Type Inference with Constrained Types

Martin Odersky
University of South Australia, School of Computer and Information Science, The Levels, South Australia 5095

Martin Sulzmann*
Yale University, Department of Computer Science, New Haven, CT 06520-8285

Martin Wehr
University of Edinburgh, Laboratory for Foundations of Computer Science (LFCS), EH7 3JZ Edinburgh

Keywords: constrained types, subtypes, record types, constraint systems, constraint solving, type inference, principal types, ideal semantics

We present a general framework HM(X) for type systems with constraints. The framework stays in the tradition of the Hindley/Milner type system. Its type system instances are sound under a standard untyped compositional semantics. We can give a generic type inference algorithm for HM(X) so that, under sufficient conditions on X, type inference will always compute the principal type of a term. We discuss instances of the framework that deal with polymorphic records, equational theories and subtypes.

1. Introduction

Many type systems extend the Hindley/Milner[Mil78] system with constraints. Examples are found in record systems [Oho95, Rém89, Wan89], overloading [Jon92, Kae92, VHJW96, NP93, CHO92, OWW95, BM97], and subtyping [CCH+89, BSvG95, AW93, EST95b, Smi91]. Extensions of Hindley/Milner with constraints are also increasingly popular in program analysis [DHM95, TJ92].

Even though these type systems use different constraint domains, they are largely alike in their type-theoretic aspects. In this paper we present a general framework HM(X) for Hindley/Milner style type systems with constraints, analogous to the CLP(X) framework in constraint logic programming [JM94]. Particular type systems can be obtained by instantiating the parameter X to a specific constraint system. The Hindley/Milner system itself is obtained by instantiating X to the standard Herbrand constraint system.

By and large, the treatment of constraints in type systems has been syntactic: constraints were regarded as sets of formulas, often of a specific form. On the other hand, constraint programming now generally uses a semantic definition of constraint systems, taking a constraint system as a cylindric algebra with some additional properties [HMT71, Sar93]. Cylindric algebras define a projection operator $\exists\bar{\alpha}$ that binds some subset of variables $\bar{\alpha}$ in the constraint. In the usual case where constraints are boolean algebras, projection corresponds to existential quantification.

Following the lead of constraint programming, we treat a constraint system as a cylindric algebra with a projection operator. Projection is very useful for our purposes for two reasons: First, projection allows us to formulate a logically pleasing and pragmatically useful rule ($\forall$ Intro) for quantifier introduction:

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

Here, C and D are constraints over the type variables in the type context Γ and the type scheme τ. We discuss some other proposals for quantifier introduction and show how our approach improves already existing ones.

Second, projection is an important source of opportunities for simplifying constraints [Jon95, Pot96, EST95a]. In our framework, simplifying means changing the syntactic representation of a constraint without changing its denotation. For example, the subtyping constraint

$$\exists\beta.(\alpha <: \beta) \wedge (\beta <: \gamma)$$

*Supported by DARPA Grant F30602-96-2-0232.

© (Year) John Wiley & Sons, Inc.

can safely be simplified to

$$(\alpha <: \gamma)$$

since the denotation is the same for both constraints. Without the projection operator, the two constraints would be different, since one restricts the variable β while the other does not.

Two of the main strengths of the Hindley/Milner system are a type soundness result and the existence of a type inference algorithm that computes principal types. HM(X) stays in the tradition of the Hindley/Milner type system. Type systems in HM(X) are sound under a standard untyped compositional semantics provided the underlying constraint system X is sound. This result can be summarized in the slogan "well–typed programs can not go wrong". One of the key ideas of our paper is to present sufficient conditions on the constraint domain X so that the principal types property carries over to HM(X). The conditions are fairly simple and natural. For those constraint systems meeting the conditions, we present a generic type inference algorithm that will always yield the principal type of a term.

The type inference algorithm is explained by treating the typing problem itself as a constraint. Generally, the constraint system X needs to be rich enough to express all constraint problems that can be generated by type derivations. On the other hand, we admit the possibility that constraints on the left hand side of the turnstile and in type schemes come from a more restricted set which we call *solved forms.* The task of type inference is then to split a typing problem into a substitution and a residual constraint in solved form. This we call *constraint normalization.* We require that normalization always yields a "best" solution, if there is a solution at all. This ensures that the type inference algorithm computes principal types.

Our work generalizes Milner's results to systems with non-standard constraints and thus makes it possible to experiment with new constraint domains without having to invent yet another type inference algorithm and without having to repeat the often tedious proofs of soundness and completeness of type inference.

Object–oriented languages. Object oriented languages are often based on record calculi and type systems supporting a notion of subtyping. Cardelli/Wegner [CW85] gave an early survey about general research directions. Reynolds [Rey85] and Mitchell [Mit84] are foundational papers that develop basic concepts of constraints and subtyping. Palsberg [Pal95] gave an efficient inference algorithm for a calculus of objects.

Subtyping is orthogonal to the notion of parametric polymorphism supported by the Hindley/Milner system. A natural approach for a type system that supports both notions is to add subtype constraints to types [AW93, EST95a]. Such systems can be expressed as instances of the HM(X) system (or, if they are based on recursive records, in an extension of it). Other encodings of object-oriented languages forgo subtyping, and are instead based on calculi for extensible records or overloading [Rém89, Wan89, OWW95, BM97]. Such systems can also be regarded as instances of our framework. We demonstrate this using Ohori's system [Oho95] as an example.

Outline. The rest of this paper is structured as follows: The next section discusses previous approaches to type systems with constraints. Section 3 gives a characterization of constraint systems. Section 4 presents our framework HM(X) for Hindley/Milner style type systems with constraints. Section 5 presents an ideal semantics for type systems in the framework from which a type soundness theorem is derived. Section 6 establishes conditions on the constraint system so that type inference is feasible and a principal types theorem holds. Section 7 describes as an instance of our framework a type system for polymorphic records. Section 8 concludes.

2. Related work

Hindley/Milner style type systems with constrained types have been used in a number of instances. All such type systems extend the type judgments $\Gamma \vdash e : \sigma$ of the original Hindley/Milner system with a constraint hypothesis on the left side of the turnstile, written $C, \Gamma \vdash e : \sigma$. Furthermore, they extend the type schemes $\forall\bar{\alpha}.\tau$ of the Hindley/Milner system with a constraint component; we write

$$\forall\bar{\alpha}.C \Rightarrow \tau$$

to express that the constraint C restricts the types that can legally be substituted for the bound variables $\bar{\alpha}$.

All type systems have essentially the same rule for eliminating quantifiers, which we write as follows:

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

The rule is a refinement of the corresponding rule in the Hindley/Milner system. It says that, when instantiating a type scheme $\forall\bar{\alpha}.D \Rightarrow \tau'$, the only valid instances are those instances $[\bar{\tau}/\bar{\alpha}]\tau'$ which satisfy the constraint part D of the type scheme.

While there is agreement about the proper technique for eliminating quantifiers in type schemes, there is remarkable disagreement about the proper way to introduce them. Figure 1 shows four different rules that have all been proposed in the literature. We have edited these rules somewhat to present them in a uniform style, and have attempted to compensate for the considerable variations in detail between published type sys-

No satisfiability check[Jon92]:

$$\frac{C \wedge D, \Gamma \vdash e : \tau \qquad \bar{\alpha} \notin fv(C) \cup fv(\Gamma)}{C, \Gamma \vdash e : \forall \bar{\alpha}.D \Rightarrow \tau} \qquad (\forall \text{ Intro-1})$$

Weak satisfiability check[AW93]:

$$\frac{C \wedge D, \Gamma \vdash e : \tau \qquad \exists D \qquad \bar{\alpha} \notin fv(C) \cup fv(\Gamma)}{C, \Gamma \vdash e : \forall \bar{\alpha}.D \Rightarrow \tau} \qquad (\forall \text{ Intro-2})$$

Strong satisfiability check[Smi91]:

$$\frac{C \wedge D, \Gamma \vdash e : \tau \qquad C \vdash [\bar{\tau}/\bar{\alpha}]D \qquad \bar{\alpha} \notin fv(C) \cup fv(\Gamma)}{C, \Gamma \vdash e : \forall \bar{\alpha}.D \Rightarrow \tau} \qquad (\forall \text{ Intro-3})$$

Duplication[EST95b]:

$$\frac{C \wedge D, \Gamma \vdash e : \tau \qquad \bar{\alpha} \notin fv(C) \cup fv(\Gamma)}{C \wedge D, \Gamma \vdash e : \forall \bar{\alpha}.D \Rightarrow \tau} \qquad (\forall \text{ Intro-4})$$

FIG. 1. Versions of the quantifier introduction rule

tems. Even though these details matter for each particular type system, we have to abstract from them here in order to concentrate on general principles. We now discuss each of the four schemes in turn.

In his work in qualified types [Jon92], Jones uses a general framework for type qualification with a rule equivalent to rule (∀ Intro-1). Any constraint can be shifted from the assumption on the left to the type scheme on the right of the turnstile; it is not checked whether the traded constraint is satisfiable. This might lead to programs that are well-typed as a whole, even though some parts have unsatisfiable constraints.

To give an example, assume that our constraints are subtyping constraints ($\leq$) in a type system with classes and a subtyping relation determined by programmer declarations. Let us assume that there is a parametrized class $\mathsf{List}\,\alpha$ which is a subtype of type $\mathsf{Comparable}\,(\mathsf{List}\,\alpha)$, where $\mathsf{Comparable}$ is declared as follows:

```
type Comparable α = {less : α → Bool}
```

Let us further assume that there is a value Nil of type $\forall\alpha.\mathsf{true} \Rightarrow \mathsf{List}\,\alpha$ that represents the empty list. Consider the following (nonsensical) program.

Example 1.

```
let
 f: ∀α.(List α ≤ Comparable α) ⇒ List α → List α
 f x = if x.less(true) then x else Nil
in 1
```

We use a Haskell-style notation, adding type annotations for illustration purposes. Using rule (∀ Intro-1), the program in Figure 1 is well-typed, even though we would not expect the constraint in function f's type scheme to have a solution, since the function type $\mathsf{List}\,\alpha$ would not be a subtype of $\mathsf{Comparable}\,\mathsf{Bool}$.

In the ideal semantics of types [MPS86], which represents universal quantification by intersection, f's type would be an empty intersection, which is equal to the whole type universe including the error element *wrong*. However, the whole program in Figure 1 is still sound because every application of f must provide a valid instantiation of the constraint. Since the constraint is unsatisfiable, no application is possible. In essence, Jones treats constraints as proof obligations that have to be fulfilled by presenting "evidence" at the *instantiation* site. This scheme is clearly inspired by Haskell's implementation of overloading by dictionary passing. It runs into problems if one ever wants to compute a value of a constrained type without any instantiation sites, as in the following slight variation of Example 1.

Example 2.

```
let
   y: ∀α.(List α ≤ Comparable α) ⇒ Bool
   y = Nil.less(true)
in 1
```

Jones excludes this code on the grounds that y's type is ambiguous, but it is unclear how to generalize this restriction to arbitrary constraint systems.

Nevertheless, it is possible to integrate Jones' approach into our HM(X) framework, thus giving it a semantic basis independent of dictionary passing. The essential idea is that we have to restrict ourselves to constraint systems in which projections of solved constraints are trivial, i.e $\vdash^e \exists\alpha.C$, for all constraints C that can appear on the left hand side of the turnstile, and for all type variables $\alpha \in fv(C)$. In this case, our rule (∀ Intro) simplifies to (∀ Intro-1).

Note that trivial projections correspond well to Haskell's "open world" assumption, which says that the range of possible instance types for an overloaded operation is not fixed in advance. Therefore, we can never rule out that a given constraint which still has free variables might have a solution. A formalization of this principle using a "bottom type" [OWW95] makes it possible to define a compositional semantics for Haskell–style overloading.

In the type system of Aiken/Wimmers [AW93], moving a constraint from the left hand side of the turnstile to the right-hand side is allowed only if the constraint is satisfiable (i.e. has a solution). Hence, none of the previous examples would be typable with rule (∀ Intro-2), which they use. However, this example is typable.

Example 3.

```
let
   f: ∀β.β → Int
   f x =
      let y: ∀α.(List α ≤ Comparable β) ⇒ Bool
         y = Nil.less(x)
      in 1
in f true
```

The constraint $\mathsf{List}\,\alpha \leq \mathsf{Comparable}\,\beta$ has a solution, namely $\beta = \mathsf{List}\,\alpha$. Therefore, using rule (∀ Intro-2) we can generalize y's type to

$$\forall\alpha.(\mathsf{List}\,\alpha \leq \mathsf{Comparable}\,\beta).\mathsf{Bool}.$$

On the other hand, if we substitute the actual parameter true in f's definition, we get again Example 1 which is not typable under the system with (∀ Intro-2). Hence, the system with (∀ Intro-2) does not enjoy the property of subject reduction, which says that if a term is typable then its reduction instances are typable as well. In a later version, they use rule (∀ Intro-4) instead.

