Elaboration on HM(X): Type Inference with Constraint Types

Marius Weidner

Chair of Programming Languages, University of Freiburg
weidner@cs.uni-freiburg.de

Abstract. We investigate HM(X) [[CITE]], a family of type systems designed to accommodate polymorphism, full type inference and constraint types. HM(X) extends the Hindley-Milner type system (HM) [[CITE]], which already limits System F, to ensure decidability and unambiguity of full type inference. The constraint system X utilized in HM(X) remains abstract, allowing instantiating X with arbitrary constraint systems that meet specific criteria. This abstraction empowers HM(X) to serve as a model for analyzing various constraint-related type features commonly encountered in practice. Notable examples include subtyping, substructural types and type classes. HM(X) encompasses a sound and complete type inference algorithm that remains independent of the actual constraint system X. As a result, the work for proving theoretical properties and designing inference algorithms for novel constraint-based type systems within an HM context is notably simplified.

Table of Contents

1 Introduction

1.1 Polymorphism and Full Type Inference in HM

The HM type system represents a well known and understood typing discipline that refines System F [[CITE]] by establishing constraints that allow type inference to be decidable. HM serves as the foundation for numerous real-world functional programming languages, including languages like Haskell and Rust.

In System F, we can introduce variables for both expressions and types. The type $\forall\alpha.\ T$, where α binds a new type in T, indicates that an expression of this type is polymorphic over some arbitrary type α. Unfortunately it is undecidable for arbitrary programs to determine when to introduce and eliminate $\forall$ types and in consequence type inference in System F is undecidable [[CITE]]. Thus, System F is equipped with explicit type abstraction ($\Lambda\alpha.\ e$) and type application ($e\ T$) on the syntax level.

The HM type system imposes several restrictions that make type inference decidable in a polymorphic context. Moreover, HM ensures that the most general type (the *principal type*) of a any given program is inferred. Consequently, there's no need for extra syntax to introduce or eliminate type variables. Instead, HM employs let bindings **let** $x = e_2$ **in** e_1 where e_2 is the only expression that is allowed to have a polymorphic type. Other constructs, such as applications or variables bound by a lambda abstractions, cannot inherit polymorphic types. This constraint is commonly referred to as 'let polymorphism'.

In the future, we will label polymorphic types as σ, where a $\forall$-type exists within σ, and we will refer to them as 'poly types'. On the contrary, all other types, including base types, functions, and type variables, will be designated as 'mono types' and denoted as τ.

Poly types are further constrained to exclusively permit $\forall$ binders at the top level. Therefore, a type like $\forall\alpha.\alpha \to \forall\alpha.\alpha$ would not be a valid poly type. Consequently, we establish that poly types consistently adhere to the structure $\forall\bar{\alpha}.\ \tau$, where $\bar{\alpha} = \alpha_1, .., \alpha_n$. By upholding these two regulations, that is let polymorphism and the exclusion of higher-order polymorphism, type inference remains decidable and yields a principal type.

Example 1 (Concatination of Lists).

```
concat : ∀α. [α] → [α] → [α]
concat = λ[].      λys. ys
         λ[x:xs]. λys. x : (concat xs ys)
```

Examples assume the extension with various language features such as lists and pattern matching. For convenience, type annotations are given for inferred function types.

In this example, HM would be capable of deducing the type $\forall\alpha.\ [\alpha] \to [\alpha] \to [\alpha]$ for the function 'concat'. This type is in fact the most general type for 'concat'.

1.2 Introducing Constraints on Types

Although parametric polymorphism is already a powerful abstraction, there are instances where we desire to constrain type variables solely to instantiations to types that satisfy specific constraints. We refer to such types as *constraint types*. Constraints can exhibit various different forms depending on the type features present in the actual language. As an illustration, consider $\text{HM}_{\mathcal{R}}$, that is HM extended with polymorphic records [[CITE]]. In this scenario, it becomes valuable to have the capability to specify that a type variable α should solely be instantiated as a record type containing specific fields.

