Hindley/Milner style type systems in constraint form

Martin Sulzmann — Yale University *
Martin Müller — Universität des Saarlandes †
Christoph Zenger — University of South Australia ‡

Abstract

This papers describes Hindley/Milner style type systems in constraint form. Previously, many type descriptions have still been partially represented in the type language. We argue that this can lead to shortcomings in the type and inference system. Examples are given to support this statement. As a solution we present a purely constraint–based formulation of Hindley/Milner style type systems which enjoys much nicer logical properties than previous approaches. More specifically, we present a solution to the open problem of type inference in non-regular theories.
Keywords: constraints, term types, equational theories, type inference.

1 Introduction

The area of Hindley/Milner [Mil78] style type systems has been extensively studied in the last couple of years. We only mention a selected number of papers which extend the expressiveness of the Hindley/Milner type system, such as record systems [Oho95, Rém89, Wan89], overloading [Kae92, VHJW96, CHO92, OWW95, NP95] and subtyping [AW93, EST95b, Smi91]. Variations of the Hindley/Milner system are also popular to perform certain kind of program analysis, such as [Ken94, DHM95, TJ92]. More general frameworks have been introduced by Jones [Jon92] and Odersky/Sulzmann/Wehr [OSW99].

In our own work we introduced a general framework for Hindley/Milner style systems, called HM(X), which is parameterized in the constraint domain X. For instance, the standard Hindley/Milner type system can be retrieved by instantiating X to the standard Herbrand constraint system. One of the main improvements was introducing a clean semantic notion of the concept of constraints. In previous work, constraints were mostly considered purely syntactic, e.g. as a set of formulas often of a specific form. In contrast, we gave a precise semantic characterization of constraints along the lines of Henkin [HMT71] and Saraswat [Sar93]. HM(X) can be seen analogous to the CLP(X) framework in constraint logic programming [JM94].

But still this is not satisfactory yet. Type descriptions are partially represented in the language of types. There are cases where this can lead to serious shortcomings, see Section 2 for some specific examples. In order to fix such problems, we propose to go the whole way to a purely constraint–based formulation of Hindley/Milner style types systems. All type descriptions are then represented in constraint form. Type descriptions usually consist of a term algebra which describes the valid type terms. Commonly, type descriptions respectively type terms are referred to as types. As an example we consider a variation of the HM system which deals with recursive types. We find the following judgment:

$$\{x : \mu\alpha.\alpha \rightarrow \beta\} \vdash xx : \beta \tag{1}$$

Term x is applied to itself, therefore it needs to be of recursive type. Recursive types are represented by the μ notation in the syntactic domain of types.

Equivalently, types could also be represented in the constraint language. Then the above judgment could also be represented as follows:

$$(\alpha = \alpha \rightarrow \beta), \{x : \alpha\} \vdash xx : \beta \tag{2}$$

where $(\cdot = \cdot)$ is an equality constraint between types. Then the constraint $(\alpha = \alpha \rightarrow \beta)$ contains all essential typing information. This illustrates the basic idea of our approach. Usually, types are represented in the syntactic domain of types which is described by a term algebra. In contrast, we view a type τ as the type variable α under the constraint $(\alpha = \tau)$.

Note the difference. In the former view, the type τ appears in term form in the type language, whereas in the latter view, τ appears only in the constraint language. Quite often we refer to the former representation of types as the *term* representation respectively term form, and to the latter we refer to as the *constraint* representation respectively constraint form.

Here, we introduce a purely constraint–based approach. Typing judgments are now of the form $C, \Gamma \vdash e : \sigma$ where C is a constraint and type scheme σ is formed as follows:

$$\begin{array}{rcl} \tau & ::= & \alpha \mid \tau \rightarrow \tau \\ \sigma & ::= & \tau \mid \forall \alpha.C \Rightarrow \sigma \end{array}$$

In the type language we only admit type variables and function types. Sometimes will be sloppy and still allow other types to appear in the type language. But this should only be seen as a convenient notation. A type τ can always be viewed as the type variable α under the constraint $(\alpha = \tau)$. Bounded variables in type schemes are further constrained by a constraint component. We require that constraints appearing on the left–hand side of the turnstile and

*Department of Computer Science, New Haven, CT 06520-8285, sulzmann@cs.yale.edu
†Programming Systems Lab, D–66041 Saarbrücken, Germany, mmueller@ps.uni-sb.de
‡School of Computer and Information Science, Adelaide, SA 5095, Australia, ciscz@cs.unisa.edu.au

in type schemes are satisfiable. This differs from the original HM(X) system where we could possibly further restrict the set of constraints appearing in typing judgments and type schemes. Then typing information is partially represented in the constraint language and the syntactic domain of types. To such systems we refer to as term–based HM systems.

Contribution This works points out shortcomings in the type and inference system which are apparent in term–based HM approaches. This observation was first made by Andrew Kennedy who gave the initial motivation for this work. We observe that representing typing information partially in the syntactic domain of types not only affects the expressiveness of the type system, but also can lead to an incomplete inference system, even in the presence of most general unifiers. As a solution we introduce a purely constraint–based formulation of the HM(X) system. Nevertheless, in some cases the term representation of judgments is important, e.g. for user interaction. The user might prefer a specific syntactic type representation instead of a representation in constraint form. This is especially important when inferring types and reporting the result to the user. To this purpose we furthermore present a revised substitution–based inference system which overcomes problems which are apparent in the standard formulation.

Outline Section 2 gives some motivating examples which exhibit the problems we are intending to solve. In Section 3 we present the revised HM(X) framework. Section 4 elaborates more on the differences between viewing systems in term respectively constraint form and establishes connections between both. Section 5 concludes.

2 Overview

We start with two motivating examples which are especially interesting in the context of program analysis. We consider dimension type analysis in style of [Ken94]. The index analysis is similar to the work by [Xi98, Zen98]. Both examples are slight variations of each other. The first examples points out shortcomings in the inference system whereas the second examples reveals shortcomings in the type system when representing typing information in term form.

Example 1 We cite the following example given by Andrew Kennedy [Ken96]. He introduces a Hindley/Milner style system that deals with physical dimensions. In addition to the usual types there are dimension types. The term algebra of dimension types is defined as follows:

$$\begin{array}{lll} \textbf{Dimensions} & d & ::\supseteq \; \alpha \mid i(d) \mid prod(d,d) \mid 1 \mid \mathrm{M} \mid \mathrm{T} \\ \textbf{Types} & \tau & ::= \; \alpha \mid \dim(d) \mid \tau \to \tau \end{array}$$

The dimension constructor $i(\cdot)$ corresponds to the inverse of a dimension and $prod(\cdot,\cdot)$ to the product of two dimensions. Dimension constants are 1 for the unit measure, M for the mass dimension and T for the time dimension. For specific applications there might be further dimension constants.

Consider the following initial type environment:

$$\begin{array}{rcrcl} \Gamma & = & \{kg & : & \dim \mathrm{M}, \\ & & s & : & \dim \mathrm{T}, \\ & & div & : & \forall d_1, d_2 . \dim prod(d_1, d_2) \to \\ & & & & \qquad \dim d_1 \to \dim d_2, \\ & & pair & : & \forall t_1, t_2 . t_1 \to t_2 \to (t_1, t_2)\} \end{array}$$

Here, kg and s are some basic dimensions, $pair$ is the pairing operator and div is a primitive operation on dimensions. Consider the following expression:

$$e \quad = \quad \lambda x.\mathsf{let}\ y = div\, x\ \mathsf{in}\ pair(y\ kg)(y\ s)$$

We want to type e under the type environment Γ. The subexpression $div\, x$ has the following type under type environment $\Gamma.x : \dim prod(d_1, d_2)$:

$$\Gamma.x : \dim prod(d_1, d_2) \vdash div\, x : \dim d_1 \to \dim d_2 \tag{3}$$

In the body of the let–statement we apply y to arguments of different types. Hence, function y must be of polymorphic type. But it is not possible to quantify over the type variables d_1 and d_2 because d_1 and d_2 also appear in the type environment. It seems that program e is not typable in the dimension type system. But we can derive another type for $div\, x$ under the same type environment:

$$\begin{array}{c} \Gamma.x : \dim prod(d_1, d_2) \\ \vdash \\ div\, x : \dim prod(d_1, d_3) \to \dim prod(i(d_3), d_2) \end{array} \tag{4}$$

We have simply instantiated d_1 with $prod(d_1, d_3)$ and d_2 with $prod(i(d_3), d_2)$. Dimension types obey the laws of an abelian group. For details we refer to [Ken96]. For instance, it holds that

$$\vdash^e \; (prod(prod(d_1, d_3), prod(i(d_3), d_2)) = prod(d_1, d_2))$$

where $\vdash^e$ is the entailment relation between constraints and $(\cdot = \cdot)$ is a primitive predicate which expresses the equality relation between types. Hence, the type environment remains the same. But now it is possible to quantify over the free type variable d_3 in typing judgment (4). Then program e becomes typable. We find that

$$\begin{array}{c} \Gamma \vdash e : \\ \forall d_1, d_2 . \dim prod(d_1, d_2) \to \\ (\dim prod(i(\mathrm{M}), prod(d_1, d_2)), \\ \dim prod(i(\mathrm{T}), prod(d_1, d_2))) \end{array} \tag{5}$$

The theory of dimension types possesses most general unifiers. There is a decidable algorithm that computes the most general unifier, see [LBB84]. Hence, we would expect that type inference yields principal types. Unfortunately, it turns out that the standard type inference algorithm fails to infer principal types. In case of dimension types we are faced with the problem of having several different representation types for a given term under the same type environment, see judgments (3) and (4). If we choose a wrong representation it might not be possible to type the rest of the program. In the above example, we first had to apply a substitution to reveal the free type variable d_3. It it crucial that this substitution leaves the type environment unchanged.