Where Aiken and Wimmers require only a weak form of satisfiability for traded constraints, G. Smith requires a strong one [Smi91]. In rule (∀ Intro-3), the traded constraint D must be solvable by instantiation of only the quantified variables $\bar{\alpha}$. Hence, all three previous examples would be untypable under his system. However, (∀ Intro-3) rule seems overly restrictive, depending on the constraint system used. For instance, let's assume that Comparable has precisely two instances:

```
Int ≤ Comparable Int
Char ≤ Comparable Char
```

Now consider the following program:

Example 4.

```
let
   f: ∀β.β → Int
   f x =
      let g y = y.less(x)
      in 1
in 1
```

When typing the definition of g, Smith's system requires a solution of the constraint $\tau \leq \mathsf{Comparable}\,\tau$, where τ is y's type. Two solutions exist: $\tau = \mathsf{Int}$ or $\tau = \mathsf{Char}$, and there is no best type for y that improves on both solutions.

The system of the Hopkins Objects Group [EST95b] differs from the previous three systems in that in rule (∀ Intro-4) the constraint D is copied instead of moved; there are no restrictions on when the copying can take place. Under this scheme, the first three examples would be rejected and the fourth one would be accepted, which corresponds fairly well to our intuition. At the same time, rule (∀ Intro-4) seems strange in that its conclusion contains two copies of the constraint D, one in which the type variables α are bound and one in which they are free. Actually, the Hopkins Objects Group uses a slightly different system in which generalization is coupled with the let rule and one of the two constraints undergoes a variable renaming. HM(X) can be seen as the proper logical formulation of their more algorithmically–formulated type system. Furthermore, instead of dealing exclusively with subtype constraints, we admit arbitrary constraint systems.

3. Constraint systems

We present a characterization of constraint systems along the lines of Henkin [HMT71] and Saraswat [Sar93]. Building on the standard notions of *simple* and *cylindric constraint systems* we introduce *term constraint systems* as constraint systems which have a well-behaved notion of substitution. These constraint systems will be the parameter which allows our framework to be customized to different application domains.

We start with the definition of a simple constraint system.

Definition. *A* simple constraint system *is a structure* $(\Omega, \vdash^e)$ *where* Ω *is a non–empty set of* tokens *or* (primitive) constraints. *We also refer to such constraints as* predicates. *The relation* $\vdash^e \subseteq p\Omega \times \Omega$ *is an* entailment relation *where* $p\Omega$ *is the set of finite subsets of* Ω. *We call* $C \in p\Omega$ *a constraint set or simply a constraint.*

A constraint system $(\Omega, \vdash^e)$ *must satisfy for all constraints* $C, D \in p\Omega$*:*

C1 $C \vdash^e P$ *whenever* $P \in C$ *and*
C2 $C \vdash^e Q$ *whenever*
$C \vdash^e P$ *for all* $P \in D$ *and* $D \vdash^e Q$

We extend $\vdash^e$ to be a relation on $p\Omega \times p\Omega$ by: $C \vdash^e D$ iff $C \vdash^e P$ for every $P \in D$. Furthermore, we define $C =^e D$ iff $C \vdash^e D$ and $D \vdash^e C$. The term $\vdash^e C$ is an abbreviation for $\emptyset \vdash^e C$ and $\mathsf{true} = \{P \mid \emptyset \vdash^e P\}$ represents the true element.

We give an example how to generate a simple constraint system based on a first–order language $\mathcal{L}$.

Example 5. For any first–order language $\mathcal{L}$*, and countably infinite set of variables Var, take* Ω *to be an arbitrary subset of open* ($\mathcal{L}$*, Var*)*–formulas, and* $\vdash^e$ *to be the entailment relation with respect to some class* Δ *of* $\mathcal{L}$*–structures. That is,* $\{P_1, \ldots, P_n\} \vdash^e Q$ *iff for every structure* $M \in \Delta$*, an* M*–valuation realizes* Q *whenever it realizes each of* $P_1, \ldots, P_n$*. Such a* $(\Omega, \vdash^e)$ *is a* *simple constraint system.*

We now extend a simple constraint system with a *projection operator* $\exists\bar{\alpha}$. This leads to a cylindric constraint system.

Definition. *A* cylindric constraint system *is a structure* $\mathcal{CS} = (\Omega, \vdash^e, Var, \{\exists\alpha \,|\, \alpha \in Var\})$ *such that:*

- $(\Omega, \vdash^e)$ *is a simple constraint system,*
- *Var is an infinite set of variables,*
- *For each variable* $\alpha \in Var$, $\exists\alpha : p\Omega \rightarrow p\Omega$ *is an operation satisfying:*
  E1 $C \vdash^e \exists\alpha.C$
  E2 $C \vdash^e D \quad$ *implies* $\quad \exists\alpha.C \vdash^e \exists\alpha.D$
  E3 $\exists\alpha.(C \wedge \exists\alpha.D) =^e (\exists\alpha.C) \wedge (\exists\alpha.D)$
  E4 $\exists\alpha.\exists\beta.C =^e \exists\beta.\exists\alpha.C$

Remark. For simplicity, we omit set notation for constraints, and connect constraints by $\wedge$ instead of the union operator $\cup$. Also, we generally do not enclose simple constraints P in opening and closing braces. For instance, $P \wedge Q$ is an abbreviation for $\{P\} \cup \{Q\}$. We assume that $\wedge$ binds tighter than $\exists\bar{\alpha}$. For instance, $\exists\bar{\alpha}.C \wedge D$ stands for $\exists\bar{\alpha}.(C \wedge D)$. We write $C =^e D$ iff $C \vdash^e D$ and $D \vdash^e C$.

Example 6. Let the token set Ω *consist of some subclass of* $(\mathcal{L}, Var)$ *formulas closed under existential quantification of finite conjunctions. Each operator* $\exists\bar{\alpha}$ *is then interpreted by the function which maps each finite set* $\{P_1, \ldots, P_n\}$ *of tokens to the set of tokens* $\{\exists\bar{\alpha}.P_1 \wedge \ldots \wedge P_n\}$. *It is easy to see that the four conditions above are satisfied.*

The projection operator $\exists\bar{\alpha}$ allows us to bind variables $\bar{\alpha}$ in a constraint. That means we can project away information. If the constraint system models a boolean algebra, projection corresponds to existential quantification. Based on the projection operator we define the free variables $fv(C)$ and satisfiability of a constraint C.

Definition. *Let* C *be a constraint. Then* $fv(C) = \{\alpha \,|\, \exists\alpha.C \not=^e C\}$.

Definition. *Let* C *be a constraint. Then* C *is* satisfiable *iff* $\vdash^e \exists fv(C).C$.

The next lemma states an important property about the projection operator. Projection of a constraint does not influence the satisfiability of the constraint.

Lemma 1. *Let* C *be a constraint. Then* C *is satisfiable iff* $\exists\alpha.C$ *is satisfiable.*

The final step in our modeling of constraint systems is the extension from cylindric constraint systems to term constraint systems. We assume a term algebra $\mathcal{T}$ with signature $\Sigma = (Var, Cons)$ as given. *Var* is a set of variables and *Cons* is a set of type constructors containing at least the function constructor $\rightarrow$ of arity 2. In examples below we will sometimes use a multi-sorted algebra, in which terms and constructors are partitioned into sorts. Always present will be the sort of *types* which is ranged over by τ.

Definition. *A* substitution ϕ *is an idempotent mapping from the set of variables Var to the term algebra* $Term(\Sigma)$ *which is the identity everywhere except on a finite set of variables.*

Definition. *A* term constraint system $\mathcal{TCS}_{\mathcal{T}} = (\Omega, \vdash^e, Var, \{\exists\alpha \,|\, \alpha \in Var\})$ *over a term algebra* $\mathcal{T}$ *is a cylindric constraint system with predicates of the form*

$$p(\tau_1, \ldots, \tau_n) \qquad (\tau_i \in \mathcal{T})$$

such that the following holds:

- *For each pair of types* τ, τ' *there is an equality predicate* $(\tau = \tau')$ *in* $\mathcal{TCS}_{\mathcal{T}}$, *which satisfies:*
  D1 $\vdash^e (\alpha = \alpha)$
  D2 $(\alpha = \beta) \vdash^e (\beta = \alpha)$
  D3 $(\alpha = \beta) \wedge (\beta = \gamma) \vdash^e (\alpha = \gamma)$
  D4 $(\alpha = \beta) \wedge \exists\alpha.(C \wedge (\alpha = \beta)) \vdash^e C$
  D5 $(\tau = \tau') \vdash^e (T[\tau] = T[\tau'])$
  where $T[]$ *is an arbitrary term context*
- *For each predicate* P,
  D6 $[\tau/\alpha]P =^e \exists\alpha.(P \wedge (\alpha = \tau))$
  where $\alpha \notin fv(\tau)$

Remark. Conditions **D1** – **D4** are the conditions imposed on a cylindric constraint system with diagonal elements, which is usually taken as the foundation of constraint programming languages. **D4** says that equals can be substituted for equals; it is in effect the Leibniz principle. **D5** states that $(=)$ is a congruence. **D6** connects the syntactic operation of a substitution over predicates with the semantic concepts of projection and equality. Substitution is extended to arbitrary constraints in the canonical way:

$$[\tau/\alpha](P_1 \wedge \ldots \wedge P_n) = [\tau/\alpha]P_1 \wedge \ldots \wedge [\tau/\alpha]P_n.$$

Here are some basic lemmas which hold in term constraint systems.

Lemma 2 Renaming. *Let* C *be a constraint and* β *a new type variable. Then* $\exists\alpha.C =^e \exists\beta.[\beta/\alpha]C$.

Lemma 3 Normal Form. *Let* C *be a constraint and* $\phi = [\bar{\tau}/\bar{\alpha}]$ *be a substitution. Then* $\phi C =^e \exists\bar{\alpha}.C \wedge (\alpha_1 = \tau_1) \wedge \ldots \wedge (\alpha_n = \tau_n)$.

In the above lemma it is essential that substitutions are idempotent mappings. In the case of substitution ϕ this ensures that none of the type variables $\bar{\alpha}$ appears in the types $\bar{\tau}$.

Lemma 4 Substitution. *Let* C, D *be constraints such that* $C \vdash^e D$ *and* ϕ *be a substitution. Then* $\phi C \vdash^e \phi D$.

We now discuss several instances of term constraint systems. Section 7 will present a more elaborate example of a term constraint system that deals with records.

Example 7. For any term algebra $\mathcal{T}$ let HERBRAND $=$ $(\Omega, \vdash^e, \mathit{Var}, \{\exists\alpha \mid \alpha \in \mathit{Var}\})$ *be the minimal term constraint system where Ω contains only primitive constraints of the form $(\tau = \tau')$ where τ and τ' are types from $\mathcal{T}$. Equality in* HERBRAND *is syntactic,* i.e. *$\mathcal{T}$ is a free algebra. Entailment between two constraints C and D can be checked by the matching algorithm. For example, $(f(x,y) = f(a,g(b,c)))$ must entail $(x = a)$ and $(y = g(b,c))$. Satisfiability can be checked by (first–order) unification.*

A more refined example of a term constraint system deals with physical dimension types in the style of Kennedy [Ken96]:

Example 8. Let $\mathcal{T}$ be the two-sorted term algebra consisting of dimensions *and* types.

$$\begin{array}{lll} \textbf{Dimensions} & d & ::\supseteq\ \alpha \mid i(d) \mid \mathit{prod}(d,d) \mid 1 \mid \mathbf{m} \mid \mathbf{s} \\ \textbf{Types} & \tau & ::=\ \alpha \mid \dim(d) \mid \tau \to \tau \end{array}$$

The dimension constructor $i(\cdot)$ corresponds to the inverse of a dimension and $\mathit{prod}(\cdot,\cdot)$ to the product of two dimensions. Dimension constants are 1 *for the unit measure,* **m** *for meters and* **s** *for seconds. There might be other dimension constructors besides the mentioned ones. A type is either a type variable, or a dimension, or a function type.* DIM *is then the term constraint system which obeys the following additional conditions, which specify that dimension types form an abelian group.*

$$\begin{array}{ll} \text{DIM}\mathbf{1} & \vdash^e\ (\mathit{prod}(\alpha,\beta) = \mathit{prod}(\beta,\alpha)) \\ \text{DIM}\mathbf{2} & \vdash^e\ (\mathit{prod}(\alpha,\mathit{prod}(\beta,\gamma)) = \mathit{prod}(\mathit{prod}(\alpha,\beta),\gamma)) \\ \text{DIM}\mathbf{3} & \vdash^e\ (\mathit{prod}(\alpha,1) = \alpha) \\ \text{DIM}4 & \vdash^e\ (\mathit{prod}(\alpha,i(\alpha)) = 1) \end{array}$$

As our final example, we consider an extension of a term constraint system with subtyping.