Example 2 (Alternative to Selector Syntax on Records for Field 'key').

```
key : ∀α,β.(α ≤ {key : β}) ⇒ α → β
key = λ{key : β, ..}. key
```

This example simulates the extraction of a particular field from a record, conventionally represented as $e.l$ where l denotes a label. We simulate the extraction of the field 'key'. Instead of the notation $e.key$, the alternative syntax $key\ e$ could then be employed.[1] We introduce two type variables, namely α and β. While β exhibits parametric polymorphism, α functions as a constraint type. The constraint imposed on α, denoted as $\alpha \leq \{\text{key} : \beta\}$, signifies that α is exclusively permitted to take on a type that corresponds to a record featuring a 'key' field of type β. For introducing constraint C on type variables α, we will adopt the notation $\forall\bar{\alpha}.C \Rightarrow \tau$. Multiple constraints will be combined to a single constraint using conjunction.

Naturally, we can envision entirely different constraints as well. Consider a language with overloading and overloading constraints $\text{HM}_{\mathcal{O}}$ [[CITE]]. In $\text{HM}_{\mathcal{O}}$, constraints are structured in the form $x : \alpha \to \tau$, wherein an instance for the overloaded identifier x with type $\alpha \to \tau$ is expected to be present.

Example 3 (Overloading the Equality Operator for Lists).

```
eq : nat → nat → bool
eq = λ0       λ0       = ⊤
     λsuc n. λsuc m. = eq n m
     λ_.      λ_.      = ⊥
eq : ∀α.(eq : α → α → bool) ⇒ [α] → [α] → bool
eq = λ[].      λ[].      = ⊤
     λ[x:xs]. λ[y:ys]. = eq x y ∧ eq xs ys
     λ_.       λ_.       = ⊥
```

This example considers the overloaded 'eq' function. Initially, 'eq' is instantiated for the base type 'nat'. Subsequently, the instantiation for lists requires us to

[1] Simplifying label syntax in this way is a component of a possible translation process from $\text{HM}_{\mathcal{R}}$ to $\text{HM}_{\mathcal{O}}$.

$$
\begin{array}{rcl}
e & := & x \\
  & | & \lambda x.e \\
  & | & e\ e \\
  & | & \textbf{let}\ x = e\ \textbf{in}\ e
\end{array}
\qquad
\begin{array}{rcl}
C & := & \top \\
  & | & C \wedge C \\
  & | & \tau = \tau \\
  & | & \exists \alpha.\ C
\end{array}
\qquad
\begin{array}{rcl}
\tau & := & \alpha \\
  & | & \tau \to \tau \\
  \\
\sigma & := & \tau \\
  & | & \forall \vec{\alpha}.C \Rightarrow \tau \\
  \\
\Gamma & := & \emptyset \\
  & | & \Gamma,\ x : \sigma
\end{array}
$$

Fig. 1: Syntax

convey that list equality is feasible only when the elements of the list can be compared, that is, there exists a instance $\alpha \to \alpha \to \text{bool}$.

Instead of focusing exclusively on these individual systems, our exploration will center on HM(X), a HM-based system that remains detached from the actual constraint domain X. Subsequently, we will proceed to instantiate X using the two showcased instances of constraints, namely overloading and polymorphic records.

2 HM(X)

2.1 Introduction

In this segment, our discussion of HM(X) will adopt a slightly informal approach. We will skip over some specifics and formalities to ensure clarity and concentrate on grasping the underlying concepts of the system, rather than deriving a full formal definition.

2.2 Syntax

The syntax of HM(X) is closely related to that of HM.

Expressions e include the constructs found in the simply typed lambda calculus. These encompass variables x, abstractions $\lambda x.\ e$, and applications $e\ e$. Additionally, let bindings are present to confine the language to let polymorphism.

Regarding types, we maintain the distinction between mono types τ and poly types σ. However, unlike in the context of HM, we are now able to introduce constraints C to poly types.