Consider now a variation of the dimension type system where dimension types are allowed to appear in constraint form. Then, we find the following variation of typing judgment (4).

$$(\alpha = prod(d_1, d_2)), \Gamma.x : \alpha \vdash div\, x : \dim d_1 \to \dim d_2$$

The left–hand side of typing judgments is extended with a constraint component. The type of the function $div\, x$ is now not only represented by its type but additionally by the constraint $(\alpha = \mathit{prod}(d_1, d_2))$. If we would unify the constraint $(\alpha = \mathit{prod}(d_1, d_2))$ we would obtain judgment (4) again. But we keep constraints in explicit form and then type variables d_1 and d_2 do not appear anymore in the type environment. Hence, we can quantify over these type variables. The function y has now the desired polymorphic type

$$\forall d_1, d_2.(\alpha = \mathit{prod}(d_1, d_2)) \Rightarrow \dim d_1 \to \dim d_2$$

The following typing judgment is a variation of judgment (5).

$$\begin{array}{c}(\alpha = \mathit{prod}(d_1', d_2')) \wedge (d_1' = \mathrm{M}) \wedge (d_2' = \beta) \wedge \\ (\alpha = \mathit{prod}(d_1'', d_2'')) \wedge (d_1'' = \mathrm{T}) \wedge (d_2'' = \gamma), \\ \Gamma \vdash e : \dim \alpha \to (\dim \beta, \dim \gamma)\end{array}$$

For instance, the constraint

$$(\alpha = \mathit{prod}(d_1', d_2')) \wedge (d_1' = \mathrm{M}) \wedge (d_2' = \beta)$$

appears by applying the term y to kg.

Then term e has type $\dim \alpha \to (\dim \beta, \dim \gamma)$ under the constraint

$$\begin{array}{c}(\alpha = \mathit{prod}(d_1', d_2')) \wedge (d_1' = \mathrm{M}) \wedge (d_2' = \beta) \wedge \\ (\alpha = \mathit{prod}(d_1'', d_2'')) \wedge (d_1'' = \mathrm{T}) \wedge (d_2'' = \gamma)\end{array}$$

This representation of e's type might be more difficult to interpret compared to the term representation. But by switching to a representation in constraint form we omit the previously mentioned problem where the standard inference algorithm could not infer e's type. It turns out that it is even possible to infer principal types in term form. In Section 4.2 we present an adapted version of the standard inference algorithm which always infers principal types, under the assumption that unification is unitary.

Example 2 The next example shows that representing systems in term or constraint form affects the expressiveness of the type system. We consider a slight variation of the last example. We have list data types which are parameterized in the length of the list. Compared to other systems, the list constructor List takes as an additional parameter the length of the list. For simplicity, we assume that all list elements are of the same type, e.g. Int. Then, we write $\mathsf{List}\, i$ to denote an integer list of length i. This is quite similar to the dimension type system. Instead of dimensions we have now indices.

$$\begin{array}{llll}\textbf{Indices} & i & ::= & \alpha \mid +(i,i) \mid 0 \mid 1 \\ \textbf{Types} & \tau & ::= & \alpha \mid \mathsf{List}\, i \mid \tau \to \tau\end{array}$$

The constant constructor 0 represents the zero, and 1 represents the one index. The constructor $+(\cdot,\cdot)$ allows to add indices and obeys the expected algebraic laws. For instance, it holds that

$$\vdash^e (+(i_1, i_2) = +(i_2, i_1))$$

which states that $+(\cdot,\cdot)$ is commutative. Furthermore, we abbreviate $+(1,1)$ by 2 and so forth.

Consider now the following initial type environment:

$$\begin{array}{rcrcl}\Gamma & = & \{\mathsf{l}_1 & : & \mathsf{List}\ 1, \\ & & \mathsf{l}_2 & : & \mathsf{List}\ 2, \\ & & \mathit{remove} & : & \forall n, m.\mathsf{List}\ +(n,m) \to \mathsf{List}\ n \to \mathsf{List}\ m, \\ & & \mathit{pair} & : & \forall t_1, t_2.t_1 \to t_2 \to (t_1, t_2)\}\end{array}$$

where l_1 and l_2 are some arbitrary lists of length one and two, *remove* is a primitive operations which takes two lists and removes as many elements from the first list as the second list contains and *pair* is the pairing operator. Consider the following expression:

$$e \quad = \quad \lambda x.\mathsf{let}\ y = \mathit{remove}\, x\ \mathsf{in}\ \mathit{pair}(y\, \mathsf{l}_1)(y\, \mathsf{l}_2)$$

The following reasoning is quite similar to the dimension type example. Again, it turns out that we can not give function y a polymorphic type in a system which represents types in term form. In an intermediate step, we find the following judgment:

$$\Gamma.x : \mathsf{List}\ +(n,m) \vdash \mathit{remove}\, x : \mathsf{List}\ n \to \mathsf{List}\ m \qquad (6)$$

Indices n and m appear on the left–hand side of the turnstile which prohibits to quantify over them. Let us switch to a system where all types are represented in constraint form. Then e's type is as follows:

$$\begin{array}{c}(\alpha = +(n', m')) \wedge (n' = 1) \wedge (m' = \beta) \wedge \\ (\alpha = +(n'', m'')) \wedge (n'' = 2) \wedge (m'' = \gamma), \\ \Gamma \vdash e : \mathsf{List}\ \alpha \to (\mathsf{List}\ \beta, \mathsf{List}\ \gamma)\end{array}$$

Now it becomes apparent why expression e is not typable in term form. The lambda–variable x with type α has to satisfy the constraint

$$\begin{array}{c}(\alpha = +(n', m')) \wedge (n' = 1) \wedge (m' = \beta) \wedge \\ (\alpha = +(n'', m'')) \wedge (n'' = 2) \wedge (m'' = \gamma)\end{array}$$

But there is no term representation for α covering all solutions which are apparent in the constraint representation. Therefore, we can not give function y a polymorphic type, hence e is not typable in a system where types are represented in term form. The questions arises under which conditions both views, term resp constraint form, are equivalent. The careful reader might already suspect that we need something like unitary unification to establish a one–to–one correspondence between both representation forms. We elaborate on this more in Section 4.1.

Summary We observe that representing types in term form can lead to some serious shortcomings. The above examples support this statement. Kennedy's examples shows that the inference system in algorithmic form is incomplete even in the presence of most general unifiers. Mgu's are represented as substitutions which are of syntactic nature. But the syntactic representation of the solutions of typing problems leads to an incomplete inference system as Kennedy's examples shows. The second example even shows that the term representation of types leads to a less expressive system.

The essential problem lies in the term representation of types which is based on the concept of substitutions [1]. Substitutions are harmless in the constraint language because we find the following identity:

$$[\tau/\alpha]C =^e \exists \alpha.((\alpha = \tau) \wedge C)$$

But problems arise when representing types as terms as illustrated by the above examples.

[1] Note, the type τ can be seen as applying substitution $[\tau/\alpha]$ to α where α is fresh type variable.

Both problems occurred in extensions of the HM system with some sort of equational theory. We believe that such variations of the HM system are quite interesting. Equational theories can describe fairly complex type systems respectively program analyses.

We already saw dimension and index analysis, another example is effect analysis including the special case of region and exception analysis.