Example 9. A subtype constraint system *over a term algebra $\mathcal{T}$ is a term constraint system with a subtype predicate $(\tau <: \tau')$ for each pair of types τ and τ' which satisfies the following conditions.*

$$\textbf{SUB1}\quad (\alpha = \alpha') =^e (\alpha <: \alpha') \wedge (\alpha' <: \alpha)$$

$$\textbf{SUB2}\quad \frac{D \vdash^e (\alpha_1' <: \alpha_1) \quad D \vdash^e (\alpha_2 <: \alpha_2')}{D \vdash^e (\alpha_1 \to \alpha_2 <: \alpha_1' \to \alpha_2')}$$

$$\textbf{SUB3}\quad \frac{D \vdash^e (\alpha_1 <: \alpha_2) \quad D \vdash^e (\alpha_2 <: \alpha_3)}{D \vdash^e (\alpha_1 <: \alpha_3)}$$

Let $\mathcal{SC}$ be a subtype constraint system with primitive types Int *and* Float *and record types of the form $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$. Records are modeled by admitting constructors of the form*

$$l_{1\_} \ldots l_{n\_} : \tau_1 \to \ldots \to \tau_n \to \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$$

in the term algebra. We assume that record fields are ordered with respect to a given ordering relation on field labels. The additional types obey the following rules.

$$\textbf{SUB4}\quad \vdash^e (\mathsf{Int} <: \mathsf{Float})$$

$$\textbf{SUB5}\quad \vdash^e (\{l_1 : \tau_1, \ldots, l_n : \tau_n, \ldots\} <: \{l_1 : \tau_1, \ldots, l_n : \tau_n\})$$

$$\textbf{SUB6}\quad \frac{D \vdash^e (\tau_1 <: \tau_1') \ldots D \vdash^e (\tau_n <: \tau_n')}{D \vdash^e (\{l_1 : \tau_1, \ldots, l_n : \tau_n\} <: \{l_1 : \tau_1', \ldots, l_n : \tau_n'\})}$$

4. The HM(X) framework

This section describes a general extension HM(X) of the Hindley/Milner type system with a term constraint system X over a term algebra $\mathcal{T}$.

Our development is similar to the original presentation [DM82]. We work with the following syntactic domains.

$$\begin{array}{lll} \textbf{Values} & v & ::=\ x \mid \lambda x.e \\ \textbf{Expressions} & e & ::=\ v \mid e\,e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \\ \textbf{Types} & \tau & ::=\ \alpha \mid\ \tau \to \tau \mid T\bar{\tau} \\ \textbf{Type schemes} & \sigma & ::=\ \tau \mid \forall\alpha.C \Rightarrow \sigma \end{array}$$

We consider only one–sorted algebras here, but it is straightforward to extend the treatment to multi–sorted algebras. This formulation generalizes the one in [DM82] in two respects. First, types are now members of an arbitrary term algebra, hence there might be other constructors besides $\to$. In the above definition T stands for additional type constructors which vary depending on a specific HM(X) instance. We have already seen examples where T has been instantiated to dimension and record types. Second, type schemes $\forall\alpha.C \Rightarrow \sigma$ now include a constraint component C, which restricts the types that can be substituted for the type variable α. We require that the constraint C has to be satisfiable. On the other hand, the language of terms is exactly as in [DM82]. That is, we assume that any language constructs that make use of type constraints are expressible as predefined values, whose names and types are recorded in the initial type environment.

The typing rules of our system can be found in Figure 2. *Typing judgments* are of the form $C, \Gamma \vdash e : \sigma$ where C is a satisfiable constraint in X, Γ a type environment and σ a type scheme. A typing judgment is *valid* if it can be derived by application of the typing rules and its constraint component is satisfiable.

Quite often we restrict the set of constraints C that can appear in type schemes and on the left hand side of the turnstile to so called *solved forms.* The set of

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

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

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

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

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

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

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

FIG. 2. Logical type system

solved forms, denoted by $\mathcal{S}$, is always a subset of the satisfiable constraints in X.

The most interesting rules in Figure 2 are the (∀ Intro) rule and the (∀ Elim) rule. By rule (∀ Intro) we quantify some type variables. We often use vector notation for type variables in type schemes. The term $\forall \bar{\alpha}.D \Rightarrow \tau$ is an abbreviation for $\forall \alpha_1.\mathsf{true} \Rightarrow \ldots \forall \alpha_n.D \Rightarrow \tau$ and $\exists \bar{\alpha}.D$ is an abbreviation for $\exists \alpha_1. \ldots \exists \alpha_n.D$.

Unlike in standard treatments of Hindley/Milner style systems we also have a subsumption rule (Sub), which allows us to derive term e with type τ' if we can derive term e with type τ and type τ subsumes type τ'. The subsumption relation $\preceq$ is determined by the constraint system X, and is assumed to satisfy the standard axioms for a partial ordering plus the contra-variance rule:

$$\textbf{REFL} \qquad (\alpha = \alpha') \vdash^{e} (\alpha \preceq \alpha') \wedge (\alpha' \preceq \alpha)$$

$$\textbf{ASYM} \qquad (\alpha \preceq \alpha') \wedge (\alpha' \preceq \alpha) \vdash^{e} (\alpha = \alpha')$$

$$\textbf{TRANS} \qquad \frac{D \vdash^{e} (\alpha_1 \preceq \alpha_2) \quad D \vdash^{e} (\alpha_2 \preceq \alpha_3)}{D \vdash^{e} (\alpha_1 \preceq \alpha_3)}$$

$$\textbf{CONTRA} \qquad \frac{D \vdash^{e} (\alpha_1' \preceq \alpha_1) \quad D \vdash^{e} (\alpha_2 \preceq \alpha_2')}{D \vdash^{e} (\alpha_1 \to \alpha_2 \preceq \alpha_1' \to \alpha_2')}$$

Except for these conditions, the choice of $\preceq$ is arbitrary.

Example 10. The Hindley/Milner system is an instance of our type system framework. Take X *to be the Herbrand constraint system over the algebra of types τ. Take the set of solved forms to be the set consisting only of* true*, which is represented by the empty token set. Take $\preceq$ to be syntactic type equality. Then the only type schemes arising in proof trees of valid typing judgments are of the form $\forall \alpha.\{\} \Rightarrow \sigma$, which we equate with Hindley/Milner type schemes $\forall \alpha.\sigma$. The subsumption rule becomes the trivial tautology which states that a judgment can be derived if it can be derived. It is easy to convince oneself that a judgment $\Gamma \vdash e : \sigma$ is derivable in Hindley/Milner if and only if $\{\}, \Gamma \vdash e : \sigma$ is derivable in* HM(HERBRAND).

Example 11. Let X *be the constraint system* DIM*, let the set of solved forms be the set consisting only of* true*, and let subsumption $\preceq$ be the equality relation $=$ in* DIM*. Then Kennedy's system can be recovered simply by adding primitives to the initial type environment Γ_0 that deal with dimensions.* E.g. *we assume that*

$$\mathit{div} : \forall d_1, d_2.\, \dim(d_1) \to \dim(d_2) \to \dim(\mathit{prod}(d_1, i(d_2)))$$

is contained in Γ_0. Other basic connectives are treated analogously.

Example 12. Let X *be the subtype constraint system $\mathcal{SC}$ and let the subsumption relation $\preceq$ be equal to the subtyping relation $<:$. Let the set of solved forms $\mathcal{S}$ be all satisfiable constraints in $\mathcal{SC}$. For every record $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$ in a program we add a datatype* *constructor*

$$l_1\_\ldots\_l_n : \tau_1 \to \ldots \to \tau_n \to \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$$

and for every field label l we add a function

$$\_.l : \{l : \tau\} \to \tau$$

to the initial type environment Γ_0. The first corresponds to record creation, the second to record selection. Other basic primitive functions are defined analogously.

The resulting system is related to the subtyping approach of the Hopkins Object Group [EST95b]. The main difference is that we use logical rules for quantifier introduction and elimination where they use a syntactic approach where quantifier introduction is coupled with let *and quantifier elimination is coupled with variable use. Another important difference is that their system also includes recursive types. Recursive types are beyond the scope of this paper, so we cannot deal with their system in its full generality. We can however deal with either a variant of their system without recursive types, or with a system of recursive records that are given as instances of explicitly declared classes, similar to the datatype constructions in functional languages or the class and interface system of Java [GLS96].*

Further applications with non-trivial constraint systems include overloading [Jon92, Kae92, VHJW96, NP93, CHO92, OWW95, BM97], record calculi [Rém89, Wan89], and static program analysis techniques such as binding time analysis [DHM95]. As an extended example we will present in Section 7 a record calculus similar to Ohori's [Oho95].

5. Semantics

We give a type soundness theorem based on an ideal semantics [MPS86] for HM(X) type systems. We show that our type system is sound, provided the underlying constraint system is sound and the subsumption predicate ($\preceq$) satisfies a coherence property. We say a constraint system is *sound* if every satisfiable constraint has a monotype solution. *Coherence* of a constraint system means that if a type τ subsumes a type τ', then the denotation of τ in the ideal model is a subset of the denotation of τ'.

Definition. *A* monotype *is a type τ with $fv(\tau) = \emptyset$.*

We let μ range over monotypes.

Definition. *A constraint system* X *is* sound *if for all type variables α and constraints $C \in \mathcal{S}$, if $\vdash^e \exists\alpha.C$ then there is a monotype μ such that $\vdash^e \exists\alpha.(\alpha = \mu) \wedge C$.*

The soundness proof is based on an ideal semantics of types which is a direct extension of the semantics in [Mil78].

The meaning of a term is a value in the CPO $\mathcal{V}$, where $\mathcal{V}$ contains all continuous functions from $\mathcal{V}$ to $\mathcal{V}$ and an error element $\mathbf{W}$, usually pronounced "wrong". Depending on the concrete type system used, $\mathcal{V}$ might contain other elements as well. We require that the values of additional type constructors are representable in the CPO $\mathcal{V}$. Then $\mathcal{V}$ is the least solution of the equation

$$\mathcal{V} \;=\; \mathbf{W}_\perp + \mathcal{V} \to \mathcal{V} + \textstyle\sum_{k \in \mathcal{K}} (k\,\mathcal{V}_1 \ldots \mathcal{V}_{\mathrm{arity}(k)})_\perp$$

where $\mathcal{K}$ is the set of values of an additional type constructor T.

The meaning function on terms is the same as in the original semantics of Hindley/Milner terms. That is, we assume that any language constructs that make use of type constraints are expressible as predefined values, whose names and types are recorded in the initial type environment.

$$\begin{array}{lcl}
\llbracket x \rrbracket\eta & = & \eta(x) \\[1ex]
\llbracket \lambda u.e \rrbracket\eta & = & \lambda v.\llbracket e \rrbracket\eta[u := v] \\[1ex]
\llbracket e\,e' \rrbracket\eta & = & \textsf{if}\,\llbracket e \rrbracket\eta \in \mathcal{V} \to \mathcal{V} \wedge \llbracket e' \rrbracket\eta \neq \mathbf{W} \\
 & & \textsf{then}\,(\llbracket e \rrbracket\eta)\,(\llbracket e' \rrbracket\eta) \\
 & & \textsf{else}\,\mathbf{W} \\[2ex]
\llbracket \textsf{let}\,x = e\,\textsf{in}\,e' \rrbracket\eta & = & \textsf{if}\,\llbracket e \rrbracket\eta \neq \mathbf{W} \\
 & & \textsf{then}\,\llbracket e' \rrbracket\eta[x := \llbracket e \rrbracket\eta] \\
 & & \textsf{else}\,\mathbf{W}
\end{array}$$

We will show in the following that the meaning of a well-typed program is always different from "wrong".

As a first step, we give a meaning to types. Following [Mil78], we let types denote ideals, i.e. non–empty, downward-closed and limit-closed subsets of $\mathcal{V}$. The meaning function $\llbracket\cdot\rrbracket$ maps closed types and type schemes to ideals. On function types and type schemes it is defined as follows:

$$\begin{array}{l}
\llbracket \mu_1 \to \mu_2 \rrbracket \;= \\
\qquad \{f \in \mathcal{V} \to \mathcal{V} \,|\, v \in \llbracket \mu_1 \rrbracket \Rightarrow f\,v \in \llbracket \mu_2 \rrbracket\} \\
\llbracket T\,\mu_1 \ldots \mu_m \rrbracket \;= \\
\qquad \{\perp\} \cup \\
\qquad \bigcup\{k\,\llbracket \mu'_1 \rrbracket \ldots \llbracket \mu'_n \rrbracket \mid \\
\qquad\qquad \textsf{true}, \Gamma_0 \vdash\; k : \mu'_1 \to \ldots \to \mu'_n \to T\,\mu_1 \ldots \mu_m\} \\
\llbracket \forall\bar{\alpha}.C \Rightarrow \tau \rrbracket \;= \\
\qquad \bigcap\{\llbracket [\bar{\mu}/\bar{\alpha}]\tau \rrbracket \mid\; \vdash^e\; [\bar{\mu}/\bar{\alpha}]C\}
\end{array}$$

We are now in the position to define coherence of the subsumption predicate ($\preceq$).

