigma & \{\ (4d)\ \} \\ = & \theta_12([\alpha{:=}\tau_3]\sigma_1) \sqsubseteq \sigma & \{\ (9a)\ \} \\ = & \theta_12\sigma_3 \sqsubseteq \sigma & \{\ (\mathbf{9b})\ \} \end{array}$$

Similarly, if $\alpha \notin \mathsf{ftv}(Q_1)$ we can apply the [GEN] rule.

Therefore, after repeated application we have $F \mid Q_n \mid \Gamma' \vdash e_1 \ : \ \sigma_n$ with $Q_n \sqsubseteq \theta_12$ and $\theta_12\sigma_n \sqsubseteq \sigma$, and $Q_n, Q_2 \sqsubseteq \theta_12$ (**9c**). Furthermore, since $\theta_12\sigma_n \sqsubseteq \sigma$ and (5a), we have by Lemma C.36, $F_2, F_3 \mid Q_2 \mid \Gamma', x{:}\sigma_n \vdash e_2 \ :$ with $(\theta' \circ \theta_12)\tau'_2 \sqsubseteq \tau_2$ (**10**) as well, where F_3 is fresh and $\mathsf{dom}(\theta') \subseteq F_3$. Therefore, we can define $\theta \ = \ \theta' \circ \theta_12$ with (9c) $Q_n \sqsubseteq \theta$, $\theta\sigma_n \sqsubseteq \sigma$, and $Q_n, Q_2 \sqsubseteq \theta$ (**II**). We can now finally apply [LET] to derive

$F_1, F_2, F_3 \mid Q_n, Q_2 \mid \Gamma' \vdash \mathsf{let}\ x\ =\ e_1\ \mathsf{in}\ e_2\ :\ \tau_2'$ (**I**), with (5d,10) $\theta\tau_2' \sqsubseteq \tau$ (**III**) (and like the [$\textsc{app}_{\textsc{hm}}$] case, $\mathsf{dom}(\theta) \subseteq \mathsf{ftv}(F_1, F_2, F_3)$ and $\mathsf{codom}(\theta) \subseteq \mathsf{ftv}((Q_n, Q_2), \Gamma')$ (**IV**)).

Case [$\textsc{inst}_{\textsc{hm}}$]: We have $\Gamma \vdash_{\mathsf{HM}} e\ :\ \forall\alpha.\sigma$ with $\Gamma \cong \theta_0\Gamma'$ (**2**), and thus by induction $F \mid Q \mid \Gamma' \vdash\ e\ :\ \sigma'$ (**3**) with $\theta\ =\ \theta' \circ \theta_0$ (**3a**) such that $Q \sqsubseteq \theta$ (**3b**) and $\theta\sigma' \sqsubseteq \forall\alpha.\sigma$ (**3c**), and $\mathsf{dom}(\theta') \subseteq F$ and $\mathsf{codom}(\theta') \subseteq \mathsf{ftv}(Q, \sigma')$ (**3d**). We can assume a fresh $\alpha \notin F$ by α-renaming (**4**).
From (3c) and [$\textsc{instance}$], we must have $\sigma' = \forall\alpha.\sigma_0$ (**5**) (for some σ_0). We can thus derive:

$$\begin{array}{ll} & F \mid Q \mid \Gamma' \vdash\ e\ :\ \forall\alpha.\sigma_0\ \ \{\ (3,5)\ \} \\ \Rightarrow & F, \alpha \mid Q \mid \Gamma' \vdash\ e\ :\ \sigma_0\ \ \ \{\ (4),\ \ (\mathbf{I})\ \} \end{array}$$

From (3b), we directly have $Q \sqsubseteq \theta$ (**II**). Moreover, by (3c,5) $\theta(\forall\alpha.\sigma_0) \sqsubseteq \forall\alpha.\sigma$. Since α is fresh, we must have $\theta\sigma_0\ \sqsubseteq\ \sigma$ (**III**). Finally, from (3d) and (4), it follows directly that $\mathsf{dom}(\theta') \subseteq F, \alpha$ and $\mathsf{codom}(\theta') \subseteq \mathsf{ftv}(Q, \sigma')$ (**IV**).