To overcome the above mentioned problems we only represent types in the constraint language and omit a term representation of types. This leads to a revised version of the HM(X) system (see the next section) which stresses much more the semantic than the syntactic properties of types. Furthermore, we find a much cleaner connection between the logical and inference system. In Section 4 we elaborate more on the impact on the type/inference system when representing type in term respectively constraint form. We give sufficient conditions under which both representation forms are equivalent. Furthermore, we present an adapted version of the inference system which solves the problem pointed out in Example 1.

3 The HM(X) framework

In this section we present the revised HM(X) framework. We only highlight the main differences to the original formulation which can be found in [OSW99]. In order to avoid confusion we make it explicit when we refer to the original formulation.

We start the presentation of the HM(X) framework with the logical system and present then the inference system where we also state connections between both.

3.1 Logical system

This section presents the logical part of the HM(X) framework. We work with the following syntactic domains.

Values	v	$::=$	$x \mid \lambda x.e$
Expressions	e	$::=$	$v \mid e\,e \mid \mathsf{let}\ x = e \mathsf{\ in\ } e$
Types	τ	$::=$	$\alpha \mid \tau \to \tau$
Type schemes	σ	$::=$	$\tau \mid \forall\alpha.C \Rightarrow \sigma$

The set of values and expressions remains unchanged whereas in the syntactic domain of types we only admit type variables and function types. All other terms are represented in the constraint language. The syntactic domain of constraints is as follows:

$$C \ ::= \ C \wedge C \mid \exists\bar{\alpha}.C \mid P$$

where P stands for arbitrary primitive constraints. We assume that we have at least $(\cdot = \cdot)$ as a primitive constraint where the $(\cdot = \cdot)$ predicate satisfies the laws of a congruence. Furthermore, we require that constraints on the left-hand side of typing judgments and in type schemes must be satisfiable. A constraint C is *satisfiable*, if $\vdash^e \exists fv(C).C$. This is the only requirement which we put on constraints which differs to the original presentation. There, we could possibly further restrict the set of constraints which appear in typing judgments and type schemes. For instance, in order to model the Hindley/Milner system we only allowed the empty constraint $\{\}$ representing true. This was motivated by the term representation of types in the Hindley/Milner system. Here, we prefer a constraint presentation of types. This change implies the occurrence of explicit equality constraints in the revised HM(X) system which were represented as substitutions in the original HM(X) presentation.

The typing rules can be found in Figure 1. We assume that type schemes are equivalent up to renaming of bound type variables. Technically, we consider a type scheme $\forall\bar{\alpha}.C \Rightarrow \tau$ as an equivalence class where the equivalence relation $\sim$ is defined as follows: $(\forall\bar{\alpha}.C_1 \Rightarrow \tau_1) \sim (\forall\bar{\beta}.C_2 \Rightarrow \tau_2)$ if $[\bar{\gamma}/\bar{\alpha}]C_1 = [\bar{\gamma}/\bar{\beta}]C_2$ and $[\bar{\gamma}/\bar{\alpha}]\tau_1 = [\bar{\gamma}/\bar{\beta}]\tau_2$ where $\bar{\gamma}$ are fresh type variables and $=$ stands for syntactic equality.

If not otherwise stated, most of the properties carry over from the original HM(X) system to the revised system. For instance, judgments are still closed under strengthening the constraint.

Lemma 1 *Given a judgment $C, \Gamma \vdash e : \sigma$ and a satisfiable constraint D such that $D \vdash^e C$. Then $D, \Gamma \vdash e : \sigma$.*

We give a brief description of the typing rules and point out the changes compared to the original formulation. Rules (Var) to ($\forall$ Intro) remain unchanged whereas we reformulated rule ($\forall$ Elim) and introduced the new rule ($\exists$ Intro).

Rule (Sub) is a general subsumption rules which allows us to derive term e with type τ' if we can type term e with type τ and type τ subsumes type τ'. The subsumption relation $(\cdot \preceq \cdot)$ is defined by the constraint system X and satisfies the standard axioms for a partial ordering plus the contra–variance rule. Usually, subsumption $\preceq$ corresponds to the equality relation $(\cdot = \cdot)$ or the subtype relation $(\cdot <: \cdot)$ between types. For more details, we refer to [OSW99].

Rule ($\forall$ Intro) is one of the novelties of the HM(X) system. Bound type variables become not only quantified in type schemes but also in the constraint part of typing judgments. The projection operator $\exists\bar{\alpha}$ allows to bind type variables in the constraint and corresponds to existential quantification if the constraint system satisfies the laws of a boolean algebra. This new formulation achieves quite some substantial improvements compared to previous formulations. We refer to [Sul98] for a detailed discussion.

Rule ($\forall$ Elim) differs compared to the original formulation which was previously as follows:

$$\frac{C, \Gamma \vdash e : \forall\bar{\alpha}.D \Rightarrow \tau' \qquad C \vdash^e [\bar{\tau}/\bar{\alpha}]D}{C, \Gamma \vdash e : [\bar{\tau}/\bar{\alpha}]\tau'}$$

This rule exhibits the influence of the substitution-motivated approach. Type instances are generated by substituting bound type variables. Here, we give up the concept of substitutions (besides variable renaming). Satisfiability of the constraint part of the constraint is ensured via the side condition $C \vdash^e \exists\alpha.D$.

Rule ($\exists$ Intro) is motivated by the following logical rule: $\forall\alpha.(P \to Q) = (\exists\alpha.P) \to Q$ where $\alpha \notin fv(Q)$. The outermost forall quantification of the free type variable α in the judgment $C, \Gamma \vdash e : \sigma$ can be moved to the constraint part, if α does only appear in the constraint part. This allows to hide type variables only appearing in the constraint part of typing judgments.

Rules (Abs) and (App) require the presence of function types in the syntactic domain of types. It would be possible to give a formulation of the typing rules where only type variables appear in the syntactic domain of types. But to us the current formulation looks more pleasing, of course we need to admit function types in the domain of types. There is no difference in the logical properties of the two

$$\text{(Var)} \qquad C, \Gamma \vdash x : \sigma \quad (x : \sigma \in \Gamma)$$

$$\text{(Sub)} \qquad \frac{C, \Gamma \vdash e : \tau \quad C \vdash^e (\tau \preceq \tau')}{C, \Gamma \vdash e : \tau'}$$

$$\text{(Abs)} \qquad \frac{C, \Gamma_x . x : \tau \vdash e : \tau'}{C, \Gamma_x \vdash \lambda x.e : \tau \to \tau'}$$

$$\text{(App)} \qquad \frac{C, \Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad C, \Gamma \vdash e_2 : \tau_1}{C, \Gamma \vdash e_1 e_2 : \tau_2}$$

$$\text{(Let)} \qquad \frac{C, \Gamma_x \vdash e : \sigma \qquad C, \Gamma_x . x : \sigma \vdash e' : \tau'}{C, \Gamma_x \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' : \tau'}$$

$$(\forall\ \text{Intro}) \qquad \frac{C \wedge D, \Gamma \vdash e : \tau \qquad \bar{\alpha} \notin fv(C) \cup fv(\Gamma)}{C \wedge \exists \bar{\alpha}.D, \Gamma \vdash e : \forall \bar{\alpha}.D \Rightarrow \tau}$$

$$(\forall\ \text{Elim}) \qquad \frac{C, \Gamma \vdash e : \forall \bar{\alpha}.D \Rightarrow \tau' \qquad C \vdash^e \exists \bar{\alpha}.D}{C \wedge D, \Gamma \vdash e : \tau'}$$

$$(\exists\ \text{Intro}) \qquad \frac{C, \Gamma \vdash e : \sigma \qquad \bar{\alpha} \notin fv(\Gamma) \cup fv(\sigma)}{\exists \bar{\alpha}.C, \Gamma \vdash e : \sigma}$$

Figure 1: Logical type system

formulations, it is only a matter of taste whether allowing additionally function types in the domain of types or not. The following lemma shows that we can always put typing judgments in canonical form.

Lemma 2 *Given a constraint C, a type environment Γ, a term e and a type τ. Then $C, \Gamma \vdash e : \tau$ iff $C \wedge (\alpha = \tau), \Gamma \vdash e : \alpha$ where α is a fresh type variable.*

In [OSW99] we showed that type system instances of the HM(X) system are sound, provided the underlying constraint system X is sound. A constraint system is *sound* if for all constraints C, if $\vdash^e \exists \bar{\alpha}.C$ then there is a monotype vector $\bar{\mu}$ such that $\vdash^e \exists \bar{\alpha}.((\bar{\alpha} = \bar{\mu}) \wedge C)$ [2]. A *monotype* is a type μ such that $fv(\mu) = \emptyset$.