Definition. *The constraint system* X *is* coherent *if for all monotypes μ and μ', if $\vdash^e\; (\mu \preceq \mu')$ then $\llbracket \mu \rrbracket \subseteq \llbracket \mu' \rrbracket$.*

Lemma 5. *Let σ be a closed type scheme. Then $\llbracket \sigma \rrbracket$ is an ideal.*

Proof. A straightforward induction on the structure of σ. ■

Furthermore, we conclude that in a sound constraint system the error element is not contained in a closed type scheme.

Lemma 6. *Given a sound constraint system* X *and a closed type scheme* σ. *Then* $\mathbf{W} \notin [\![\sigma]\!]$.

Proof. This is true for all monotypes μ. Consider now a type scheme $\sigma = (\forall \bar{\alpha}.C \Rightarrow \tau)$. Because σ is closed we get $\vdash^e \exists \bar{\alpha}.C$ (remember that all constraints that appear in the typing judgments of a derivation need to be least satisfiable). Also, $\mathcal{C}$ is sound, thus there is a monotype vector $\bar{\mu}$ such that $\vdash^e [\bar{\mu}/\bar{\alpha}]C$. Hence, the denotation of $[\![\sigma]\!]$ is not an empty intersection. $\mathbf{W}$ is not contained in the denotation of any monotype $[\bar{\mu}/\bar{\alpha}]\bar{\tau}$. Thus $\mathbf{W}$ is not contained in $[\![\sigma]\!]$. ■

Definition. *A variable environment* η models *a closed typing environment* Γ, *written* $\eta \models \Gamma$, *if for all* $x : \sigma \in \Gamma$, $\eta(x) \in [\![\sigma]\!]$.

Theorem 7 Type Soundness. *Let* $C, \Gamma \vdash e : \sigma$ *be a valid typing judgment in* HM(X), *where* X *is a sound and coherent constraint system. Let* ϕ *be a substitution such that* $\phi\Gamma$ *and* $\phi\sigma$ *are closed and such that* $\vdash^e \phi C$. *Let* η *be a variable environment such that* $\eta \models \phi\Gamma$. *Then*

$$\begin{array}{ll} (1) & \mathbf{W} \notin [\![\phi\sigma]\!] \\ (2) & [\![e]\!]\eta \in [\![\phi\sigma]\!] \end{array}$$

Proof. (1) follows immediately from Lemma 6. We prove now (2) by a structural induction on typing derivations. There are three interesting cases.

Case (Var) The last step of the derivation is:

$$C, \Gamma \vdash x : \sigma \qquad\qquad (x : \sigma \in \Gamma)$$

Therefore $x : \phi\sigma \in \phi\Gamma$. Since $\eta \models \phi\Gamma$, $[\![x]\!]\eta = \eta(x) \in [\![\phi\sigma]\!]$.

Case ($\forall$ Intro) The last step of the derivation is:

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

Let ϕ be such that $\phi\Gamma$ and $\phi(\forall \bar{\alpha}.D \Rightarrow \tau)$ are closed and such that $\vdash^e \phi(C \wedge \exists \bar{\alpha}.D)$. Furthermore, we assume there are no name clashes between ϕ and $\bar{\alpha}$. Let $\bar{\mu}$ be an arbitrary vector of monotypes such that

$$\vdash^e \exists \bar{\alpha}.((\bar{\alpha} = \bar{\mu}) \wedge \phi D)$$

Since $\mathcal{C}$ is sound there is at least one such vector $\bar{\mu}$. Let $\phi' = [\bar{\mu}/\bar{\alpha}] \circ \phi$. Then since $\bar{\alpha} \notin fv(C)$, $\phi'(C \wedge D) = \phi C \wedge \phi' D$, which expands to $\phi C \wedge \exists \bar{\alpha}.((\bar{\mu} = \bar{\alpha}) \wedge \phi D)$. By our assumption this constraint is valid. Furthermore, $\phi'\Gamma$ and $\phi'\tau$ are both closed. By the induction hypothesis, $[\![e]\!]\eta \in [\![\phi'\tau]\!]$. Since $\bar{\mu}$ was arbitrary such that $\vdash^e [\bar{\mu}/\bar{\alpha}](\phi D)$,

$$\begin{array}{rcl} [\![e]\!]\eta & \in & \bigcap\{[\![[\bar{\mu}/\bar{\alpha}](\phi\tau)]\!] \mid \;\vdash^e [\bar{\mu}/\bar{\alpha}](\phi D)\} \\ & = & [\![\phi(\forall \bar{\alpha}.D \Rightarrow \tau)]\!]. \end{array}$$

Case (Sub) The last step of the derivation is:

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

We know that there is a substitution ϕ such that $\phi\Gamma$ and $\phi\tau'$ are closed and such that $\vdash^e \phi C$. It follows that $\vdash^e (\phi\tau \preceq \phi\tau')$. It might be the case that $\phi\tau$ still contains some free variables. We can extend ϕ to a substitution ϕ' such that $\phi'\tau$ is closed. Because ϕ' is an extension of ϕ we get that $\phi'\Gamma$ is closed and $\vdash^e \phi' C$. Applying the induction hypothesis, we get that $[\![e]\!]\eta \in [\![\phi'\tau]\!]$. Because X is coherent we know that $[\![\phi'\tau]\!] \subseteq [\![\phi'\tau']\!]$. Because $\phi\tau'$ is a closed type and ϕ' extends ϕ we get that $[\![\phi'\tau']\!] = [\![\phi\tau']\!]$ and this yields $[\![e]\!]\eta \in [\![\phi\tau']\!]$. ■

The type soundness theorem can be simplified to top–level programs. As a corollary, we find Milner's slogan "well types programs do not go wrong" carries over to sound constraint extensions.

Corollary. *Let* X *be a sound and coherent constraint system. Let* $\mathsf{true}, \Gamma \vdash e : \sigma$ *be a valid closed typing judgment in* HM(X). *If* $\eta \models \Gamma$ *then* $[\![e]\!]\eta \neq \mathbf{W}$.

Proof. Immediate from (1) and (2) of Theorem 7. ■

We find that HM(HERBRAND), HM(DIM) and HM($\mathcal{SC}$) satisfy the requirements. Hence, these applications are sound with respect to the provided semantics.

6. Type inference

We now turn to the problem of type inference in HM(X) type systems. We follow the standard approach of translating a typing problem into a constraint problem. Then a typing problem is solvable if the constraint problem is solvable. The solution of a constraint problem is a constraint in solved form in $\mathcal{S}$. If no solution exists then the typing problem is not solvable. For instance, consider a function application $e_1 e_2$ where e_1 has inferred type τ_1 and e_2 has inferred type τ_2. To solve the typing problem $e_1 e_2$ we need to solve the constraint $(\tau_1 \preceq \tau_2 \rightarrow \alpha)$ with the fresh type variable α corresponding to the yet unspecified result type of the application $e_1 e_2$.

For the moment, we take a closer look at two specific typing situations. In HM($\mathcal{SC}$) the subsumption predicate $\preceq$ corresponds to the subtype predicate $<:$. The set $\mathcal{S}$ is defined as the set of all satisfiable constraints in $\mathcal{SC}$. Then solving a constraint problem means simply checking whether the constraint is satisfiable or not. In another example we considered the Hindley/Milner system as an instance HM(HERBRAND) of the HM(X) framework. Here, the subsumption predicate $\preceq$ corresponds to the type equality predicate $=$ and $\mathcal{S}$ is the set consisting of just true. In this case solving a constraint problem requires more than just a satisfiability test. We additionally have to discard all equality problems, which can be achieved by Herbrand unification.

We can observe that type inference consists of two phases: *constraint generation* and *constraint solving.* Constraint generation is always the same for all HM(X) type systems. We simply generate constraints of the form $(\tau \preceq \tau')$. But the kind of constraint solving might differ in different typing situations. Depending on the structure of the set $\mathcal{S}$ of solved forms we have to apply different methods to obtain a constraint in solved form. The least requirement which we put on $\mathcal{S}$ is that the constraints in $\mathcal{S}$ are satisfiable. Hence, solving of a constraint problem requires at least a satisfiability test. But our constraint systems and the structure of the set $\mathcal{S}$ can be arbitrary complex. Therefore, solving of constraint problems might involve more sophisticated methods than e.g. a satisfiability test or Herbrand unification. In the latter, we refer to solving of constraint problems as *constraint normalization* or *normalization* for short. In the next section we give a formal treatment of normalization in a constraint system X. Then, we give a generic type inference algorithm for HM(X) type systems and state our main results, namely that type inference is sound, and under sufficient conditions on X also complete.

6.1 Normalization

In this section we study normalization of constraints. Before giving an axiomatic description of normalization, we first introduce some preliminary definitions.

Preliminaries: Let $\phi_{|U}$ be the restriction of the substitution ϕ to the domain U. That is, $\phi_{|U}(x) = \phi(x)$ if $x \in U$ and $\phi_{|U}(x) = x$ otherwise. For substitutions ϕ and ψ we write $\psi =_U \phi$ iff $\vdash^e \ (\psi(x) = \phi(x))$ for all $x \in U$. We write $\psi \leq_U^{\phi'} \phi$ iff $\phi' \circ \psi =_U \phi$. We write $\psi \leq_U \phi$ if $\exists \phi' : \psi \leq^{\phi'} \phi$. Sometimes, we omit the set U.

Note that this makes the "more general" substitution the smaller element in the pre–order $\leq_U$. This choice, which reverses the usual convention in treatments of unification (e.g. [LMM87]), was made to stay in line with the semantic notion of type instances.

We make $\leq_U$ a partial order by identifying substitutions that are equal up to variable renaming, or equivalently, by defining $\psi =_U \phi$ iff $\psi \leq_U \phi$ and $\phi \leq_U \psi$. It follows from [LMM87] that $\leq_U$ is a complete lower semi–lattice where least upper bounds, if they exist, correspond to unifications and greatest lower bounds correspond to anti–unifications.

We consider now the task of normalization. Generally, a typing problem is translated into a constraint C in the term constraint system $\mathcal{C}$ and a substitution ψ. We will refer to the pair (C, ψ) as a *constraint problem.* Normalization means then computation of a *normal form* of a constraint problem (C, ψ).

Definition. *Let* X *be a term constraint system over a term algebra* $\mathcal{T}$ *and* $\mathcal{S}$ *be the set of solved constraints in* X. *Let* $C \in \mathcal{S}$ *and* $D \in$ X *be constraints and let* ϕ,ψ *be substitutions. Then* (C, ψ) *is a* normal form *of* (D, ϕ) *iff* $\phi \leq \psi$, $C \vdash^e \psi D$ *and* $\psi C = C$.

(C, ψ) *is* principal *if for all normal forms* (C', ψ') *of* (D, ϕ) *we have that* $\psi \leq \psi'$ *and* $C' \vdash^e \psi' C$.

The principal normal form represents the *best* solution of a constraint problem. As an example consider the constraint system HERBRAND. There, a principal normal form corresponds to a most general unifier and a normal form corresponds to a unifier of a constraint problem.

The next lemma states that all principal normal forms are unique up to variable renaming.

Lemma 8 Uniqueness. *Let* (C, ψ) *and* (C', ψ') *be principal normal forms of* (D, ϕ). *Then there is a variable renaming* ϕ' *such that* $C' =^e \phi' C$ *and* $\psi' = \phi' \circ \psi$.

We identify two normal forms that are equivalent up to variable renaming. We can thus define a well–defined function *normalize* from constraint problems (D, ϕ) to normal forms as follows:

$$\begin{array}{ll} \multicolumn{2}{l}{\mathit{normalize}(D, \phi)} \\ = (C, \psi) & \text{if } (C, \psi) \text{ principal normal form of } (D, \phi) \\ = \mathit{fail} & \text{otherwise} \end{array}$$

We now extend the property of having a principal normal form to constraint systems.

Definition. *Given a constraint system* X *over a term algebra* $\mathcal{T}$ *and a set of solved constraints* $\mathcal{S}$ *in* X. *The constraint system* X *has the* principal constraint property *if for every constraint* $D \in$ X *and substitution* ϕ, *either* (D, ϕ) *does not have a normal form or* (D, ϕ) *has a principal normal form.*

We also say that the HM(X) type system has the principal constraint property if X has the principal constraint property.

In Section 7 we discuss in detail a type system for Ohori-style records that satisfies the principal constraint property. This example belongs to a class of constraint systems where constraint solving involves some form of unification. Further examples of constraint systems of this kind are HERBRAND and DIM. We can apply similar techniques as those introduced in Section 7 to show that HERBRAND and DIM satisfy the principal constraint property.

