Notes on HM(X)

Martin Müller
Programming Systems Lab, Universität des Saarlandes
mmueller@ps.uni-sb.de

August 3, 1998

Abstract

This note makes available some observations on the HM(X) scheme.

1 Introduction

The HM(X) scheme due to Sulzmann, Odersky, and Wehr [10] allows one to define a type system with ML-style (Hindley/Milner [1, 4]) polymorphism in a generic manner, if a constraint system X is given that suffices to explain the monomorphic fragment of the type system. HM(X) also provides a generic type inference algorithm. The merit of HM(X) is that it provides generic proofs of correctness, principality, and completeness of type inference and so simplifies future design and investigation of type systems.

I am an HM(X) fan. I was always bothered by the feeling that designers of type systems à la ML had been proving the same theorems over and over again. And I share the conviction that the interesting part of ML-style type system is the monomorphic fragment. Now HM(X) is there. Just the right thing. I have, however, some objections against some of the design decisions which I would like to put under consideration. This is what this note is about.

This note contains many claims and no proofs. That means that there may be errors in technical details; but I believe that the main arguments are right and I hope to give them in a convincing manner. I assume familiarity with the HM(X) scheme as presented in [9, 10] and do not define the notions introduced there.

2 Suggestions

In summary, I suggest the following three changes to HM(X).

1. Prefer constraints to type terms wherever possible.
2. In particular, define normal forms not as pairs (φ, C) but simply on constraints.
3. Seperate three issues of the function *normalize*:
   (a) Satisfiability checking as relevant for correctness and principality.

(b) Constraint simplification as relevant for space efficiency.

(c) Term reconstruction as relevant for completeness and for user interaction.

The remainder of this note explains and illustrates these suggestions.

3 Terms versus Constraints

A polymorphic type in the HM(X) scheme is of the form

$$\forall\overline{\alpha}[C].\ \tau \tag{1}$$

where $\overline{\alpha}$ is a sequence of type variables, C is a constraints, and τ is a type term.[1] A type environment Γ in HM(X) is a finite mapping from program variables x to type terms τ.

$$x_1 : \tau_1, \ldots, x_n : \tau_n \tag{2}$$

A judgement in HM(X) is of the form

$$C;\ \Gamma\ \vdash\ e\ :\ \tau \tag{3}$$

saying that program expression e has type τ under the type assumptions in Γ subject to the constraint C. If $C = true$, we drop the constraint and write $\Gamma \vdash e\ :\ \tau$ instead.

Both type terms and constraints *describe* types. For example, the type term $\alpha \to \alpha$ describes some type depending on the (unknown) denotation of α. With constraints, one could express the same information as the solution for α' of an equation $\alpha' = \alpha \to \alpha$. Since HM(X) is based on constraints anyway, one can easily simplify the set-up so that polymorphic types have the form

$$\forall\overline{\alpha}[C].\ \alpha' \tag{4}$$

and that judgements always have the form

$$C;\ \Gamma\ \vdash\ e\ :\ \alpha \tag{5}$$

It is straightforward to transform HM(X) type terms into this new format by introduction of new variables. For example, the type term

$$\alpha \to \alpha \tag{6}$$

(which can equivalently be understood as $\forall\emptyset.\ \alpha \to \alpha$) becomes $\forall\alpha'[\alpha' = \alpha \to \alpha].\ \alpha'$, and the recursive type term

$$\mu\alpha.\alpha \to \alpha' \tag{7}$$

becomes

$$\forall\alpha[\alpha = \alpha \to \alpha'].\ \alpha \tag{8}$$

[1] Instead of $\forall\alpha.C \Rightarrow \tau$ I write $\forall\alpha[C].\ \tau$; to me, this seems more readable when the constraint part C becomes large. I also drop C if it is trivial, *i. e.*, $\forall\overline{\alpha}[true].\ \tau = \forall\overline{\alpha}.\ \tau$ and I drop the quantifier prefix if it binds the empty sequence of variables, *i. e.*, $\forall\emptyset.\ \tau = \tau$.

The elimination rule of HM(X) also simplifies to

$$\frac{C;\, \Gamma \;\vdash\; e \,:\, \forall\overline{\alpha}[D].\,\beta \quad C \vdash^{e} D[\bar{\gamma}/\overline{\alpha}]}{C \wedge D[\bar{\gamma}/\overline{\alpha}];\, \Gamma \;\vdash\; e \,:\, \beta[\bar{\gamma}/\overline{\alpha}]}\ \forall\text{-Elim}$$

I hasten to admit that terms are usually much nicer to read than constraints and so I shall still use *notation* τ for type terms as in HM(X). But I use terms as a abbreviating notation only in place for the corresponding constraints. See also Sections 5.3 and 6.