Hence, to achieve type soundness we need to be able to represent all typing information in term form. For further details we refer to [OSW99].

3.2 Inference system

The inference system, see Figure 2, is formulated as a deduction system with clauses of the form $C, \Gamma \vdash^W e : \alpha$ where Γ and e are input values and C and α are output values.

To achieve complete inference we only need a satisfiability check. This check can be performed at any time because all inference rules preserve satisfiability. Remember, in the original HM(X) system we could put further restrictions on the set of constraints appearing in typing judgments. There, inference could also involve generation of substitutions to achieve certain term representations. In contrast to the previous substitution–based formulation of the inference system. we can give a purely constraint–based formulation of the inference system. All typing information is represented in constraint form. Therefore, we only need to keep type variables on the right–hand side of $\vdash^W$. Furthermore, unnecessary type variables are hidden by the projection operator (this corresponds to the ($\exists$ Intro) rule in the logical system). In a substitution–based system we would have used variable renaming which are usually quite cumbersome. Here, the projection operator does the job. We find the invariant given $C, \Gamma \vdash^W e : \alpha$ then $fv(C) \subseteq fv(\Gamma, \alpha)$ and $\alpha \notin fv(\Gamma)$. This simplifies generalization in case of the (Let) rule. We immediately can build the type scheme $\forall \alpha.C \Rightarrow \alpha$. However, depending on the actual constraint system, there might be a way to split constraint C_1 into two parts D_1, D_2 such that $C_1 =^e D_1 \wedge D_2$ and $\alpha \notin fv(D_2)$. Then, only D_1 needs to be moved into the type scheme. But whether or not such a splitting is possible depends on the specific constraint system.

Rule (Equ) allows equivalent transformations of constraints in case one prefers certain canonical/minimal representation of constraints. This rule can be applied at any time, e.g. after each inference step or only at the end. We note that constraint simplification is especially important in case of the (Let) rule. The careful reader might notice that keeping types in constraint form might lead to a blow–up. Assume we have the let–bound variable x with type $\forall \bar{\alpha}.C \Rightarrow \tau$. Applying x to different monomorphic arguments results [3] into the two constraints $\exists \bar{\alpha}.((\beta = \tau) \wedge C)$ and $\exists \bar{\alpha}.((\beta' = \tau) \wedge C)$. Fortunately both constraints are semanti-

[2] We use $(\bar{\alpha} = \bar{\mu})$ as an abbreviation for $(\alpha_1 = \mu_1) \wedge \cdots \wedge \cdots (\alpha_n = \mu_n)$.

[3] Before we apply x we generate a generic instance of x, see the (Var) rule.

$$\text{(Var)} \quad \frac{x : (\forall \bar{\alpha}.D \Rightarrow \tau) \in \Gamma \qquad \beta \text{ new}}{\exists \bar{\alpha}.((\beta = \tau) \wedge D), \Gamma \vdash^W x : \beta}$$

$$\text{(Abs)} \quad \frac{C, \Gamma_x.x : \alpha \vdash^W e : \gamma \qquad \beta \text{ new}}{\exists \alpha, \gamma.((\beta = \alpha \rightarrow \gamma) \wedge C), \Gamma_x \vdash^W \lambda x.e : \beta}$$

$$\text{(App)} \quad \frac{C_1, \Gamma \vdash^W e_1 : \alpha_1 \qquad C_2, \Gamma \vdash^W e_2 : \alpha_2 \qquad \alpha \text{ new}}{\exists \alpha_1, \alpha_2.(C_1 \wedge C_2 \wedge (\alpha_1 \preceq \alpha_2 \rightarrow \alpha)), \Gamma \vdash^W e_1 e_2 : \alpha}$$

$$\text{(Let)} \quad \frac{C_1, \Gamma_x \vdash^W e : \alpha \qquad C_3, \Gamma_x.x : \forall \alpha.C_1 \Rightarrow \alpha \vdash^W e' : \beta}{(\exists \alpha.C_1) \wedge C_3, \Gamma_x \vdash^W \mathsf{let}\ x = e\ \mathsf{in}\ e' : \beta}$$

$$\text{(Equ)} \quad \frac{C, \Gamma \vdash^W e : \alpha \qquad C =^e D}{D, \Gamma \vdash^W e : \alpha}$$

Figure 2: Type inference

cally equivalent to the constraint $\exists \bar{\alpha}.((\beta = \tau) \wedge (\beta' = \tau) \wedge C)$. Hence, the constraint C does not get duplicated. Of course, there is a blow–up of subsumption constraints which leads to exponential complexity of the inference problem. This topic is well–studied for the special case of the standard Hindley/Milner system, see [KMM91]. Constraint simplification is also important before generating type schemes. We believe it is rather difficult to give general simplification strategies. Such strategies depend on each specific constraint system and we postpone this topic to future work. We only mention [Pot98, EST95a] where the interested reader can find simplification strategies in the special case of subtype constraints.

We can state the following results.

Theorem 1 (Soundness of Inference) *Given a term e and a type environment Γ. If $C, \Gamma \vdash^W e : \alpha$ then $C, \Gamma \vdash e : \alpha$.*

For the completeness result we only consider those typing judgments $C, \Gamma \vdash e : \sigma$ where the type environment and the constraint on the left hand side of the turnstile are realizable, i.e. have a type instance. A type environment Γ is *realizable* in a constraint C if for every $x : \sigma \in \Gamma$ there is a τ such that $C \vdash^i \sigma \preceq \tau$ [4].

Theorem 2 (Completeness of Inference) *Given a typing judgment $C, \Gamma \vdash e : \sigma$ such that Γ is realizable in C. Then $C', \Gamma \vdash^W e : \alpha$ such that*

$$C \vdash^e \exists \alpha.C'$$
$$C \vdash^i \forall \alpha.C' \Rightarrow \alpha \preceq \sigma$$

We need the realizability condition to state the property about the constraint. Otherwise, we could not state that $C \vdash^e \exists \alpha.C'$.

The completeness result could be stated in a much more concise way than in the original HM(X) system. To achieve principal types we only need satisfiability. A decidable satisfiability test results into decidable type inference.

[4] The relation $\vdash^i$ allows to compare types wrt to a constraint. The formulation is standard, we refer to [OSW99] for details.

Corollary 1 (Principal Types) *Let $\mathsf{true}, \Gamma \vdash e : \sigma$ be a closed typing judgment such that Γ is realizable in true. Then $C, \Gamma \vdash^W e : \alpha$ such that*

$$\vdash^e \exists \alpha.C$$
$$\vdash^i \forall \alpha.C \Rightarrow \alpha \preceq \sigma$$

4 Terms vs constraints

In this section we elaborate more on the differences of HM systems in term vs constraint form. Specifically, we consider differences in the type system of the original and the revised HM(X) framework. The original HM(X) framework can be seen as a general framework for term–based HM systems. Furthermore, we present an adapted version of the substitution–based inference system of the original HM(X) system which overcomes the problem revealed by Kennedy's example. Other approaches for term–based inference are discussed in the last section.

4.1 Type system

First, we make some observations regarding the two new rules ($\forall$ Elim) and ($\exists$ Intro) found in the HM(X) framework. Rule ($\forall$ Elim) is already present in the original HM(X) system but formulated in a different way. There the formulation reflects the term–based approach whereas the new formulation reflects the constraint–based approach. But it turns out that both formulations are essentially equivalent. The following states that we could also use the constraint–based formulation of the ($\forall$ Elim) rule in the original HM(X) system.

Lemma 3 *Given an instance of the original HM(X) system. If the judgment $C, \Gamma \vdash e : \sigma$ is derivable using the constraint–based ($\forall$ Elim) rule then $C, \Gamma \vdash e : \sigma$ is derivable using the substitution–based formulation of the ($\forall$ Elim) rule instead. The reverse direction holds too.*

Rule ($\exists$ Intro) is not present in the original HM(X) system. This rule does not add any additional expressiveness to the revised HM(X) system. Merely, it allows to hide type variables which are only present in the constraint part of typing judgments. Given $C, \Gamma \vdash e : \sigma$ such that $\alpha \notin fv(\Gamma, \sigma)$. Then we find $\exists\alpha.C, \Gamma \vdash e : \sigma$. In the original system, we could have applied a renaming substitution which achieves a similar effect, yielding $[\beta/\alpha]C, \Gamma \vdash e : \sigma$ where β is new. But rule ($\exists$ Intro) prevents from introducing new type variables, type variables can immediately be hidden by the projection operator $\exists\alpha$.