The constraint syntax described here constitutes the *minimal* essential components necessary for the syntax of constraint domain X. The underlying notion is that the constraint syntax is later extended when instantiating HM(X), and conceivably also the syntax for expressions and types, to incorporate those new constraints.

$$\frac{C,\ x : \sigma \in \Gamma}{C,\ \Gamma \vdash x : \sigma}\ \text{(Var)} \qquad \frac{C,\ (\Gamma,\ x : \tau') \vdash e : \tau}{C,\ \Gamma \vdash \lambda x.e : \tau}\ \text{(Abs)}$$

$$\frac{C,\ \Gamma \vdash e_1 : \tau \to \tau' \quad C,\ \Gamma \vdash e_2 : \tau'}{C,\ \Gamma \vdash e_1\ e_2 : \tau'}\ \text{(App)} \qquad \frac{C,\ \Gamma \vdash e : \sigma \quad C,\ (\Gamma,\ x : \sigma) \vdash e' : \tau'}{C,\ \Gamma \vdash \mathbf{let}\ x = e\ \mathbf{in}\ e' : \tau'}\ \text{(Let)}$$

$$\frac{C \wedge D, \Gamma \vdash e : \tau \quad \vec{\alpha} \notin free(C, \Gamma)}{C \wedge \exists\vec{\alpha}.D,\ \Gamma \vdash e : \forall\vec{\alpha}.D \Rightarrow \tau}\ (\forall\text{I}) \qquad \frac{C,\ \Gamma \vdash e : \forall\vec{\alpha}.C' \Rightarrow \tau \quad C \vdash^e [\vec{\tau}/\vec{\alpha}]C'}{C,\ \Gamma \vdash e : [\vec{\tau}/\vec{\alpha}]\tau}\ (\forall\text{E})$$

Fig. 2: Logical Type System ($C, \Gamma \vdash e : \sigma$)

Fig. 3: Syntax

A constraint C is either $\top$ (true), a conjunction of two constraints $C \wedge C$, an equality between mono types $\tau = \tau$, or a projection $\exists\alpha.\ C$. The projection operator introduces a new type variable α to constraint C. Through projection, it becomes possible to express the existence of a type without the necessity of actually introducing a new type variable at the type level. This mechanism proves advantageous within the non-algorithmic typing rules and type inference algorithms. Moreover, it actually extends the expressive capacity beyond that of solely introducing all type variables present in constraints using the $\forall$ operator[2].

2.3 Typing

Our current emphasis will be on the logical type system. Unlike the inference algorithm, which is also provided for HM(X), the logical type system does not represent a determinable process for typing a provided expression. This is due to the ($\forall$E) and ($\forall$I) rules lacking syntax direction and being applicable at various points in the process.

Both (Abs) and (App) are the familiar standards found in the simply typed lambda calculus and operate on mono types, thus enforcing let polymorphism. On the other hand, the (Var) rule allows us to retrieve a variable with a poly type, potentially introduced through a let binding. The (Let) rule likewise incorporates let polymorphism by permitting the expression bound by x to exhibit polymorphism, even though the outcome itself remains monomorphic.

2.4 Type Inference

3 Instantiating HM(X)

3.1 HM($\mathcal{R}$): Instantiation with Polymorphic Records

Extensions

[2] Refer to example 5

Fig. 4: Constraints

Fig. 5: Syntax

Example

3.2 HM($\mathcal{O}$): Instantiation with Overloading

Extensions

Example

4 Metatheory

4.1 Soundness

4.2 Type Inference

5 Related Work & Conclusion

5.1 Related Work

5.2 Conclusion

References

[1] Martin Odersky, Martin Sulzmann, and Martin Wehr. “Type Inference with Constrained Types”. In: *TAPOS* 5 (Jan. 1999), pp. 35–55. DOI: 10.1002/(SICI)1096-9942(199901/03)5:1<35::AID-TAPO4>3.0.CO;2-4.

[2] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004. ISBN: 0262162288.

[3] Christian Skalka and François Pottier. “Syntactic Type Soundness for HM(X)”. In: *Electronic Notes in Theoretical Computer Science* 75 (2003). TIP’02, International Workshop in Types in Programming, pp. 61–74. ISSN: 1571-0661. DOI: https://doi.org/10.1016/S1571-0661(04)80779-5. URL: https://www.sciencedirect.com/science/article/pii/S1571066104807795.

Fig. 6: Constraints