The situation is different for the constraint system $\mathcal{SC}$. There, the set $\mathcal{S}$ of solved forms consists of

$$\text{(Var)} \qquad \frac{\begin{array}{c} x : (\forall\bar{\alpha}.D \Rightarrow \tau) \in \Gamma \qquad \bar{\beta} \text{ new} \\ (C,\psi) = \mathit{normalize}(D, [\bar{\beta}/\bar{\alpha}]) \end{array}}{\psi_{|fv(\Gamma)}, C, \Gamma \vdash^{W} x : \psi\tau}$$

$$\text{(Abs)} \qquad \frac{\psi, C, \Gamma_x.x : \alpha \vdash^{W} e : \tau \qquad \alpha \text{ new}}{\psi_{\backslash\{\alpha\}}, C, \Gamma_x \vdash^{W} \lambda x.e : \psi(\alpha) \to \tau}$$

$$\text{(App)} \qquad \frac{\begin{array}{c} \psi_1, C_1, \Gamma \vdash^{W} e_1 : \tau_1 \qquad \psi_2, C_2, \Gamma \vdash^{W} e_2 : \tau_2 \\ \psi' = \psi_1 \sqcup \psi_2 \\ D = C_1 \wedge C_2 \wedge (\tau_1 \preceq \tau_2 \to \alpha) \qquad \alpha \text{ new} \\ (C,\psi) = \mathit{normalize}(D, \psi') \end{array}}{\psi_{|fv(\Gamma)}, C, \Gamma \vdash^{W} e_1 e_2 : \psi(\alpha)}$$

$$\text{(Let)} \qquad \frac{\begin{array}{c} \psi_1, C_1, \Gamma_x \vdash^{W} e : \tau \qquad (C_2, \sigma) = \mathit{gen}(C_1, \psi_1\Gamma, \tau) \\ \psi_2, C_3, \Gamma_x.x : \sigma \vdash^{W} e' : \tau' \\ \psi' = \psi_1 \sqcup \psi_2 \qquad D = C_2 \wedge C_3 \\ (C,\psi) = \mathit{normalize}(D, \psi') \end{array}}{\psi_{|fv(\Gamma_x)}, C, \Gamma_x \vdash^{W} \mathsf{let}\ x = e\ \mathsf{in}\ e' : \psi\tau'}$$

FIG. 3. Type inference

all satisfiable constraints. Given a constraint problem (D,ϕ) we distinguish between two cases. If ϕD is unsatisfiable then (D,ϕ) does not have a normal form. Assume ϕD is satisfiable then $(\phi D, id)$ is the principal normal form of (D,ϕ). Given another normal form (D',ϕ') of (D,ϕ). Then it holds that $\phi \leq \phi'$ and $D' \vdash^{e} \phi' D$. But then it follows immediately that $(\phi D, id)$ is principal. We conclude that the constraint system $\mathcal{SC}$ satisfies the principal constraint property, and that a *normalize* function can be defined as follows:

$$\begin{array}{ll} \multicolumn{2}{l}{\mathit{normalize}(C,\phi)} \\ = (\phi C, id) & \text{if } \phi C \text{ is satisfiable} \\ = \mathit{fail} & \text{otherwise} \end{array}$$

The normalization function is computable since satisfiability in $\mathcal{SC}$ is decidable. This follows easily by adapting techniques developed in [TS96].

6.2 Type inference algorithm

We now connect the principal constraint property of a constraint system with the principal types property of a type system. Figure 3 gives a generic type inference algorithm that computes principal types if the constraint system satisfies the principal constraint property. The algorithm is formulated as a deduction system over clauses of the form $\psi, C, \Gamma \vdash^{W} e : \tau$ with type environment Γ, expression e as input values and substitution ψ, constraint C, type τ as output values. For each syntactic construct of expressions e we have one clause. The deduction rules can be interpreted operationally, as a logic program that constructs a bottom–up derivation of $\vdash^{W}$ clauses.

In the (Var) rule, we assume that an unqualified type τ can be represented as $\forall\emptyset.\mathsf{true} \Rightarrow \tau$. This avoids a separate case of this rule for unqualified types. Note that (Var) makes use of the function *normalize*, specified in the last subsection. Our deduction rules yield an algorithm only if *normalize* is computable. In the following, we assume that we are dealing only with computable normalization functions.

The type inference algorithm $\vdash^{W}$ is a straightforward extension of algorithm *W*, see [DM82]. The algorithm $\vdash^{W}$ consists of the following three basic components: constraint generation, constraint normalization and generalization of unbound type variables. All three components can already be found in the original algorithm *W* but are now extended to deal with constraints. We already discussed constraint generation and normalization. The generalization procedure for our algorithm is left underspecified; we only require that it satisfies:

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

where C is a constraint such that $C =^{e} C' \wedge D$, Γ is a type environment, σ is a type scheme, $\bar{\alpha} = (\mathit{fv}(\sigma) \cup \mathit{fv}(C)) \backslash \mathit{fv}(\Gamma)$ and $\mathit{fv}(D) \cap \bar{\alpha} = \emptyset$. That is, generalization splits a constraint into two parts. Generalized variables can be free only in one of the two parts, C', but not the other, D. Only the C' part ends up as a constraint in the generalized type scheme. Note that the above requirement can always be fulfilled by taking D to be true. However, depending on the actual constraint system used there might exist better strategies, which keep the constraint in the generalized type scheme smaller.

Our type inference algorithm interleaves constraint generation and normalization. Each inference rule combines the constraint problems of the premises and performs then a normalization step. That means we perform *strict* normalization during type inference. In essence, we only need to perform normalization right before a (Let) rule (because the constraint in a type scheme needs to be in normal form) or at the end. This corresponds to *lazy* normalization. An example of a lazy formulation of type inference for the Hindley/Milner type system can already be found in [Wan87]. The following lemma states that both views are equivalent. We can perform normalization in any order and always obtain the same result.

Lemma 9. *Given constraints D, D' and substitutions ϕ, ϕ'. Then*

$$\begin{array}{c} \mathit{normalize}((D,\phi) \sqcup (D',\phi')) \\ = \\ \mathit{normalize}(\mathit{normalize}(D,\phi) \sqcup \mathit{normalize}(D',\phi')) \end{array}$$

where the term $(D,\phi) \sqcup (D',\phi')$ stands for $(D \wedge D', \phi \sqcup \phi')$.

6.3 Main results

To state our main results concisely, we extend the subsumption predicate $\preceq$ to type schemes. Subsumption on type schemes is defined by a deduction system with clauses of the form $C \vdash^i \sigma \preceq \sigma'$, which state that the type scheme σ is more general than the type scheme σ' under the constraint C. The deduction system is defined as follows.

$$\text{(Sub)} \qquad \frac{C \vdash^e (\tau \preceq \tau')}{C \vdash^i \tau \preceq \tau'}$$

$$(\preceq \forall) \quad \frac{C \wedge D \vdash^i \sigma \preceq \sigma' \qquad \alpha \notin tv(\sigma) \cup tv(C)}{C \vdash^i \sigma \preceq (\forall \alpha. D \Rightarrow \sigma')}$$

$$(\forall \preceq) \quad \frac{C \vdash^i [\tau/\alpha]\sigma \preceq \sigma' \qquad C \vdash^e [\tau/\alpha]D}{C \vdash^i (\forall \alpha. D \Rightarrow \sigma) \preceq \sigma'}$$

The result triple of the type inference algorithm $\vdash^W$ forms a *typing configuration* (C, σ, ψ), which consists of a constraint $C \in \mathcal{S}$, a type scheme σ and a substitution ψ such that $\psi C = C$, $\psi\sigma = \sigma$ and ψ is consistent with respect to Γ. A substitution ϕ is *consistent* with respect to a type scheme $\sigma = \forall \bar{\alpha}. D \Rightarrow \tau$ if $\psi D \in \mathcal{S}$ where we assume there are no name clashes between $\bar{\alpha}$ and ψ. This extends naturally to type environments. Given two typing configurations (C, σ, ψ), (C', σ', ψ') we say (C, σ, ψ) is *more general* than (C', σ', ψ') iff $\psi \leq^{\phi'}_{fv(\Gamma)} \psi$, $C' \vdash^e \phi' C$ and $C' \vdash^i \phi'\sigma \preceq \sigma'$. In such a situation we write $(C, \sigma, \psi) \preceq (C', \sigma', \psi)$.

Lemma 10. *Given a type environment Γ and a term e. If $\psi, C, \Gamma \vdash^W e : \tau$ then (C, τ, ψ) is a typing configuration.*

Furthermore, this typing configuration always represents a valid typing of the given term under the given type environment.

Theorem 11 (Soundness of Inference). *Given a term e and a type environment Γ. If $\psi, C, \Gamma \vdash^W e : \tau$ then $C, \psi\Gamma \vdash e : \tau$, $\psi C = C$ and $\psi\tau = \tau$.*

A sketch of the proofs of soundness and completeness of type inference can be found in the appendix. For a more detailed discussion we refer to [Sul97].

We now discuss completeness of type inference for HM(X) type systems. In general, we always require that an HM(X) type system has to fulfill the principal constraint property to achieve complete type inference. But as it turns out this is not sufficient. There are examples of *non–regular* equational theories where unification is unitary (that means we have most general unifiers) but algorithm $\vdash^W$ does not infer principal types. An equational theory is *regular* if $\vdash^e (\tau = \tau')$ implies $fv(\tau) = fv(\tau')$. We say a constraint system X is regular if the underlying equational theory is regular. An example of a non–regular theory is the dimension constraint system DIM. We find that $\vdash^e (\mathit{prod}(i(d), d) = 1)$ but $fv(\mathit{prod}(i(d), d)) = \{d\} \neq \emptyset = fv(1)$. In Section 6.1 we observed that DIM satisfies the principal constraint property. But algoritm $\vdash^W$ fails to infer principal types for the dimension type system HM(DIM). This observation is due to A.J. Kennedy. At the end of this section we give a concrete example where we can see why algorithm $\vdash^W$ fails.

Nevertheless, we can state a completeness theorem for two large classes of HM(X) type systems. First, we consider the class of constraint systems X where the set $\mathcal{S}$ of solved forms in X contains all satisfiable constraints in X. We denote by $\mathcal{X}^a$ the set of all those constraint systems that additionally satisfy the principal constraint property. In the second class we put further restrictions on the set $\mathcal{S}$ of solved forms. We assume that all constraints in $\mathcal{S}$ are in *simplified* form, which means that all non–trivial equality problems have been resolved. A constraint $C \in \mathcal{S}$ is in *simplified* form if $C \vdash^e (\tau = \tau')$ implies $\vdash^e (\tau = \tau')$. We denote by $\mathcal{X}^r$ the set of all regular constraint systems X which satisfy the principal constraint property and for which every solved form is also a simplified form.

An example for a member of $\mathcal{X}^a$ is the constraint system $\mathcal{SC}$. The constraint systems HERBRAND and the record constraint system introduced in Section 7 are examples for members of $\mathcal{X}^r$. But DIM is not in $\mathcal{X}^r$ because DIM is non–regular.

To obtain a completeness result for type inference, we assume that we have an HM(X) type system where X belongs to $\mathcal{X}^a$ or $\mathcal{X}^r$. Furthermore, we consider only 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 $C \vdash^i \sigma \preceq \tau$.

Now, we present our completeness result. Informally speaking, we want to have the following. Given a derivation $C', \phi\Gamma \vdash e : \sigma'$, our type inference algorithm should report a constraint that is at least as small as C' and a type that is at least as general as σ'.

Theorem 12 (Completeness of Inference). *Let $C', \phi\Gamma \vdash e : \sigma'$ be a typing judgment such that $\phi\Gamma$ is realizable in C'. Then*

$$\psi, C, \Gamma \vdash^W e : \tau$$

for some substitution ψ, constraint C, type τ, such that

$$\begin{array}{rcl} gen(C, \psi\Gamma, \tau) & = & (C_o, \sigma_o) \\ (C_o, \sigma_o, \psi) & \preceq & (C', \sigma', \phi) \end{array}$$

The completeness theorem can be simplified for top–level programs to the following corollary, which states that our type inference algorithm computes principal types.