We turn our attention to the different styles of representing typing information in the original and revised HM(X) system. The essential difference between the new and the original HM(X) system lies in the representation of types. In the new HM(X) formulation, all types are represented in constraint form whereas in the original system some typing information can still be represented in term form, i.e. in the syntactic domain of types. In the introduction we gave the following judgment in constraint form

$$(\alpha = \alpha \rightarrow \beta), \{x : \alpha\} \vdash xx : \beta$$

which can also be represented in term form

$$\mathsf{true}, \{x : \mu\alpha.\alpha \rightarrow \beta\} \vdash xx : \beta$$

Section 2 showed that a judgment in constraint form can not always be represented in term form. Representing typing information in constraint form possibly results in a more expressive system.

We establish now sufficient conditions under which both representations are equivalent. The essential problem is how to transform a judgment in constraint form into a judgment in term form. A system in term representation is defined by the kind of constraints and the kind of types which are allowed to appear in typing judgments. In our original formulation of HM(X) [OSW99] this was formalized as follows. Given a constraint system X. Then the type language is defined through a term algebra $\mathcal{T}$ and the set of constraints which are allowed to appear in typing judgments is referred to as solved forms, denoted by $\mathcal{S}$. The set $\mathcal{S}$ of solved forms is a subset of all satisfiable constraints in X. For instance, the standard HM system can easily be modelled by using the Herbrand constraint system and taking $\mathcal{S}$ to consist only of $\{\}$ which represents the true constraint.

We assume that we are given a specific constraint system X, a term algebra $\mathcal{T}$ and a set $\mathcal{S}$ of solved form. We refer to $\mathcal{X}^c$ as an instance of the new HM(X) system. We refer to $\mathcal{X}^t$ as the corresponding type system in the original HM(X) system where constraints in typing judgments are further restricted by a set $\mathcal{S}$ of constraints in solved forms and types are now members of a term algebra $\mathcal{T}$. But still, both type systems $\mathcal{X}^c$ and $\mathcal{X}^t$ share the same underlying constraint system X.

We consider now the problem of transforming judgments in $\mathcal{X}^c$ into judgments in $\mathcal{X}^t$. The opposite direction is trivial. Given $C, \Gamma \vdash e : \tau$ we can always transform this judgment to $C \wedge (\alpha = \tau), \Gamma \vdash e : \alpha$ where α is a fresh type variable. The essential problem in transforming judgments in constraint form to judgments in term form lies in finding solutions to constraints.

Solutions are defined as follows, see [OSW99] for details. Given a constraint C in X and D a constraint in $\mathcal{S}$ and a substitution ϕ. Then the tuple (D, ϕ) is a *solution* to C iff $D \vdash^e \phi C$. We often refer to (D, ϕ) as a *normal form* of C. (D, ϕ) is a *principal* normal form if for all other normal forms (D', ϕ') we have that $\phi \leq \phi'$ [5] and $D' \vdash^e \phi' D$.

We furthermore remark that principal normal forms are equivalent up to variable renaming. Thus, we can define a well–defined function *normalize* from constraints to normal forms as follows:

$$\begin{array}{lll} \multicolumn{3}{l}{\mathit{normalize}(C)} \\ = & (D, \phi) & \text{if } (D, \phi) \text{ principal normal form of } C \\ = & \mathit{fail} & \text{otherwise} \end{array}$$

If the set $\mathcal{S}$ of solved forms equals the set of all satisfiable constraints, then the *normalize* function represents only a satisfiability. In case of an equational theory where $\mathcal{S}$ contains only the empty constraint $\{\}$, normalization corresponds to unification. Further examples can be found in [OSW99].

We extend the property of principal normal forms to constraint systems. Given a constraint system X over a term algebra $\mathcal{T}$ and a set $\mathcal{S}$ of solved forms in X. Then X satisfies the *principal constraint property* if for every constraint C in X, either C has a principal normal form or no normal form exists.

We are now in the position to state under which conditions there is a correspondence between judgments in constraint and term form.

Theorem 3 (Full and Faithful) *Given a constraint system X over a term algebra $\mathcal{T}$ and a set $\mathcal{S}$ of solved forms in X where X satisfies the principal constraint property. Then each judgment in $\mathcal{X}^c$ can be transformed to a judgment in $\mathcal{X}^t$ and vice versa.*

In the special case of equational theories, the principal constraint property corresponds to unitary unification. For instance, Example 2 which we gave in Section 2 does not satisfy the principal constraint property. The constraint

$$\begin{array}{l} (\alpha = +(n', m')) \wedge (n' = 1) \wedge (m' = \beta) \wedge \\ (\alpha = +(n'', m'')) \wedge (n'' = 2) \wedge (m'' = \gamma) \end{array}$$

does not have a most general unifier. Remember, we do not have a minus constructor in the index theory. Adding the minus constructor to the theory would allow to represent α in term form. But this would only fix the problem for this specific example. The index theory still does not satisfy the principal constraint property.

We find that the constraint–based formulation of the index type system is more expressive than the term–based formulation. Hence, program analyses in constraint form are more expressive than analyses in term form.

4.2 Substitution–based inference

We turn our attention to the inference problem in term–based systems. Kennedy's example shows that the standard inference algorithm does not infer principal types, even in the presence of unitary unification. The standard inference algorithm performs unification eagerly, after each step. This can be a problem in case of non–regular equational theories. An equational theory is *non–regular* if there exists types τ, τ' such that $(\tau = \tau')$ and $fv(\tau) \neq fv(\tau')$. For instance, dimension types form a non–regular theory. We find that $(prod(\alpha, i(\alpha)) = 1)$ but $fv(1) = \emptyset$. In [OSW99] we could already show that regular theories always have principal types

[5] The relation $\leq$ is an ordering relation on substitutions. For details see [OSW99].

$$\text{(Var)} \quad \frac{x : (\forall \bar{\alpha}.D \Rightarrow \tau) \in \Gamma \qquad \beta \text{ new}}{\exists \bar{\alpha}.((\beta = \tau) \wedge D), \Gamma \vdash^W x : \beta}$$

$$\text{(Abs)} \quad \frac{C, \Gamma_x.x : \alpha \vdash^W e : \gamma \qquad \beta \text{ new}}{\exists \alpha, \gamma.((\beta = \alpha \to \gamma) \wedge C), \Gamma_x \vdash^W \lambda x.e : \beta}$$

$$\text{(App)} \quad \frac{C_1, \Gamma \vdash^W e_1 : \alpha_1 \qquad C_2, \Gamma \vdash^W e_2 : \alpha_2 \qquad \alpha \text{ new}}{\exists \alpha_1, \alpha_2.(C_1 \wedge C_2 \wedge (\alpha_1 \preceq \alpha_2 \to \alpha)), \Gamma \vdash^W e_1 e_2 : \alpha}$$

$$\text{(Let)} \quad \frac{C_1, \Gamma_x \vdash^W e : \alpha \qquad C_3, \Gamma_x.x : \forall \alpha.C_1 \Rightarrow \alpha \vdash^W e' : \beta}{(\exists \alpha.C_1) \wedge C_3, \Gamma_x \vdash^W \mathsf{let}\ x = e \,\mathsf{in}\ e' : \beta}$$

$$\text{(Equ)} \quad \frac{C, \Gamma \vdash^W e : \alpha \qquad C =^e D}{D, \Gamma \vdash^W e : \alpha}$$

$$\text{(Term reconstruction)} \quad \frac{C, \Gamma \vdash^W e : \alpha \qquad (D, \psi) = \mathit{normalize}(C)}{\psi_{|fv(\Gamma)}, D, \Gamma \vdash^{W'} e : \psi(\alpha)}$$

Figure 3: Substitution–based type inference

under the assumption that the constraint system satisfies the principal constraint property (which corresponds to unitary unification in case of an equational theory). A similar result in the special case of regular equational theories was also shown by Rémy [Rém92].

Kennedy could give an adapted version of his inference algorithm which infers principal types for non–regular theories. But he can only state principal types under some additional conditions. We provide more details in the next section where we also discuss another alternative approach given by Valery Trifonov [TS96].

Here, we present a revised substitution–based inference system which always infers principal types if the constraint system satisfies the principal constraint property. The revised inference system can be found in Figure 3. It consists of two parts. The purely constraint–based inference system introduced in Section 3 and a term reconstruction step. Constraints are only accumulated in intermediate steps. Only in the last step we perform normalization which involves computing principal normal forms of constraint problems. That is what we call the term reconstruction step. In the standard formulation of the inference system, term reconstruction is done in each step. But Kennedy's example shows that this can be critical in case of non–regular theories.