4 Normalization

The presentation of HM(X) type inference is parametrised by a constraint normalization function called *normalize* [9, 10]. The function *normalize* depends on the notion of a constraint normal form, and it maps a constraint to a normal form if such a normal form exists. Sulzmann [9] presents three strategies of type inference which apply *normalize* at different points in time.

There are two issues: What does normalization do and when does it do it?

4.1 What is normalization?

In HM(X), the function *normalize* has both a semantic and a syntactic purpose.

1. *normalize*(C) checks whether the constraint C is satisfiable (semantics),
2. and returns a normal form of C if C is satisfiable (syntactic).

I think these issues should be separated: On the one hand, just to separate semantic from syntactic arguments. On the other hand, since satisfiability is essential for correctness and principality (the Section 5.1) while normalisation is not (see Sections 5.2 and 5.3).

3. In HM(X), normal forms are not distinguished constraints, but rather pairs (φ, C) where φ is a substitution and C is a constraint. So *normalize* performs two syntactic operations: constraint normalization and term reconstruction.

Again I think that both should be specified separately (see Sections 5.2 and 5.3 in contrast).

4. Satisfiability is assumed decidable.

This is not essential for correctness and principality. It is, of course, needed for an effective type inference algorithm, but this is a different issue.

4.2 When to normalize?

Sulzmann [9] proposes three type inference strategies that he calls “eager”, “lazy”, and “very lazy”.

1. *eager normalization* means to apply *normalize* with every type inference rule,

2. *lazy normalization* means to apply *normalize* only right before type generalization (*i. e.*, the analysis of let), and

3. *very lazy normalization* means to not apply *normalize* but at the final step.

I think that different strategies apply to the different tasks of *normalize*, *i. e.*, and syntactic, semantic, and pragmatic ones.

5 Separating Issues

This section discusses different aspects that are all modelled within the same function *normalize*. As an simple example, consider the polymorphic type

$$\forall \alpha\alpha'\gamma[\alpha = \alpha' \to \gamma \wedge \alpha = \gamma \to \alpha'].\ \alpha \tag{9}$$

where the constraint $\alpha = \alpha' \to \gamma \wedge \alpha = \gamma \to \alpha'$ is indeed satisfiable. Constraint simplification allows one to get rid of the variable γ:

$$\forall \alpha\alpha'[\alpha = \alpha' \to \alpha'].\ \alpha \tag{10}$$

while term reconstruction yields

$$\forall \alpha'.\ \alpha' \to \alpha' \tag{11}$$

5.1 Satisfiability Check

1. The satisfiability check is necessary: *Correctness* of the type system depends on the constraint C in $C;\ \Gamma \vdash e\ :\ \tau$ to be satisfiable.

2. The satisfiability check should be done eagerly in order to *report type errors* as early as possible.

3. Decidability of satisfiability is not relevant for the correctness of type inference, only for its effectiveness. Correctness alone would allow one to have an oracle that answers satisfiability requests.

4. Satisfiability need not be linked to normalization of constraints: Correctness of the type system is independent of the syntactic representation of the constraint C in $C;\ \Gamma \vdash e\ :\ \tau$.

5. The latter remark indicates that a rule

$$\frac{D;\ \Gamma\ \vdash\ e\ :\ \tau \quad C \vdash^e D}{C;\ \Gamma\ \vdash\ e\ :\ \tau}\ \mathit{Equiv}$$

should be allowed as part of the "logical type system" [10] (here, $\vdash^e$ denotes the constraint entailment relation, and the satisfiability assumption on C is left implicit). Sulzmann [9] shows that a corresponding meta-property on derivations holds.

6. Principality of type inference can be guaranteed with the strategy to always use the following instance of the $\forall$-Intro rule:

$$\frac{C;\ \Gamma\ \vdash\ e : \tau \quad \overline{\alpha} \notin \mathrm{fv}(C) \cup \mathrm{fv}(\Gamma)}{\exists\overline{\alpha}C;\ \Gamma\ \vdash\ e : \forall\overline{\alpha}[C].\,\tau}\ \forall\text{-Intro}$$

   The general form of $\forall$-Intro allows one to only generalize over a part of the constraint from the environment.

$$\frac{C \wedge D;\ \Gamma\ \vdash\ e : \tau \quad \overline{\alpha} \notin \mathrm{fv}(C) \cup \mathrm{fv}(\Gamma)}{C \wedge \exists\overline{\alpha}D;\ \Gamma\ \vdash\ e : \forall\overline{\alpha}[D].\,\tau}\ \forall\text{-Intro}$$

   See also remark 2 in Section 5.2 below.

5.2 Constraint Normalization

I use *constraint normalization* for the replacement of one constraint C by another, equivalent constraint D that is called a normal form.