Corollary. *Let* $\mathsf{true}, \Gamma \vdash e : \sigma$ *be a closed typing judgment such that* Γ *is realizable in* true. *Then* $\phi, C, \Gamma \vdash^W e : \tau$ *for some substitution* ϕ, *constraint* C, *such that*

$$\begin{array}{rcl} gen(C, \phi\Gamma, \tau) & = & (\mathsf{true}, \sigma_o) \\ \vdash^i \sigma_o & \preceq & \sigma \end{array}$$

In the case of HM(X) type systems where X in $\mathcal{X}^a$ we have formulated the completeness result in more general terms than actually necessary. In Section 6.1 we observed that normalization in $\mathcal{SC}$ corresponds to a satisfiability test. This observation can be generalized to all constraint systems in the class $\mathcal{X}^a$. But then we can conclude that type inference always returns the identity substitution. Type inference only consists in accumulating constraints and checking whether the constraints are satisfiable or not. This holds for the (Var) case. We rename the bound type variables in the constraint and check satisfiability of the renamed constraint. If this constraint is satisfiable we return the renamed constraint. The renaming substitution is equivalent to the identity substitution on the free type variables of the given type environment. We find that no substitutions are introduced in the base case nor through the normalization procedure. Then type inference in $\mathcal{X}^a$ always returns the identity substitution. Hence, substitution ψ is always the identity substitution in the completeness theorem for the class $\mathcal{X}^a$.

In case of HM(X) type systems where X in $\mathcal{X}^r$ we have put stronger conditions on the set $\mathcal{S}$ of solved constraints. The set $\mathcal{S}$ must now be in simplified form. Therefore, normalization also involves computation of a residual substitution. The restriction to regular theories in case of the class $\mathcal{X}^r$ is important to establish complete type inference as we will see in the following example, due to A.J. Kennedy [Ken96].

In the dimension type system HM(DIM), define an initial type environment as follows:

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

Here, *kg* and *s* are some basic dimensions, *pair* is the pairing operator and *div* is a primitive operation on dimensions. Now 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 $\Gamma.x : \dim prod(d_1, d_2)$:

$$\Gamma.x : \dim prod(d_1, d_2) \vdash div\, x : \dim d_1 \rightarrow \dim d_2$$

Here, it is not possible to quantify over the type variables d_1 and d_2. But we can derive another type for $div\, x$ under the same type environment:

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

We have simply instantiated d_1 with $prod(d_1, d_3)$ and d_2 with $prod(i(d_3), d_2)$. Kennedy calls this the problem of *unrevealed* polymorphism. Neither of the two types for $div\, x$ is more general than the other, and there is no third type that generalizes both. Hence, algorithm $\vdash^W$ fails to infer a principal type for expression e under type environment Γ.

It is interesting to point out that $\vdash^W$ computes principal types for dimension types if $\mathcal{S}$ contains all satisfiable constraints in DIM. Then DIM belongs to $\mathcal{X}^a$ and for that class we have a completeness result. The reason is that now all unification problems are explicit. No unification is involved during type inference. Type inference performs only a satisfiability test. The problem of unrevealed polymorphism comes into play if normalization involves unification in a non–regular theory.

7. Polymorphic records

Following ideas of Ohori [Oho95] we give an instance of our HM(X) system which deals with polymorphic records. Ohori's system, abbreviated $\mathcal{O}$ in the following, has besides type variables and function types also record types denoted by $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$, where l_i is an element of an enumerable set of record labels. We assume that there is an ordering relation between all field labels. All record fields are ordered with respect to this ordering relation. Because we have a fixed ordering of record fields we can apply Herbrand unification for solving equality constraints between records.

Type quantification in $\mathcal{O}$ is kinded; in the type scheme $\forall\alpha.\alpha :: \kappa \Rightarrow \sigma$ the type variable α ranges only over kind κ. A kind is of the form $\langle l_1 : \tau_1, \ldots, l_n : \tau_n\rangle$; it comprises all records that contain at least fields $l_1, \ldots, l_n$ with types $\tau_1, \ldots, \tau_n$.

Instead of a constraint on the left hand side of a typing judgment, Ohori uses a *kind assignment* $\mathcal{K}$ which can be considered as a function which assigns each type variable α its kind k. He writes $\mathcal{K} \wedge (\alpha :: k)$ for the disjoint extension of $\mathcal{K}$ with a new type variable α with kind k.

Here's an example of a program typed in $\mathcal{O}$.

Example 13.

```
f: ∀α, β.(α :: ⟨l : β⟩) ⇒ α → Int
f x =
   let g: β → Bool
       g = λ y. eq y (x.l)
   in 1
```

We use a Haskell-style notation, with type scheme annotations added for illustration purposes. The program assumes that there is a function

$$\mathrm{eq} : \forall\alpha.\alpha \to \alpha \to \mathsf{Bool}$$

in the initial type environment.

7.1 Type system

We now translate $\mathcal{O}$ into the HM(X) framework. We add to the initial type environment Γ_0 primitive constructs that deal with record formation, selection and update. For every ordered sequence of record labels $l_1, \ldots, l_n$ we postulate an n-ary parameterized data type $R_{l_1\_\ldots\_l_n}$. The record type $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$ is then represented as $R_{l_1\_\ldots\_l_n}\tau_1 \ldots \tau_n$. For simplicity we will keep using the record type notation as a synonym for the datatype notation. For every record datatype $R_{l_1\_\ldots\_l_n}$ we have in the initial environment a datatype constructor

$$l_1\_\ldots\_l_n : \tau_1 \to \ldots \to R_{l_1\_\ldots\_l_n}\tau_1 \ldots \tau_n$$

Then, $l_1\_\ldots\_l_n\, e_1 \ldots e_n$ represents record formation $\{l_1 = e_1, \ldots, l_n = e_n\}$. For each field label l we add to the initial type environment Γ_0 the two functions

$$\begin{array}{r} \_.l : \forall\alpha,\beta.(\alpha :: \langle l : \beta\rangle) \Rightarrow \alpha \to \beta \\ \mathit{modify}_l : \forall\alpha,\beta.(\alpha :: \langle l : \beta\rangle) \Rightarrow \alpha \to \beta \to \alpha \end{array}$$

The first function corresponds to record selection, the second to record update.

Kinded quantification in $\mathcal{O}$ is modeled by primitive constraints of the form $(\tau :: k)$ where τ is a type and k is a kind. Technically, this means we add $(\tau :: k)$ to the set Ω of primitive constraints where $(::)$ is a primitive predicate of arity 2. We define REC as the smallest term constraint system that satisfies the following additional rules:

$$\begin{array}{ll} \textbf{REC1} & \vdash^e (\{l_1 : \tau_1, \ldots l_n : \tau_n\} :: \langle l_i : \tau_i\rangle) \\ & \text{where } l_1, \ldots, l_n \text{ are distinct} \\ \textbf{REC2} & (\tau :: \langle l : \tau_1\rangle) \wedge (\tau :: \langle l : \tau_2\rangle) \vdash^e (\tau_1 = \tau_2) \\ \textbf{REC3} & (\{\ldots, l : \tau_1, \ldots\} :: \langle l : \tau_2\rangle) \vdash^e (\tau_1 = \tau_2) \\ \textbf{REC4} & \exists\alpha.(\alpha :: k) =^e \mathsf{true} \\ & \text{where } \alpha \notin \mathit{fv}(k) \end{array}$$

Note that these conditions rule out recursive records, since our type algebra does not have recursive types. On the other hand, we do allow recursive constraints between type variables in REC. For instance, the constraint $(\alpha :: \langle l : \alpha \to \alpha\rangle)$ is well-formed. But that constraint is not satisfiable and therefore cannot appear as a solved form. Also ruled out (by conditions **REC2** and **REC3**) is overloading of field labels.

The set $\mathcal{S}$ of solved forms in HM(REC) consists of all satisfiable constraints of the form

$$C \;::=\; \{\} \mid (\alpha :: \langle l : \tau\rangle) \mid C \wedge C \mid \exists\bar{\alpha}.C$$

where we take the empty token set as a representation of true. Furthermore, we require that the constraints in $\mathcal{S}$ are in simplified form, i.e. $C \vdash^e (\tau = \tau')$ must imply $\vdash^e (\tau = \tau')$. For instance,

$$(\alpha :: \langle l : \beta\rangle) \wedge (\alpha :: \langle l : \gamma \to \gamma\rangle)$$

is not in simplified form and is therefore excluded.

The type system HM(REC) is as given in Figure 2, with subsumption $(\preceq)$ being modeled by $(=)$. As an example, here the annotated program from Example 7 re-formulated in HM(REC):

Example 14.

```
f: ∀α.(∃β.(α :: ⟨l : β⟩)) ⇒ α → Int
f x =
  let g : ∀β.(α :: ⟨l : β⟩) ⇒
          β → Bool
      g = λ y. eq y (x.l)
  in 1
```

In HM(REC) we quantify in the innermost let over type variable β, leaving just α to be quantified in the toplevel function f. This is not possible in $\mathcal{O}$, since α's kind depends on β. The question arises whether this makes HM(REC) a more permissive type system than $\mathcal{O}$. Specifically, are there examples where we can use function g polymorphically? The answer is no. Every instance of g has to satisfy the constraint

$\exists\beta.(\alpha :: \langle l_1 : \beta\rangle)$. But α can only have one field entry with label l_1. Therefore, we can use g in the let-body only monomorphically. In general, we can observe that $\mathcal{O}$ and HM(REC) type exactly the same programs, but the types are more precise in HM(REC).

Theorem 13 Full and Faithful. *Every program typable in $\mathcal{O}$ is typable in* HM(REC) *and vice versa.*

7.2 Type inference

We now consider type inference for HM(REC). Since REC is a regular constraint system, we can obtain type inference with principal types, provided it fulfills the principal constraint property. To show the principal constraint property for REC, we proceed in three steps. First, we show that it is always possible to formulate a constraint as a projection over a projection–free subpart. A constraint D is *projection–free* if D (considered as a set) contains only tokens of the form $(\alpha :: k)$ and $(\tau = \tau')$. Then we give a procedure which computes the principal normal form of projection–free constraints, or fails if no normal form exists. Finally, we show that it is sufficient to compute principal normal forms of projection–free constraints. This is achieved by a lifting method. Given an arbitrary constraint C we compute the principal normal form of the projection–free part. Then we lift this result to the projected part. We show that this lifting method is sound and complete.

In a first step we transform a constraint into a projection over a projection–free subpart. The idea is that we can always rename type variables which are bound by the projection operator. It holds that

$$\exists\alpha.C =^e \exists\beta.[\beta/\alpha]C$$

where β is a new type variable. That means, w.l.o.g. there are no name clashes between two projected constraints $(\exists\alpha.C)\wedge(\exists\beta.D)$. Then we can lift all projection operators to the outermost level using condition **E3** of a cylindric constraint system:

$$(\exists\alpha.C) \wedge (\exists\beta.D) =^e \exists\alpha.(\exists\beta.(C \wedge D))$$

We can summarize these observations in the following lemma.

Lemma 14. *Let $C \in$ REC. Then there exists a projection–free constraint D such that $C =^e \exists\bar{\alpha}.D$.*

In the next step we show how to compute principal normal forms for projection–free constraints. We assume that we have a projection–free constraint D which contains only primitive predicates of the form $(=)$ and $(::)$. W.l.o.g., we can assume that all predicates $(::)$ are of the form $(\alpha :: k)$. This can be achieved because we know that

$$(\tau :: k) =^e \exists\alpha.((\alpha = \tau) \wedge (\alpha :: k))$$

where α is a new type variable. The closure $\mathrm{Cl}(D)$ of D is the smallest constraint which fulfills the following conditions:

1. $D \subseteq \mathrm{Cl}(D)$
2. If $(\alpha = \{l_1 : \tau_1, \ldots, l_n : \tau_n\}) \in \mathrm{Cl}(D)$
   then $(\alpha :: \langle l_1 : \tau_1\rangle), \ldots, (\alpha :: \langle l_n : \tau_n\rangle) \in \mathrm{Cl}(D)$
3. If $(\alpha :: \langle l : \tau_1\rangle), (\alpha :: \langle l : \tau_2\rangle) \in \mathrm{Cl}(D)$
   then $(\tau_1 = \tau_2) \in \mathrm{Cl}(D)$

From a semantic view point we have not done anything because $\mathrm{Cl}(D) =^e D$. We only have changed the syntactic representation of D. The intention of building the closure of D is to generate all predicates $(\tau :: \langle l : \tau'\rangle)$ which might cause any inconsistencies. Given all such predicates we can generate all unification problems $(\tau = \tau')$ which have to be resolved. The following lemma states that we really have generated all such predicates.

Lemma 15. *Given a field label l and types τ, τ'. If $\not\vdash^e (\tau :: \langle l : \tau'\rangle)$ then $(\tau :: \langle l : \tau'\rangle) \in Cl(D)$ iff $D \vdash^e (\tau :: \langle l : \tau'\rangle)$. Furthermore, if $\not\vdash^e (\tau = \tau')$ then $(\tau = \tau') \in Cl(D)$ iff $D \vdash^e (\tau = \tau')$.*

We can apply unification over Herbrand terms [Rob65] to resolve all equality predicates $(=)$ in $\mathrm{Cl}(D)$. We obtain a most general unifier ϕ of the equality predicates $(=)$ in $\mathrm{Cl}(D)$. It remains to check whether this most general unifier ϕ is consistent with $\mathrm{Cl}(D)$. This can be done by checking whether there are any inconsistencies in $\phi\mathrm{Cl}(D)$. If not, $(\phi\mathrm{Cl}(D), \phi)$ represents the principal normal form of (D, id). We can summarize this observation in the following lemma.