We also find the (Equ) rule which allows to transform constraints into a semantically equivalent but better suitable representation. For efficiency reasons this can be very critical. Because we only accumulate constraints, constraints get duplicated in case of let–statements. Therefore, keeping constraints in compact representation is important. Such strategies depend on each specific constraint system and we postpone this topic to future work.

We find the following results.

Theorem 4 (Soundness of Inference) *Given a type environment* Γ *and a term* e. *If* $\psi, C, \Gamma \vdash^{W'} e : \tau$ *then* $C, \psi\Gamma \vdash e : \tau$ *where* $\psi C =^e C$ *and* $\psi\tau = \tau$.

The following completeness results extends the result given in [OSW99]. Completeness can be achieved under the assumption that the constraint system satisfies the principal constraint property. We do not need to restrict the result to regular theories as in [OSW99], or require additional conditions as done in [Ken96].

Theorem 5 (Completeness of Inference) *Given an instance of the original* HM(X) *system where* X *satisfies the principal constraint property. Given* $C, \phi\Gamma \vdash e : \sigma$ *such that* $\phi\Gamma$ *is realizable in* C. *Then* $\psi, D, \Gamma \vdash^{W'} e : \tau$ *for some substitution* ψ, *constraint* D *and type* τ *such that*

$$(\sigma_o, C_o) = \mathit{gen}(D, \psi\Gamma, \tau)$$
$$\psi \leq^{\phi'}_{fv(\Gamma)} \phi \qquad C \vdash^i \phi'\sigma_o \preceq \sigma \qquad C \vdash^e \phi' C_o$$

The generalization procedure *gen* is defined as follows:

$$\mathit{gen}(C, \Gamma, \sigma) = (\exists \bar{\alpha}.C, \forall \bar{\alpha}.C \Rightarrow \sigma)$$

where $\bar{\alpha} = (\mathit{fv}(\sigma) \cup \mathit{fv}(C)) \backslash \mathit{fv}(\Gamma)$.

A proof sketch of both theorems can be found in the appendix. We can furthermore simplify the completeness results to top–level programs which states that the inference system computes principal types.

Corollary 2 (Principal types) *Given an instance of the original* HM(X) *system where* X *satisfies the principal constraint property. Given a closed typing judgment* $\mathsf{true}, \Gamma \vdash e : \sigma$ *where* Γ *is realizable in* true. *Then* $\psi, D, \Gamma \vdash^{W'} e : \tau$ *for some substitution* ψ, *constraint* D *and type* τ *such that*

$$(\sigma_o, \mathsf{true}) = \mathit{gen}(D, \psi\Gamma, \tau)$$
$$\vdash^i \sigma_o \preceq \sigma$$

4.3 Discussion

In [Ken96] Kennedy discovered the problem of performing type inference in non–regular theories. The central problem seems that in non–regular theories there might be some implicitly free type variables. Such type variables need to be revealed when generalizing over free type variables before applying the let–rule. For instance, consider Example 1 in Section 2. There we applied a substitution to the whole judgment which revealed an implicitly free type variable, leaving the type environment unchanged. The question is whether such a treatment is always possible. Kennedy gave a sufficient axiomatization under which such implicitly free type variables can always be revealed. But it is not known yet whether there is a decidable algorithm which satisfies the axiomatization. For some specific theories he could give algorithms which satisfy the axiomatization. If additionally unification is unitary then a principal types theorem holds. In [Ken96] Kennedy posed the question whether these extra assumptions are necessary. We can answer this question now. Our revised substitution–based inference algorithm always infers principal types giving unitary unification. There are no further additional conditions necessary.

The basic trick we apply lies in postponing the term reconstruction step to the very end. In intermediate steps we always keep inference judgments in normalized form $C, \Gamma \vdash^W e : \alpha$. Then generalization is unproblematic. Term reconstruction is only performed at the very end. This achieves sound and complete type inference.

Valery Trifonov [TS96] introduces what he calls λ–lifting which also seems to overcome Kennedy's problem. He writes judgments in the form

$$\Gamma' \vdash e : \forall\bar{\alpha}.\Gamma \Rightarrow \tau \,|\, C$$

The type environment Γ' contains only let–bound variables whereas all other free variables are contained in Γ. Employing his notation would allow to type function y in Example 1 in Section 2 as follows

$$\emptyset \vdash y : \forall d_1, d_2.\Gamma.x : \dim(prod(d_1, d_2)) \Rightarrow \dim(d_1) \rightarrow \dim(d_2)$$

Now, y has the desired polymorphic type. His notation is quite different from the standard notation. For details see [TS96]. The λ–lifting notation was also employed by Francois Pottier [Pot98]. Both used λ–lifting in the context of subtyping. But it seems that their notation is also applicable to other settings. We are currently working [TP99] on establishing sufficient conditions under which we can prove the equivalence between the λ–lifting notation and the standard notation. We hope to report results in the near future.

5 Conclusion

We gave a purely constraint–based formulation of the HM(X) system which follows the principle; prefer constraints to terms whenever possible. This is motivated by the observation that type systems in term form are less flexible. In Section 2 we gave some specific examples. In contrast, constraints can describe fairly complex terms easily (as solutions for some variable). Furthermore, we could give a purely constraint–based formulation of the inference system. To achieve principal types we only need a decidable satisfiability test. Type soundness follows if we can provide monotype solutions to constraint problems.

The new purely constraint–based formulation of the HM(X) system allows to abstract away from any syntactic considerations when designing a new type system/program analysis. Usually, definitions of type systems have started with the definition of an appropriate domain of types. We hold that types should first be seen as a semantic domain and to introduce only as much syntax as necessary to describe all relevant types unambiguously. Nevertheless, type descriptions are important, especially when inferring types and representing the result in a readable form to the user. Then it becomes important to represent solutions in compact term representation. Section 2 revealed that the standard inference algorithm to infer types in term representation fails to infer principal types, even in the presence of most general unifiers. The problem occurs when representing constraint solutions as substitutions in case of non–regular theories. In Section 4.2 we presented an adaption of the substitution–based inference system. It not only solves the problem, but furthermore provides an alternative way to perform inference.

In this work we did not concentrate on a specific form of type system or program analysis. We rather made some general observations. Specifically, we provided a general framework to design Hindely/Milner systems in constraint form. Furthermore, we pointed out some, what we believe quite serious, problems which are apparent in term–based Hindley/Milner style type systems. The solution we provided, hopefully gives better insights when designing type systems/program analyses.

Acknowldegments

We are grateful for helpful discussions with Martin Odersky and Valery Trifonov. Part of this work was carried out at the the University of South Australia. The first author thanks for the hospitality and system support while visiting the University of South Australia. Furthermore, the first author acknowledges support by DARPA Grant F30602-96-2-0232. The second author was partially supported by the Deutsche Forschungsgemeinschaft through SFB 378 at the Universität des Saarlandes, 1996–98.

References

[AW93] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *FPCA '93: Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, pages 31–41, New York, June 1993. ACM Press.

[CHO92] Kung Chen, Paul Hudak, and Martin Odersky. Parametric type classes. In *Proc. of Lisp and F.P.*, pages 170–191. ACM Press, June 1992.

[DHM95] Dirk Dussart, Fritz Henglein, and Christian Mossin. Polymorphic binding-time analysis in polynomial time. In *Proceedings of SAS*, pages 118–135. Springer Verlag, September 1995.

[EST95a] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *OOPSLA '95 Conference Proceedings*, volume 30(10) of *ACM SIGPLAN Notices*, pages 169–184, 1995.

[EST95b] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type inference for recursively constrained types and its application to object oriented programming. In *Electronic Notes in Theoretical Computer Science*, volume 1, 1995.

[HMT71] L. Henkin, J.D. Monk, and A. Tarski. *Cylindric Algebra.* North-Holland Publishing Company, 1971.

[JM94] Joxan Jaffar and Michael Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19(20):503–581, 1994.

[Jon92] Mark P. Jones. *Qualified Types: Theory and Practice.* D.phil. thesis, Oxford University, September 1992.

[Kae92] S. Kaes. Type inference in the presence of overloading, subtyping and recursive types. volume 5, pages 193–204, 1992. Proceedings of the 1992 ACM Conference on LISP and Functional Programming.

[Ken94] Andrew Kennedy. Dimension types. In Donald Sannella, editor, *Programming Languages and Systems—ESOP'94, 5th European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 348–362, Edinburgh, U.K., 11–13 April 1994. Springer.

[Ken96] Andrew J. Kennedy. Type inference and equational theories. Technical Report LIX/RR/96/09, LIX, Ecole Polytechnique, 91128 Palaiseau Cedex, France, September 1996.