Case [$\textsc{gen}_{\textsc{hm}}$]: We have $\Gamma \vdash_{\mathsf{HM}} e\ :\ \sigma$ with $\Gamma \cong \theta_0\Gamma'$ (**2**), and $\alpha \notin \mathsf{ftv}(\Gamma)$ (**3**), and thus by induction $F \mid Q \mid \Gamma' \vdash\ e\ :\ \sigma'$ (**3a**) with $\theta\ =\ \theta' \circ \theta_0$ (**3b**) such that $Q \sqsubseteq \theta$ (**3c**) and $\theta\sigma' \sqsubseteq \sigma$ (**3d**) (and $F \not\pitchfork \mathsf{ftv}(\Gamma')$ (**3e**)). From (3) and the definition of ($\cong$), we have $\alpha \notin \mathsf{ftv}(\Gamma')$ (**4**).
With $\alpha \in \mathsf{ftv}(\sigma)$, then from (3d) we have $\alpha \in \mathsf{ftv}(\theta\sigma')$ (**5a**) and thus $\alpha \notin \mathsf{dom}(\theta)$ (**5c**). We now have three cases to consider:

A. Suppose $\alpha \notin \mathsf{ftv}(Q)$, in that case we can apply [$\textsc{gen}$] with (3a,4) and derive $F \mid Q \mid \Gamma' \vdash\ e\ :\ \forall\alpha.\sigma'$ (**I**) with (3c) $Q \sqsubseteq \theta$ (**II**) and (3d,5c) $\theta(\forall\alpha.\sigma') \sqsubseteq \forall\alpha.\sigma$ (**III**). Todo: (**IV**).
B. Suppose we have $\alpha \in \mathsf{dom}(Q)$, in that case by (3c), $\alpha \in \mathsf{dom}(\theta)$ but that contradicts (5c).
C. Otherwise, we must have $\alpha \in \mathsf{codom}(Q)$. With Theorem C.25, this implies $Q\ \equiv\ Q_1 \cdot \beta{=}\tau$ (**6a**) with $\alpha \in \mathsf{ftv}(\tau)$ (**6b**) and $\beta \notin \mathsf{ftv}(Q_1, \tau)$ (**6c**).
   Suppose $\beta \in \mathsf{ftv}(\Gamma')$. If $x_\lambda : \beta \in \Gamma'$, then with (3c) we have $x_\lambda : \tau \in \Gamma$ with $\alpha \in \mathsf{ftv}(\Gamma)$ (6b) which contradicts (3). Otherwise, $x_{\mathsf{let}} : \sigma_0 \in \Gamma'$ with $\beta \in \mathsf{ftv}(\sigma_0)$ which implies by (3b,3c) that $\beta \in \mathsf{dom}(\theta')$, and thus $\beta \in F$ – but that contradicts (3e). Therefore, we must have $\beta \notin \mathsf{ftv}(\Gamma')$ (**7**). With (6c,7) we can now use [$\textsc{gensub}$] to derive $F \mid Q_1 \mid \Gamma' \vdash\ e\ :\ [\beta{:=}\tau]\sigma'$, where (3c,6a) $Q_1 \sqsubseteq \theta$, and $\theta \circ [\beta{:=}\tau]\ =\ \theta$, and thus $\theta[\beta{:=}\tau]\sigma'\ \sqsubseteq\ \sigma$. We can repeatedly apply [$\textsc{gensub}$] until case (A) applies.

□

C.10 Soundness of the Type Rules

Theorem 2.3. (*Soundness*)
If $F \mid Q \mid \Gamma \vdash\ e : \sigma$, then we can also derive $Q[\Gamma] \vdash_{\mathrm{HM}} e : Q[\sigma]$.

We use the substitution Lemma from HM in the soundness proof:

Lemma C.39. (*Weakening of HM*)
If $\Gamma \vdash_{\mathrm{HM}} e : \sigma$ then also $\theta\Gamma \vdash_{\mathrm{HM}} e : \theta\sigma$.

We often use this Lemma when $Q_1 \sqsubseteq Q$, and $Q_1[\Gamma] \vdash_{\mathrm{HM}} e : Q_1[\sigma]$, then we also have $Q[\Gamma] \vdash_{\mathrm{HM}} e : Q[\sigma]$.