Lemma 16. *Given a projection–free constraint $D \in$ REC and a substitution ϕ. Then (D, ϕ) has a principal normal form, which can be computed by the procedure described above, or else no normal form exists.*

It remains to lift this procedure to arbitrary constraints. First, we state some essential lemmas that are necessary to establish this lifting method. Then we apply this lifting method to state that REC satisfies the principal constraint property.

The next lemma gives us a procedure to lift principal normal forms of constraints to arbitrary constraints. It states that whenever we can compute the principal normal form of a constraint D then we get the principal normal form of the constraint $\exists\alpha.D$ for free.

Lemma 17. *Let $D \in$ REC and ϕ be a substitution where $\alpha \notin codom(\phi) \cup dom(\phi)$. If $(C, \psi) = normalize(D, \phi)$ then $(\exists\alpha.C, \psi_{\backslash\{\alpha\}}) = normalize(\exists\alpha.D, \phi)$.*

The next lemma states that a normal form of a constraint exists iff a normal form of the projected constraint exists.

Lemma 18. *Given a substitution ϕ where $\alpha \notin codom(\phi) \cup dom(\phi)$ and a constraint $D \in$ REC. Then (D, ϕ) has a normal form iff $(\exists\alpha.D, \phi)$ has a normal form.*

We have now everything at hand to prove that REC satisfies the principal constraint property. The proof of the theorem consists in describing a method how to lift computation of principal normal forms for projection–free constraints to arbitrary constraints.

Theorem 19. *The constraint system* REC *satisfies the principal constraint property.*

Proof. Given an arbitrary constraint problem (D, ϕ) where $D =^e \exists\bar{\alpha}.D'$ such that D' is projection–free. We consider two cases.

First, assume (D, ϕ) has no normal form. Because of Lemma 18 we know that this holds iff (D', ϕ) does not have a normal form either. The latter can be checked by the normalization procedure for projection–free constraints.

Now, assume (D, ϕ) does have a normal form. We apply Lemma 18 and find that the normal form of (D', ϕ) exists. By assumption we know how to normalize (D', ϕ). That means (D', ϕ) does have a principal normal form and we can compute its principal normal form. With Lemma 17 we can lift the principal normal form of the projection–free constraint problem and obtain the principal normal form of (D, ϕ).

We can conclude that REC satisfies the principal constraint property. ■

8. Conclusion

We have presented a general framework for Hindley/Milner style type systems with constraints. An innovative aspect of the framework is its new formulation of the quantifier introduction rule, which avoids problems in previous work. The formulation requires the presence of a projection operator $\exists$ on constraints. This requirement was the main motivation to progress from a syntactic notion of constraints as sets of formulas to a semantic notion of constraints as cylindric algebras. Cylindric algebras always have a projection operator even though the operator need not be present in syntactic form. Projection is also readily available for the syntactic constraint systems that have been used in type system literature. A simple way to introduce it is by marking some variables as projected. In fact such a marking can usually be reconstructed from a type judgment: simply mark all variables that appear free in neither the final type schemes or the final type environment as projected.

Projection provides an important opportunity for constraint simplification: It is legal to eliminate variables from constraints as long as these variables are projected since such an elimination does not change the constraint's denotation. Simplification in the context of subtypes has already been studied by Pottier [Pot96] and the Hopkins Object Group [TS96]. We plan to investigate in the future how their simplification techniques fit into the HM(X) framework.

Since our framework also includes a subsumption rule based on a given subsumption relation in the constraint system, it can be adapted to a wide variety of type system instances. For instance, the classical Hindley/Milner system falls out by taking subsumption to be syntactic equality in a free algebra, Wand/Rémy style records [Rém89, Wan89] or dimension types [Ken96] fall out by taking some richer notion of equality as subsumption, and standard object calculi [EST95a] fall out by identifying the subtyping and the subsumption relations.

We could give a type soundness result for sound and coherent HM(X) type systems based on a standard untyped denotational semantics. Furthermore, we formulated a generic type inference algorithm for HM(X) type systems. For a large class of constraint systems we could state sufficient conditions under which type inference computes principal types. To design a full language or static analysis based on our approach, one must simply check that the conditions on the constraint system are met. If this is the case, one gets a type inference algorithm and the principal type property for free.

We hope that our results will open the door to a new class of program analyses for program checking which can be tailored to specific application domains. For instance, it should be possible to add a dimension analysis to an existing programming language after the fact and in a modular way, without changing the semantics of the base language or its compiler. Our type system framework would then be the basis of a language tool framework which can be tailored to specific analysis needs. The construction and investigation of such a tool framework remains a topic for future research.

ACKNOWLEDGEMENTS

We thank Alex Aiken, Kim Marriott, Harald Sondergaard, Phil Wadler and the referees for their valuable comments.

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.

BM97. François Bourdoncle and Stephan Metz. Type Checking Higher–Order Polymorphic Multi–Methods. In *Confer-*

ence Record of the Twentyfourth Annual ACM Symposium on Principles of Programming Languages, Paris, France. ACM Press, January 1997.

BSvG95. Kim B. Bruce, Angela Schuet, and Robert van Gent. Polytoil: A type-safe polymorphic object-oriented language (extended abstract). In *Procceding of ECOOP*, pages 27–51. Springer Verlag, 1995. LNCS 952.

CCH$^+$89. Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Functional Programming Languages and Computer Architecture*, pages 273–280, September 1989.

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.

CW85. Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

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.

DM82. Luis Damas and Robin Milner. Principal type schemes for functional programs. January 1982.

EST95a. Jonathan Eifrig, Scott Smith, and Valery Trifonov. Sound polymorphic type inference for objects. In *OOPSLA*, 1995.

EST95b. Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type inference for recursivly 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.

Jon95. Mark P. Jones. Simplifying and improving qualified types. In *FPCA '95: Conference on Functional Programming Languages and Computer Architecture.* ACM Press, 1995.

GLS96. James Gosling, Bill Joy, and Guy Steele. The Java language specification. Java Series, Sun Microsystems, ISBN 0-201-63451-1, 1996.

Kae92. Stefan Kaes. Type inference in the presence of overloading, subtyping, and recursive types. pages 193–204, June 1992.

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

LMM87. J. Lassez, M. Maher, and K. Marriott. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming.* Morgan Kauffman, 1987.

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

Mit84. John C. Mitchell. Coercion and type inference. In *Proceedings of the 11th ACM Symposium on Principles of Programming Languages*, pages 175–185, 1984.

MPS86. D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71:95–130, 1986.

NP93. Tobias Nipkow and Christian Prehofer. Type checking type classes. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, January 10–13, 1993*, pages 409–418. ACM Press, January 1993.

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

OWW95. Martin Odersky, Philip Wadler, and Martin Wehr. A second look at overloading. In *Proc. ACM Conf. on Functional Programming and Computer Architecture*, pages 135–1469, June 1995.

Pal95. Jens Palsberg. Efficient inference of object types. *Information and Computation*, 123(2):198–209, 1995.

Pot96. Francois Pottier. Simplifying subtyping constraints. In *International Conference on Functional Programming*, pages 122–133, May 1996.

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

Rém92a. Didier Rémy. Extending ML type system with a sorted equational theory. Research Report 1766, Institute National de Recherche en Informatique et en Automatique, 1992.

Rém92b. Didier Rémy. Typing record concatenation for free. pages 166–176, January 1992.

Rey85. John C. Reynolds. Three approaches to type structure. In *Proceedings TAPSOFT/CAAP 1985*, pages 97–138. Springer-Verlag, 1985. Lecture Notes in Computer Science 185.

Rob65. J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12:23–41, 1965.

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.

Sul97. Martin Sulzmann. Proofs of Soundness and Completeness of Type Inference for HM(X). Research Report YALEU/DCS/RR-1102, Yale University, Department of Computer Science, February 1997.

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.

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.

Wan87. Mitchell Wand. A simple algorithm and proof for type inference. In *Fundamenta Informaticae X*, pages 115–122. North-Holland, 1987.

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.

Appendix: Proof of Theorem 11 (Soundness)

The following two lemmas can both be proven by a straightforward induction on the derivation $\vdash$. We say a substitution ϕ is *consistent* with respect to a type scheme $\sigma = \forall \bar{\alpha}.D \Rightarrow \tau$ if $\psi D \in \mathcal{S}$ where we assume there are no name clashes between $\bar{\alpha}$ and ψ. This extends naturally to type environments. Furthermore, a substitution ϕ is *consistent* with respect to a constraint C if $\phi C \in \mathcal{S}$.

Lemma 1. *Given $C, \Gamma \vdash e : \sigma$ and a substitution ϕ such that ϕ is consistent with respect to C and Γ. Then $\phi C, \phi\Gamma \vdash e : \phi\sigma$.*

Lemma 2. *Given $C, \Gamma \vdash e : \sigma$ and a constraint $D \in \mathcal{S}$ such that $D \vdash^e C$. Then $D, \Gamma \vdash e : \sigma$.*

We restate Theorem 11 in the following lemma.

Lemma 3 Soundness of $\vdash^W$. *Given a type environment Γ and a term e. If $\psi, C, \Gamma \vdash^W e : \tau$ then $C, \psi\Gamma \vdash e : \tau$, $\psi C = C$ and $\psi\tau = \tau$.*

Proof. We apply induction on the derivation $\vdash^W$. We only consider one case. The other cases can be proven in a similar style.

Case (App) We have the following situation:

$$\frac{\begin{array}{c}\psi_1, C_1, \Gamma \vdash^W e_1 : \tau_1 \qquad \psi_2, C_2, \Gamma \vdash^W e_2 : \tau_2 \\ \psi' = \psi_1 \sqcup \psi_2 \\ D = C_1 \wedge C_2 \wedge (\tau_1 \preceq \tau_2 \rightarrow \alpha) \qquad \alpha \text{ new} \\ (C, \psi) = \mathit{normalize}(D, \psi')\end{array}}{\psi_{|fv(\Gamma)}, C, \Gamma \vdash^W e_1 e_2 : \psi(\alpha)}$$

We apply the induction hypothesis to the left and right premise and obtain

$$C_1, \psi_1\Gamma \vdash e_1 : \tau_1 \quad \psi_1 C_1 = C_1 \quad \psi_1\tau_1 = \tau_1$$

and

$$C_2, \psi_2\Gamma \vdash e_2 : \tau_2 \quad \psi_2 C_2 = C_2 \quad \psi_2\tau_2 = \tau_2$$

With Lemma 2 we can conclude that

$$C, \psi_1\Gamma \vdash e_1 : \tau_1 \quad C, \psi_2\Gamma \vdash e_2 : \tau_2$$

W.l.o.g. we can assume that all identifier in Γ are contained in e_1 and e_2 and not more. This fact and normalization ensures that ψ is consistent in C and Γ. Then we can apply Lemma 1 and obtain

$$C, \psi\Gamma \vdash e_1 : \psi\tau_1 \quad C, \psi\Gamma \vdash e_2 : \psi\tau_2$$

We know that $C \vdash^e (\psi\tau_1 \preceq \psi\tau_2 \rightarrow \psi(\alpha))$ and apply the (Sub) rule to get $C, \psi\Gamma \vdash e : \psi\tau_2 \rightarrow \psi(\alpha)$. It remains to apply the (App) rule and we find

$$C, \psi\Gamma \vdash e_1 e_2 : \psi(\alpha)$$

■

Appendix: Proof of Theorem 12 (Completeness)

We give now a proof sketch for completeness for HM(X) type systems where X $\in \mathcal{X}^r$ satisfies the principal constraint property. Some technical lemmas (which we will point out) rely on the fact that X is regular. The proof for $\mathcal{X}^a$is similar, but there we only need weaker versions of these technical lemmas which do not rely on the regularity of the constraint system. In order to prove completeness we have to do a little more work. The idea is to introduce two intermediate derivations, and to show that all derivations have the same expressive power.

First, we introduce some conventions. The generalization procedure *gen* takes a constraint C, a type environment Γ and a type τ and returns the generalized constraint and type, written $\mathit{gen}(C, \Gamma, \tau) = (C', \sigma)$. We use two specialized generalization versions: $\mathit{gen}_1(C, \Gamma, \tau)$ returns only the constraint part and $\mathit{gen}_2(C, \Gamma, \tau)$ returns only the type scheme part.

We introduce some basic lemmas. Most of them are stated without proof. A detailed discussion can be found in [Sul97]. The following two lemmas rely on the fact that we only consider regular theories. We give the proof for one lemma where one can see that X needs to be a regular theory. The first lemma states that we can lift entailment between two constraints to the generalized constraints.