[KMM91] P.C. Kanellakis, H.G. Mairson, and J.C. Mitchell. Unification and ML type reconstruction. In *Computational Logic, Essays in Honor of Alan Robinson*, pages 444–478. MIT Press, 1991.

[LBB84] D. S. Lankford, G. Butler, and B. Brady. Abelian group unification algorithms for elementary terms. In W. Bledsoe and W. Loveland, editors, *Automated Theorem Proving: After 25 Years.* AMS, 1984.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, Dec 1978.

[NP95] Tobias Nipkow and Christian Prehofer. Type reconstruction for type classes. *Journal of Functional Programming*, 5(2):201–224, 1995.

[Oho95] Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM TOPLAS*, 6(6):805–843, November 1995.

[OSW99] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1), 1999.

[OWW95] Martin Odersky, Philip Wadler, and Martin Wehr. A second look at overloading. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 135–146, La Jolla, California, June 25–28, 1995. ACM SIGPLAN/SIGARCH and IFIP WG2.8, ACM Press.

[Pot98] Francois Pottier. A framework for type inference with subtyping. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, pages 228–238, September 1998.

[Rém89] D. Rémy. Typechecking records and variants in a natural extension of ML. pages 77–88. ACM, January 1989.

[Rém92] Didier Rémy. Extending ML type system with a sorted equational theory. Technical Report 1766, INRIA, October 1992.

[Sar93] Vijay A. Saraswat. *Concurrent Constraint Programming.* Logic Programming Series, ACM Doctoral Dissertation Award Series. MIT Press, Cambridge, Massachusetts, 1993.

[Smi91] Geoffrey S. Smith. *Polymorphic type inference for languages with overloading and subtyping.* PhD thesis, Cornell University, Ithaca, NY, August 1991.

[Sul98] Martin Sulzmann. Polymorphism in Hindley/Milner Style Type Systems with Constraints. Research Report YALEU/DCS/RR-1155 (revised version), Yale University, Department of Computer Science, August 1998.

[TJ92] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. In *Seventh Annual IEEE Symposium on Logic in Computer Science, Santa Cruz, California*, pages 162–173, Los Alamitos, California, June 1992. IEEE Computer Society Press.

[TP99] Valery Trifonov and Francois Pottier. Personal communication, February 1999.

[TS96] Valery Trifonov and Scott Smith. Subtyping Constrained Types. In *Proceedings of the Third Internationl Static Analysis Symposium*, volume 1145 of *LNCS*, pages 349–365, 1996.

[VHJW96] Cordelia V.Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM TOPLAS*, 18(2):109–138, March 1996.

[Wan89] Mitchell Wand. Type inference for record concatenation and multiple inheritance. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, pages 92–97, June 1989.

[Xi98] Hongwei Xi. *Dependent Types in Practical Programming.* PhD thesis, Carnegie Mellon University, 1998. forthcoming.

[Zen98] Christoph Zenger. *Index Types.* PhD thesis, Universität Karlsruhe, 1998. forthcoming.

6 Substitution–based inference

We give a proof sketch of soundness and completeness of the substitution–based inference system in Section 4.2. The proofs relies on sound and complete inference for the constraint–based system. Soundness and completeness proofs for the constraint–based system are rather straightforward and left out due to space restrictions. The technical problem lies in transforming judgments in constraint form into judgments in term form. Remember, the inference system in Section 4.2 performs inference first in the purely constraint–based fragment of HM(X). Then we perform the term reconstruction step. The output is a judgment in the term–based fragment of HM(X). Technically, we need to show how to transform judgments/proof trees of the constraint–based fragment of HM(X) into judgments/proof trees of the term–based fragment. In order to distinguish between both kinds of derivations we refer to $\vdash^{\mathcal{X}^c}$ as the derivation in the constraint–based fragment and to $\vdash^{\mathcal{X}^t}$ as the derivation in the term–based fragment. Furthermore, we refer to $\mathcal{X}^c$ as a type system instance of the constraint–based fragment, respectively to $\mathcal{X}^t$ as a type system instance of the corresponding term–based fragment.

The following definition approximates type schemes in constraint form by type schemes in term form.

Definition 1 *Given a type σ in $\mathcal{X}^c$ and a type σ' and a constraint C in $\mathcal{X}^t$. Then $C \vdash \sigma \rightsquigarrow \sigma'$ iff the following two conditions hold*

- $\vdash^i \sigma \preceq \sigma'$
- *Given a type σ'' in $\mathcal{X}^t$ such that $C \vdash^i \sigma \preceq \sigma''$. Then $C \vdash^i \sigma' \preceq \sigma''$.*

The judgment $\rightsquigarrow$ extends to type environments in the usual way.

The next two lemmas are used in the soundness proof to transform proof trees in the constraint–based fragment of HM(X) into proof trees in the term–based fragment.

Lemma 4 *Given a typing judgment $C, \Gamma \vdash^{\mathcal{X}^c} e : \tau$ and Γ' in $\mathcal{X}^t$ such that $C \vdash \Gamma \rightsquigarrow \Gamma'$. Then $C, \Gamma' \vdash^{\mathcal{X}^c} e : \tau$.*

Lemma 5 (Transformation) *Given a type environment Γ' in $\mathcal{X}^t$ and a constraint C in $\mathcal{X}^t$ where $C \vdash \Gamma \rightsquigarrow \Gamma'$. Then the following holds.*

- *If $C, \Gamma \vdash^{\mathcal{X}^c} e : \tau$ then $C, \Gamma' \vdash^{\mathcal{X}^t} e : \tau$.*
- *If $C, \Gamma \vdash^{\mathcal{X}^c} e : \sigma$ where σ realizable in C then $C, \Gamma' \vdash^{\mathcal{X}^t} e : \sigma'$ where $C \vdash \sigma \rightsquigarrow \sigma'$.*

Proof: The proof proceeds by induction on the number of occurrences of (Sub), ($\forall$ Elim) and ($\forall$ Intro) and then on the expression size. We only show the interesting cases.

Case (Let) We have the following situation:

$$\frac{C, \Gamma_x \vdash^{\mathcal{X}^c} e : \sigma \qquad C, \Gamma_x.x : \sigma \vdash^{\mathcal{X}^c} e' : \tau'}{C, \Gamma_x \vdash^{\mathcal{X}^c} \mathsf{let}\ x = e\,\mathsf{in}\ e' : \tau'}$$

W.l.o.g. we can assume that σ is a type scheme. The same argument would hold for types. By induction we find

$$C, \Gamma'_x \vdash^{\mathcal{X}^t} e : \sigma' \qquad C \vdash \sigma \rightsquigarrow \sigma'$$

We rewrite the right part of the premise (we can apply Lemma 4) and obtain

$$C, \Gamma_x.x : \sigma' \vdash^{\mathcal{X}^c} e' : \tau'$$

We can now apply the induction hypothesis and obtain

$$C, \Gamma'_x.x : \sigma' \vdash^{\mathcal{X}^t} e' : \tau'$$

Finally, we apply the (Let) rule and find

$$C, \Gamma'_x \vdash^{\mathcal{X}^t} \mathsf{let}\ x = e\,\mathsf{in}\ e' : \tau'$$

Case ($\forall$ Elim) We have the following situation [6]:

$$\frac{C, \Gamma \vdash^{\mathcal{X}^c} e : \forall\bar{\alpha}.D \Rightarrow \tau \qquad C \vdash^e [\bar{\tau}/\bar{\alpha}]D}{C, \Gamma \vdash^{\mathcal{X}^c} e : [\bar{\tau}/\bar{\alpha}]\tau}$$

By induction we find

$$C, \Gamma' \vdash^{\mathcal{X}^t} e : \sigma'$$

where $C \vdash (\forall\bar{\alpha}.D \Rightarrow \tau) \rightsquigarrow \sigma'$. We also know that

$$C \vdash^i \forall\bar{\alpha}.D \Rightarrow \tau \preceq [\bar{\tau}/\bar{\alpha}]\tau$$

Because of the properties we have put on $\rightsquigarrow$ we get that $C \vdash^i \sigma' \preceq [\bar{\tau}/\bar{\alpha}]\tau$. But then we obtain

$$C, \Gamma' \vdash^{\mathcal{X}^t} e : [\bar{\tau}/\bar{\alpha}]\tau$$

Case ($\forall$ Intro) We have the following situation:

$$\frac{C \wedge D, \Gamma \vdash^{\mathcal{X}^c} e : \tau \qquad \bar{\alpha} \notin fv(C) \cup fv(\Gamma)}{C \wedge \exists\bar{\alpha}.D, \Gamma \vdash^{\mathcal{X}^c} e : \forall\bar{\alpha}.D \Rightarrow \tau}$$