Proof. (*of Theorem 2.3*) By induction on the typing rules of $F \mid Q \mid \Gamma \vdash\ e : \sigma$:

Case [$\textsc{var}$]: We have $x\!:\!\sigma \in \Gamma$ (**1**) and $Q = \varnothing$ (**2**), and thus $Q[\Gamma] = \Gamma$ (**3**) and $Q[\sigma] = \sigma$ (**4**).

$$\begin{array}{lll} & x\!:\!\sigma\ \in \Gamma & \{\ (1)\ \} \\ \Rightarrow & \Gamma \vdash_{\mathrm{HM}} x : \sigma & \{\ [\textsc{var}_{\mathrm{HM}}]\ \} \\ = & Q[\Gamma] \vdash_{\mathrm{HM}} x : \Gamma[\sigma] & \{\ (3,4)\ \} \end{array}$$

Case [$\textsc{fun}$]: We have $F \mid Q \mid \Gamma, x\!:\!\alpha \vdash\ e : \tau$ (**1**) (with $\alpha \notin F$). We can now derive:

$$\begin{array}{lll} & Q[\Gamma, x\!:\!\alpha] \vdash_{\mathrm{HM}} e : Q[\tau] & \{\ \textit{induction over}\ (1)\ \} \\ = & Q[\Gamma], x\!:\!Q[\alpha] \vdash_{\mathrm{HM}} e : Q[\tau] & \{\ \textit{def.}\ \} \\ \Rightarrow & Q[\Gamma] \vdash_{\mathrm{HM}} \lambda x.e : Q[\alpha] \to Q[\tau] & \{\ [\textsc{fun}_{\mathrm{HM}}]\ \} \\ = & Q[\Gamma] \vdash_{\mathrm{HM}} \lambda x.e : Q[\alpha \to \tau] & \{\ \textit{def.}\ \} \end{array}$$

Case [$\textsc{app}$]: We have $F_1 \mid Q_1 \mid \Gamma \vdash\ e_1 : \tau_1$ (**1**) and $F_2 \mid Q_2 \mid \Gamma \vdash\ e_2 : \tau_2$ (**2**) with $Q_3 \vdash\ \tau_1 \approx \tau_2 \to \alpha$ (**3**) and $\models Q_1, Q_2, Q_3$ (**4**). Writing $Q = Q_1, Q_2, Q_3$, we have by Lemma C.21, $Q_1 \sqsubseteq Q$, $Q_2 \sqsubseteq Q$, and $Q_3 \sqsubseteq Q$ (**6**).

$$\begin{array}{lll} & Q_1[\Gamma] \vdash_{\mathrm{HM}} e_1 : Q_1[\tau_1] & \{\ \textit{induction over}\ (1)\ \} \\ \Rightarrow & Q[\Gamma] \vdash_{\mathrm{HM}} e_1 : Q[\tau_1] & \{\ \textit{Lemma C.39},\ (6),\ (7)\ \} \end{array}$$

and also:

$$\begin{array}{lll} & Q_2[\Gamma] \vdash_{\mathrm{HM}} e_2 : Q_2[\tau_2] & \{\ \textit{induction over}\ (2)\ \} \\ \Rightarrow & Q[\Gamma] \vdash_{\mathrm{HM}} e_2 : Q[\tau_2] & \{\ \textit{Lemma C.39},\ (6),\ (\mathbf{8})\ \} \end{array}$$

Furthermore:

$$\begin{array}{lll} & Q_3 \vdash\ \tau_1 \approx \tau_2 \to \alpha & \{\ (3)\ \} \\ \Rightarrow & Q_3[\tau_1]\ =\ Q_3[\tau_2 \to \alpha] & \{\ \textit{Theorem C.34}\ \} \\ \Rightarrow & Q[\tau_1]\ =\ Q[\tau_2 \to \alpha] & \{\ (6)\ \} \\ \Rightarrow & Q[\tau_1]\ =\ Q[\tau_2] \to Q[\alpha] & \{\ \textit{def.}\ \} \end{array}$$

and by (7), $Q[\Gamma] \vdash_{\mathrm{HM}} e_1 : Q[\tau_2] \to Q[\alpha]$, and with [$\textsc{app}_{\mathrm{HM}}$] and (8), we have $Q[\Gamma] \vdash_{\mathrm{HM}} e_1\ e_2 : Q[\alpha]$.