1. Constraint normalization can be done at arbitrary points in time, as far as correctness or principality are concerned.

2. Notice that the "principal constraint property" [10] is trivial if the normal form of a constraint is just some other equivalent constraint (instead of a pair (ϕ, C) as done in HM(X)).

3. Apparently, the generalization strategy mentioned above is undesirable space-wise, since the constraint part of a polymorphic type scheme is copied on every instantiation. Thus, when applying $\forall$-Intro to a judgement $C';\ \Gamma \vdash e\ :\ \tau$, one wants to split C' into C and D such that D is as small as possible. More precisely, one wants

   (a) to find a constraint C'' that is equivalent to C' and contains a minimal number of variables, and then

   (b) to split C'' into $C \wedge D$ such that D contains the minimal number of constraints that constrain the variables in τ (or simply the type variable τ according to the suggestion in Section 3). *Constrainedness* can be defined recursively, as done, for instance in [8] or in [6].

4. For *space-efficiency*, constraint normalization should be done latest before type generalization.

5.3 Term Reconstruction

1. Type terms are easier to read than constraints and thus term reconstruction should definitely take place before the type inference result is reported to the programmer. But it suffices to do this "very lazily", *i. e.*, right before the user interaction.

2. Type terms might be needed "eagerly" if one wants to prove a constraint-based type system *complete* with respect to a type system that is defined by type terms. For example, consider the completeness proof of HM(Herbrand) with respect to the classical HM system in [10]. The flipside of this argument is the fact that type terms are not strictly required at all for newly designed type systems.

3. Type terms are particularly inconvenient in presence of recursive types, since μ-terms as term representation of infinite regular trees are cumbersome. But this is a syntactic issue which can be avoided completely for the discussion of correctness and principality.

5.4 Constraint-based Type Inference

1. (Monomorphic) Type inference for an expression e can be understood as returning a constraint on the variable chosen to represent the types of e. All solutions of this constraint are valid (monomorphic) types for e.

2. For example type inference for $\lambda x.x$ whose type is represented by α will return the constraint $\exists\alpha'(\alpha = \alpha' \to \alpha')$. The solutions of this constraints are all trees $\tau \to \tau$.

3. If the type environment contains free type variables, the constraint returned by type inference will also be principal for these. Notice in passing, that the HM(X) scheme has built-in the focus on *principal typings* instead of *principal types* (put forward by Jim in [2]), since the constraint returned by type inference is part of the environment.

4. The term reconstruction problem is the problem of giving a compact representation of all solutions of a constraint.

5. Equations over finite trees (Herbrand) have the nice property that there is a simple and pleasing term representation of the solutions of a constraint (with a single free variable). Type inference for more complicated type systems commonly uses more complex constraint systems for which the term reconstruction problem becomes a lot harder. For instance, compare type inference for Ohori's polymorphic record calculus [7], or type inference for first-class messages as discussed in [5,6].

The problem of term reconstruction is by no means a non-trivial one: Even better if one can discuss this separately as an operation on constraints that does not at all affect the type system itself.

6 Kennedy's Example

In [3], Kennedy reports on a difficulty with principality in a constraint-based dimension type system. Kennedy considers dimension types (kg, m, s, *etc.*) subject to an equational theory that provides for multiplication and division (*e. g.*, $m \cdot m/s$ is a type and equal to $(m \cdot m \cdot kg)/(s \cdot kg)$).

Here is the example. Assume the binary division function *div* to have the type $\forall \delta_1 \delta_2.\ \delta_1 \cdot \delta_2 \to \delta_1 \to \delta_2$, which says that the dimension of the first argument must be divisible by the dimension of the second argument and the dimension of the result type is given by the quotient of the argument dimensions. For example, $div\ (42\, m \cdot m/s)\ (2\, m)$ returns $21\ m/s$. The question is, what is the most general type of $div\ x$ under the assumption that x has type $\delta_1 \cdot \delta_2$. Apparently, this is a valid judgement:

$$x : \delta_1 \cdot \delta_2, div : \forall \delta_1 \delta_2.\ \delta_1 \cdot \delta_2 \to \delta_1 \to \delta_2 \vdash div\ x\ :\ \delta_1 \to \delta_2 \tag{12}$$

Neither δ_1 not δ_2 can be generalized over since both appear free in the environment in the type of x. However, $\delta_1 \to \delta_2$ is *not* the most general type of $div\ x$. As Kennedy observes, representing $\delta_1 \cdot \delta_2$ as $\delta_1 \cdot \delta_3 \cdot \delta_3^{-1} \cdot \delta_2$ licenses the following judgement:

$$x : \delta_1 \cdot \delta_2, div : \forall \delta_1 \delta_2.\ \delta_1 \cdot \delta_2 \to \delta_1 \to \delta_2 \vdash div\ x\ :\ \forall \delta_3.\ \delta_1 \cdot \delta_3 \to \delta_3^{-1} \cdot \delta_2 \tag{13}$$

Sulzmann already notices [9] that the problem can be used with "very lazy normalization" which postpones term reconstruction until the very end. I agree with this point of view: More strongly, however, it seems to me that there is no problem in the first place if terms are replaced by constraints as suggested in Section 3.