We know $\forall\bar{\alpha}.D \Rightarrow \tau$ is realizable in $C \wedge \exists\bar{\alpha}.D$. That means, there is a ϕ (we know $\phi = [\bar{\tau}/\bar{\alpha}]$ for some $\bar{\tau}$) which only acts on $\bar{\alpha}$ such that

$$C \wedge \exists\bar{\alpha}.D \vdash^e \phi D$$

We can conclude that

$$C \wedge \exists\bar{\alpha}.D \vdash^e C \wedge \phi D$$

By assumption $C \wedge \exists\bar{\alpha}.D$ is in $\mathcal{X}^t$. But that means $(C \wedge \exists\bar{\alpha}.D, \phi)$ is a normal form of $(C \wedge D, id)$. Note, it holds that $\phi C =^e C$ and $\phi(\exists\bar{\alpha}.D) =^e \exists\bar{\alpha}.D$. Hence,

$$(\psi(C \wedge D), \psi) = normalize(C \wedge D, id)$$

and because $\psi \leq \phi$ we know that ψ acts only on $\bar{\alpha}$. But then we find

$$\psi(C \wedge D) =^e C \wedge \psi D$$

We apply substitution ψ to the premise and obtain

$$C \wedge \psi D, \Gamma \vdash^{\mathcal{X}^c} e : \psi\tau$$

We apply now the induction hypothesis and find

$$C \wedge \psi D, \Gamma' \vdash^{\mathcal{X}^t} e : \psi\tau$$

[6] Note, we consider a slight variation of the constraint–based fragment of HM(X) where the domain of type is the equal to the domain of types in the term–based fragment. Furthermore, we use the substitution–based formulation of the quantifier elimination rule. This fragment enjoys the same properties as the purely constraint–based fragment.

W.l.o.g. $\bar{\alpha} \notin fv(\Gamma')$. Application of rule ($\forall$ Intro) yields

$$C \wedge \exists\bar{\alpha}.\psi D, \Gamma' \vdash^{\mathcal{X}^t} e : \forall\bar{\alpha}.\psi D \Rightarrow \psi\tau$$

It holds that $\phi D \vdash^e \exists\bar{\alpha}.\psi D$. Then, we find that $C \wedge \exists\bar{\alpha}.D \vdash^e C \wedge \exists\bar{\alpha}.\psi D$. This leads to

$$C \wedge \exists\bar{\alpha}.D, \Gamma' \vdash^{\mathcal{X}^t} e : \forall\bar{\alpha}.\psi D \Rightarrow \psi\tau$$

It is easy to see that

$$\vdash^i (\forall\bar{\alpha}.D \Rightarrow \tau) \preceq (\forall\bar{\alpha}.\psi D \Rightarrow \psi\tau)$$

Assume $\forall\beta.D' \Rightarrow \tau'$ in $\mathcal{X}^t$ such that

$$C \wedge \exists\bar{\alpha}.D \vdash^i (\forall\bar{\alpha}.D \Rightarrow \tau) \preceq (\forall\beta.D' \Rightarrow \tau')$$

We have to show that

$$C \wedge \exists\bar{\alpha}.D \vdash^i (\forall\bar{\alpha}.\psi D \Rightarrow \psi\tau) \preceq (\forall\beta.D' \Rightarrow \tau')$$

This holds because we have computed the principal normal form.

■

We can observe that if we want to apply the above lemma to a judgment $C, \Gamma \vdash^{\mathcal{X}^c} e : \sigma$ we have to ensure that all type schemes, generated in intermediate steps, are realizable. This condition gets fulfilled through the term reconstruction step. We always keep quantified constraints on the left–hand side of the turnstile, see the ($\forall$ Intro) rule. Then normalization yields solutions to quantified constraints. Hence, type schemes are realizable.

We are now in the position to prove Theorem 4.

Theorem 6 (Soundness of Inference) *Given a type environment Γ and a term e. If $\psi, C, \Gamma \vdash^{W'} e : \tau$ then $C, \psi\Gamma \vdash e : \tau$ where $\psi C =^e C$ and $\psi\tau = \tau$.*

Proof: We only consider the final step of the derivation $\vdash^W$. We have

$$\frac{C, \Gamma \vdash^W e : \alpha \qquad (D, \psi) = \mathit{normalize}(C)}{\psi_{|fv(\Gamma)}, D, \Gamma \vdash^{W'} e : \psi(\alpha)}$$

From the premise we obtain

$$C, \Gamma \vdash^{\mathcal{X}^c} e : \alpha$$

This relies on the fact that constraint–based inference is sound. Then we find [7]

$$D, \psi\Gamma \vdash^{\mathcal{X}^c} e : \psi(\alpha)$$

Normalization ensures that $\psi\Gamma$ is in $\mathcal{X}^t$. We apply the Transformation Lemma and obtain

$$D, \psi\Gamma \vdash^{\mathcal{X}^t} e : \psi(\alpha)$$

and we are done.

■

It remains to prove completeness.

[7] Note, we consider a slight variation of the constraint–based fragment of HM(X) where the domain of type is the equal to the domain of types in the term–based fragment. This fragment enjoys the same properties as the purely constraint–based fragment.

Theorem 7 (Completeness of Inference) *Given an instance of the original* HM(X) *system where* X *satisfies the principal constraint property. Given $C, \phi\Gamma \vdash^{\mathcal{X}^t} e : \sigma$ such that $\phi\Gamma$ realizable in C. Then $\psi, D, \Gamma \vdash^W e : \tau$ for some substitution ψ, constraint D and type τ such that*

$$(\sigma_o, C_o) = \mathit{gen}(D, \psi\Gamma, \tau)$$
$$\psi \leq^{\phi'}_{fv(\Gamma)} \phi \qquad C \vdash^i \phi'\sigma_o \preceq \sigma \qquad C \vdash^e \phi' C_o$$

Proof: Given $C, \phi\Gamma \vdash^{\mathcal{X}^t} e : \sigma$. We perform type inference in $\mathcal{X}^c$. We obtain

$$\tilde{C}, \Gamma \vdash^W e : \alpha$$

such that

$$\mathit{id} \leq^{\phi''}_{fv(\Gamma)} \phi \quad C \vdash^e \phi'' C'_o \quad C \vdash^i \phi''\sigma'_o \preceq \sigma$$

where $\sigma'_o = \forall\alpha.\tilde{C}$ and $C_o = \exists\alpha.\tilde{C}$. Furthermore, we assume that $\sigma = \forall\bar{\gamma}.D' \Rightarrow \tau'$. From $C \vdash^i \phi'\sigma_o \preceq \sigma$ we can deduce there exists $\pi = [\tau/\alpha]$ such that

$$C \wedge D' \vdash^e (\pi \circ \phi'')\tilde{C} \qquad C \wedge D' \vdash^i (\pi \circ \phi'')\tau \preceq \tau' \qquad (7)$$

But then we know the principal normal form of $\tilde{C}$ exists. We set

$$(D, \psi) = \mathit{normalize}(\tilde{C})$$

and find that

$$\psi \leq^{\phi'''} (\pi \circ \phi'') \qquad C \wedge D' \vdash^e \phi''' D \qquad (8)$$

Furthermore, we find that

$$\psi, D, \Gamma \vdash^{W'} e : \psi(\alpha)$$

We define $(\sigma_o, C_o) = \mathit{gen}(D, \psi\Gamma, \psi(\alpha))$ where we assume $\sigma_o = \forall\bar{\beta}.D \Rightarrow \psi(\alpha)$ and $C_o = \exists\bar{\beta}.D$. From 7 and 8 we follow that

$$C \vdash^i \phi'''(\forall\bar{\beta}.D \Rightarrow \psi(\alpha)) \preceq \sigma$$

Furthermore, we find that $\psi \leq^{\phi'''}_{fv(\Gamma)} \phi$.

It remains to prove that $C \vdash^e \phi'' C_o$. We know that $\phi\Gamma$ is realizable in C but then also σ is realizable in C. Hence, there is a θ such that $C \vdash^e \theta D'$. We set $\phi' = \theta \circ \phi'''$. The substitution ϕ' satisfies the same properties as the substitution ϕ'''. From $C \wedge D' \vdash^e \phi''' D$ we can follow that $C \vdash^e \phi' C_o$. We find that

$$\psi \leq^{\phi'}_{fv(\Gamma)} \phi \qquad C \vdash^i \phi'\sigma_o \preceq \sigma \qquad C \vdash^e \phi' C_o$$

which establishes the completeness result.

■