Case [$\textsc{gensub}$]: We have $F \mid Q \cdot \alpha{=}\tau \mid \Gamma \vdash\ e : \sigma$ (**1**) with $\alpha \notin \mathrm{ftv}(Q, \Gamma)$ (**2**). Writing Q' for $Q \cdot \alpha{=}\tau$, we can derive:

$$\begin{array}{lll} & Q'[\Gamma] \vdash_{\mathrm{HM}} e : Q'[\sigma] & \{\ \textit{induction over}\ (1)\ \} \\ = & Q[[\alpha{:=}\tau]\Gamma] \vdash_{\mathrm{HM}} e : Q[[\alpha{:=}\tau]\sigma] & \{\ \textit{Lemma 3.7}\ \} \\ = & Q[\Gamma] \vdash_{\mathrm{HM}} e : Q[[\alpha{:=}\tau]\sigma] & \{\ (2)\ \} \end{array}$$

Case [$\textsc{gen}$]: We have $F \mid Q \mid \Gamma \vdash\ e : \sigma$ (**1**) with $\alpha \notin \mathrm{ftv}(Q, \Gamma)$ (**2**). We can derive:

$$\begin{array}{lll} & Q[\Gamma] \vdash_{\mathrm{HM}} e : Q[\sigma] & \{\ \textit{induction over}\ (1)\ \} \\ \Rightarrow & Q[\Gamma] \vdash_{\mathrm{HM}} e : \forall\alpha.Q[\sigma] & \{\ [\textsc{gen}_{\mathrm{HM}}],\ (2)\ \} \\ = & Q[\Gamma] \vdash_{\mathrm{HM}} e : Q[\forall\alpha.\sigma] & \{\ (2)\ \} \end{array}$$

Case [INST]: We have $F \mid Q \mid \Gamma \vdash\ e\ :\ \forall\alpha.\sigma$ (**1**) with $\alpha \notin F$ (**2**), and $F, \alpha \not\cap \mathrm{ftv}(Q, \Gamma)$ and thus $\alpha \notin \mathrm{ftv}(Q, \Gamma)$ (**3**). We can derive:

$$\begin{array}{lll}
 & Q[\Gamma] \vdash_{\mathsf{HM}} e\ :\ Q[\forall\alpha.\sigma] & \{\ \textit{induction over}\ (1)\ \} \\
= & Q[\Gamma] \vdash_{\mathsf{HM}} e\ :\ \forall\alpha.Q[\sigma] & \{\ (3)\ \} \\
\Rightarrow & Q[\Gamma] \vdash_{\mathsf{HM}} e\ :\ [\alpha{:=}\alpha](Q[\sigma]) & \{\ [\textsc{inst}_{\textsc{hm}}],\ (3)\ \} \\
= & Q[\Gamma] \vdash_{\mathsf{HM}} e\ :\ Q[\sigma] & \{\ \textit{def.}\ \}
\end{array}$$

Case [LET]: We have $F_1 \mid Q_1 \mid \Gamma \vdash\ e_1\ :\ \sigma$ (**1**) and $F_2 \mid Q_2 \mid \Gamma, x : \sigma \vdash\ e_2\ :\ \tau$ (**2**) with $\mathrm{ftv}(\sigma) \subseteq \mathrm{ftv}(\Gamma)$ (**3**) and $\vDash Q_1, Q_2$ (**4**). Writing $Q = Q_1, Q_2$, by Lemma C.21, we also have $Q_1 \sqsubseteq Q$ and $Q_2 \sqsubseteq Q$ (**5**). We can derive:

$$\begin{array}{lll}
 & Q_1[\Gamma] \vdash_{\mathsf{HM}} e_1\ :\ Q_1[\sigma] & \{\ \textit{ind. over}\ (1)\ \} \\
\Rightarrow & Q[\Gamma] \vdash_{\mathsf{HM}} e_1\ :\ Q[\sigma] & \{\ \textit{Lemma}\ C.39,\ (5),\ (\mathbf{6})\ \}
\end{array}$$

and similarly:

$$\begin{array}{lll}
 & Q_2[\Gamma, x : \sigma] \vdash_{\mathsf{HM}} e_2\ :\ Q_2[\tau] & \{\ \textit{ind. over}\ (2)\ \} \\
\Rightarrow & Q[\Gamma, x : \sigma] \vdash_{\mathsf{HM}} e_2\ :\ Q[\tau] & \{\ \textit{Lemma}\ C.39\ (5),\ (\mathbf{7})\ \}
\end{array}$$

We can now apply [LET$_{\textsc{hm}}$] to derive $Q[\Gamma] \vdash_{\mathsf{HM}} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ :\ Q[\tau]$.

□

Created with Madoko.net.