For illustration, let $\Gamma = x : \alpha, div : \forall \delta_1 \delta_2 \alpha[\alpha = \delta_1 \cdot \delta_2].\ \alpha \to \delta_1 \to \delta_2$. Then we have the following derivation.

$$\cfrac{\cfrac{\alpha = \delta_1 \cdot \delta_2;\ \Gamma\ \vdash\ div\ :\ \alpha \to \delta_1 \to \delta_2}{\alpha = \delta_1 \cdot \delta_2;\ \Gamma\ \vdash\ div\ x\ :\ \delta_1 \to \delta_2}}{\Gamma \vdash div\ x\ :\ \forall \delta_1 \delta_2[\alpha = \delta_1 \cdot \delta_2].\ \delta_1 \to \delta_2}\ \forall\text{-Intro} \tag{14}$$

So the polymorphic type of $div\ x$ comes naturally without the need for an additional argument.

By the way: As suggested in Section 3, the derivation above should be understood as an abbreviation for this one where (monomorphic) type terms have been completely moved into the constraint part.

$$\cfrac{\cfrac{\alpha = \delta_1 \cdot \delta_2 \wedge \alpha' = \alpha \to \gamma \wedge \gamma = \delta_1 \to \delta_2;\ \Gamma\ \vdash\ div\ :\ \alpha'}{\alpha = \delta_1 \cdot \delta_2 \wedge \alpha' = \alpha \to \gamma \wedge \gamma = \delta_1 \to \delta_2;\ \Gamma\ \vdash\ div\ x\ :\ \gamma}}{\Gamma \vdash div\ x\ :\ \forall \delta_1 \delta_2 \gamma[\alpha = \delta_1 \cdot \delta_2 \wedge \gamma = \delta_1 \to \delta_2].\ \gamma}\ \forall\text{-Intro} \tag{15}$$

7 Assessment

We have suggested some simplifications of HM(X), most of which were driven by the desire to take constraints even more seriously. I think that these simplifications are in line with the driving motivations of HM(X) which they appear to apply more radically.

The main suggestion is to prefer constraints over terms whenever possible. Preferring constraints over terms has an impact on the definition of polymorphic types and judgements (Section 3) and it avoids the principality problem of eager term reconstruction (Section 6).

While there is no principal problem with using term languages, these tend to be less flexible as a constraint language. Furthermore, a more general notion of constraints is available in HM(X) anyway, so why introduce an additional concept too early? Specifically, since an additional term concept is not necessary for the specification of a sound type system and principal type inference!

Our suggestions outline the following strategy for the design of new type systems.

1. Cook up a domain of types.
2. Specify a system of constraints over this domain, which can express all properties of a program that are needed to decide its well-typedness.
3. Specify the type system in HM(X) style.
4. Only last, design a term language as a representation language for the solution of constraints.

Acknowledgements

The motivation to write this note came up during the writing of [5] with Susumu Nishimura and an e-mail conversation with Martin Sulzmann on HM(X). I would to thank both for invaluable feedback, and also acknowledge helpful comments on HM(X) by Martin Odersky.

References

[1] R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, Dec. 1969.

[2] T. Jim. What are principal typings and what are they good for? In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 32–53, St. Petersburg Beach, Florida, Jan. 1996. ACM Press, New York.

[3] A. Kennedy. Type inference and equational theories. Technical Report LIX/RR/96/09, LIX, Ecole Polytechnique, 91128 Palaiseau Cedex, France, Sept. 1996.

[4] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17(3):348–375, 1978.

[5] M. Müller and S. Nishimura. Type inference for first-class messages with feature constraints. 1998. submitted.

[6] S. Nishimura. Static typing for dynamic messages. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pages 266–278. ACM Press, New York, 1998.

[7] A. Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, 1995.

[8] G. Smolka and R. Treinen. Records for logic programming. *The Journal of Logic Programming*, 18(3):229–258, Apr. 1994.

[9] M. Sulzmann. Proofs of properties about HM(X). Technical Report YALEU/DCS/RR-1102, Yale University, 1998.

[10] M. Sulzmann, M. Odersky, and M. Wehr. Type inference with constrained types (extended abstract). In B. Pierce, editor, *Proceedings of the 4th International Workshop on Foundations of Object-oriented Programming*, Jan. 1997. Full version to appear in TAPOS, 1998.