Lemma 1. *Given a type context Γ, constraints $C, \tilde{C}$, types τ, τ' and substitutions ϕ, ϕ', ψ such that $C \vdash^e \phi'\tilde{C}$ and $\psi \leq^{\phi'}_{fv(\Gamma)} \phi$. Then $C_o \vdash^e \phi'\tilde{C}_o$ where $C_o = \mathit{gen}_2(C, \phi\Gamma, \tau')$ and $\tilde{C}_o = \mathit{gen}_2(\tilde{C}, \psi\Gamma, \tau)$.*

Proof. W.l.o.g. we assume $C_o = \exists\bar{\alpha}.C$ and $\tilde{C}_o = \exists\bar{\beta}.\tilde{C}$. We show that $\bar{\alpha} \notin \mathit{fv}(\phi'\tilde{C}_o)$. Assume the contrary. W.l.o.g.

$$\bar{\alpha} \notin \mathit{fv}(\Gamma) \cup \mathit{fv}(\tilde{C}_o) \cup \mathit{codom}(\phi) \qquad \text{(A1)}$$

because we can always rename bound variables and during type inference always new type variables have been introduced. That means there is a $\gamma \in \mathit{fv}(\tilde{C}_o)$ such that $\bar{\alpha} \in \mathit{fv}(\phi'(\gamma))$. Further it holds that $\gamma \notin \mathit{fv}(\psi\Gamma)$. Assume $\gamma \in \mathit{fv}(\psi\Gamma)$ then there is a $\delta \in \mathit{fv}(\Gamma)$ such that $\delta \in \mathit{fv}(\psi(\gamma))$. We know that $\phi(\delta) = \phi' \circ \psi(\delta)$ (here we need the fact that X is regular, both sides of the equation contain the same set of free variables) and then we find $\bar{\alpha} \in \mathit{codom}(\phi)$ which is a contradiction to A1. We get $\gamma \notin \mathit{fv}(\psi\Gamma)$ and $\gamma \in \mathit{fv}(\tilde{C}_o)$. But this is again a contradiction because $\tilde{C}_o$ is a generalized constraint. Our starting assumption was false and we find that $\bar{\alpha} \notin \mathit{fv}(\phi'\tilde{C}_o)$.

Now, we can conclude that $\tilde{C} \vdash^e \tilde{C}_o$. Then it follows that $\phi'\tilde{C} \vdash^e \phi'\tilde{C}_o$. This yields $C \vdash^e \phi'\tilde{C}_o$. Finally, we obtain $C_o \vdash^e \exists\bar{\alpha}.\phi'\tilde{C}_o$ and because $\bar{\alpha} \notin fv(\phi'\tilde{C}_o)$ that we means we get $C_o \vdash^e \phi'\tilde{C}_o$ as desired. ■

Remark. The proof of the previous lemma relies on the fact that X is regular. For X in $\mathcal{X}^a$ we only need a restricted version of this lemma. Therefore, we still can achieve complete type inference for X in $\mathcal{X}^a$.

The next lemma is similar to the previous one, except that it compares types instead of constraints.

Lemma 2. *Given a type context Γ, constraints $C, \tilde{C}$, types τ, τ' and substitutions ϕ, ϕ', ψ such that $C \vdash^e \phi'\tilde{C}$, $C \vdash^i \phi'\tau \preceq \tau'$ and $\psi \leq^{\phi'}_{fv(\Gamma)} \phi$. Then $\vdash^i \phi'\tilde{\sigma}_o \preceq \sigma_o$ where $\sigma_o = gen_1(C, \phi\Gamma, \tau')$ and $\tilde{\sigma}_o = gen_1(\tilde{C}, \psi\Gamma, \tau)$.*

The next lemma states that we can lift some properties about a constraint and a substitution to the same constraint but extended substitution.

Lemma 3. *Given a set U of variables, constraints C_1, C' and substitutions $\psi, \psi_1, \psi_2, \phi, \phi_1$ such that $\psi_1 C_1 = C_1$, $C' \vdash^e \phi'_1 C_1$, $\psi = \psi_1 \sqcup \psi_2$, $\psi \leq^{\phi'}_U \phi$, $\psi_1 \leq^{\phi'_1}_U \phi$, $codom(\psi_2) \cap fv(C_1) \subseteq U$ and $codom(\psi_1) \cap fv(C_2) \subseteq U$. Then $C' \vdash^e (\phi' \circ \psi)C_1$.*

The next Lemma is similar to the previous one but it is stated for the $\vdash^i$ relation.

Lemma 4. *Given a set U of variables, a constraint C', type schemes $\tilde{\sigma}, \sigma''$ and substitutions $\psi, \psi_1, \psi_2, \phi, \phi_1$ such that $\psi_1\tilde{\sigma} = \tilde{\sigma}$, $C' \vdash^i \phi'_1\tilde{\sigma} \preceq \sigma''$, $\psi = \psi_1 \sqcup \psi_2$, $\psi \leq^{\phi'}_U \phi$, $\psi_1 \leq^{\phi'_1}_U \phi$, $codom(\psi_2) \cap fv(C_1) \subseteq U$ and $codom(\psi_1) \cap fv(C_2) \subseteq U$. Then $C' \vdash^i (\phi' \circ \psi)\tilde{\sigma} \preceq \sigma''$.*

Now, we introduce the intermediate derivations. We introduce a derivation $\vdash^2$ which is based on derivation $\vdash$ in figure 2. Instead of rule ($\forall$ Elim) we have the following new rule:

$$\text{(Inst)} \quad C, \Gamma \vdash^2 x : \tau \quad (x : \sigma \in \Gamma \quad C \vdash^i \sigma \preceq \tau)$$

All other rules stay unchanged. Note, also the (Var) rule is still present in derivation $\vdash^2$. The idea of derivation $\vdash^2$ is simply to enforce ($\forall$ Elim) steps as early as possible.

Next, we consider a syntax directed derivation $\vdash^d$. We also want to get rid of the ($\forall$ Intro) rule. This rule is combined with the (Let) rule. Furthermore, the (Var) and (Inst) rules are combined in the (Var–Inst) rule. The rules are as follows:

$$\text{(Var–Inst)} \quad C, \Gamma \vdash^d x : \tau \quad (x : \sigma \in \Gamma \quad C \vdash^i \sigma \preceq \tau)$$

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

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

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

$$\text{(Let)} \quad \frac{\begin{array}{c} C, \Gamma_x \vdash^d e : \tau \quad (C', \sigma) = gen(C, \Gamma_x, \tau) \\ C'', \Gamma_x.x : \sigma \vdash^d e' : \tau' \end{array}}{C' \wedge C'', \Gamma_x \vdash^d \mathsf{let}\ x = e\ \mathsf{in}\ e' : \tau'}$$

In the (Let) rule we implicitely require that the constraint $C' \wedge C''$ is in solved form. Remember that the set of constraints of solved forms is not necessarily closed under $\wedge$. That means, when we apply the (Let) rule we always have to ensure that $C' \wedge C''$ is in solved form.

The next lemmas state how these derivations are connected. The first two of these lemmas can both be proven by a straightforward induction on the derivation relation.

Lemma 5 Equivalence of $\vdash$ and $\vdash^2$. *Given a type environment Γ, a constraint C, a term e and a type scheme σ. Then $C, \Gamma \vdash e : \sigma$ iff $C, \Gamma \vdash^2 e : \sigma$.*

Lemma 6 Soundness of $\vdash^d$. *Given $C, \Gamma \vdash^d e : \tau$. Then $C, \Gamma \vdash e : \tau$.*

We now show that $\vdash^d$ is complete with respect to $\vdash^2$ and $\vdash^W$ is complete with respect to $\vdash^2$. In order to prove it we have to strengthen the assumption about the given type environment. This is due to the (Let) rule where the two premises use different type environments. Therefore, we introduce the following definition.

Definition. *Let C be a constraint and Γ and Γ' be type environments such that $\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ and $\Gamma' = \{x_1 : \sigma'_1, \ldots, x_n : \sigma'_n\}$. Then $C \vdash^i \Gamma' \preceq \Gamma$ iff $C \vdash^i \sigma'_i \preceq \sigma_i \quad \forall i : i \in \{1, \ldots, n\}$.*

In the following theorem it is essential that the type environment Γ' is realizable. Remember, a type environment Γ' is realizable in a constraint C if for every $x : \sigma \in \Gamma'$ there is a τ such that $C \vdash^i \sigma \preceq \tau$.

Lemma 7 Completeness of $\vdash^d$. *Given $C', \Gamma' \vdash^2 e : \sigma'$, $C' \vdash^i \Gamma \preceq \Gamma'$ and Γ' is realizable in C'. Then*

$$\begin{array}{ll} \textit{(a)}\ \ \sigma' = \tau : & C, \Gamma \vdash^d e : \tau \quad C' \vdash^e C \\ \textit{(b) otherwise} : & C, \Gamma \vdash^d e : \tau \quad (\sigma_o, C_o) = gen(C, \Gamma, \tau) \\ & C' \vdash^e C_o \quad C' \vdash^i \sigma_o \preceq \sigma \end{array}$$

Proof. We use induction on the derivation $\vdash^d$. Due to space limitation we only show two cases.

Case (Var) We know that $C', \Gamma' \vdash^2 x : \sigma'$ where $x : \sigma' \in \Gamma'$. By assumption we know there is a $x : \sigma$ in Γ such that $C' \vdash^i \sigma \preceq \sigma'$. If $\sigma' = \tau$ then we can immediately apply the (Var–Inst) rule and we are done. Otherwise, w.l.o.g. we can assume that $\sigma = \forall \bar{\alpha}.D \Rightarrow \tau'$. We set $C = [\bar{\beta}/\bar{\alpha}]D$ and $\tau = [\bar{\beta}/\bar{\alpha}]\tau'$ where $\bar{\beta}$ are fresh type variables. We apply again the (Var–Inst) rule and find $C, \Gamma \vdash^d x : \tau$. We set $(\sigma_o, C_o) = gen(C, \Gamma, \tau)$ where σ_o is essentially a renamed version of σ. We find that $C' \vdash^i \sigma_o \preceq \sigma'$. By assumption Γ' is realizable in C', hence there is a τ such that $C' \vdash^e [\bar{\tau}/\bar{\alpha}]D$. This leads us to the conclusion that $C' \vdash^e C_o$ and we are done.

Case (Let) We have the following situation:

$$\frac{C', \Gamma'_x \vdash^2 e : \sigma \quad C', \Gamma'_x.x : \sigma \vdash^2 e' : \tau'}{C', \Gamma'_x \vdash^2 \mathsf{let}\ x = e\,\mathsf{in}\ e' : \tau'}$$

First, we consider the case if σ is a type τ. We apply the induction hypothesis to left premise and obtain $C_1, \Gamma_x \vdash^d e : \tau$ and $C' \vdash^e C_1$. We set $(\sigma_o, C_o) = gen(C_1, \Gamma_x, \tau)$. It is an easy observation that $C' \vdash^i \sigma_o \preceq \tau$ holds. Now, we apply the induction hypothesis to the right premise. This yields $C_2, \Gamma_x.x : \sigma_o \vdash^d e' : \tau'$ and $C' \vdash^e C_2$. We know that $C' \vdash^e C_o \wedge C_2$ which ensures that $C_o \wedge C_2$ is in solved form. We can apply the (Let) rule and obtain $C_o \wedge C_2, \Gamma_x \vdash^d \mathsf{let}\ x = e\,\mathsf{in}\ e' : \tau'$.

Now, let us consider the case if σ is a type scheme. Application of the induction hypothesis to the left premise yields:

$$\begin{gathered} C_1, \Gamma_x \vdash^d e : \tau \quad (\sigma_o, C_o) = gen(C_1, \Gamma_x, \tau) \\ C' \vdash^i \sigma_o \preceq \sigma \quad C' \vdash^e C_o. \end{gathered}$$

To apply the induction hypothesis to the right premise we have to show that $\Gamma'_x.x : \sigma$ is realizable in C'. We know that $C', \Gamma'_x.x : \sigma \vdash^2 e : \tau'$ holds. If x does not appear in the free variables of e it is sufficient to consider only Γ'_x which is by assumption realizable. Otherwise we know that that the type of x must have been instantiated to a monomorphic type which shows that $\Gamma'_x.x : \sigma$ is realizable in C'. Then we can apply the induction hypothesis to the right and find

$$C_2, \Gamma_x.x : \sigma_o \vdash^d e : \tau' \quad C' \vdash^e C_2.$$

We can conclude that $C' \vdash^e C_o \wedge C_2$ which ensures that $C_o \wedge C_2$ is in solved form. We can apply the (Let) rule and find

$$C_o \wedge C_2, \Gamma_x \vdash^d \mathsf{let}\ x = e\,\mathsf{in}\ e' : \tau'$$

■

Lemma 8. *(Completeness of $\vdash^W$) Given $C', \phi\Gamma \vdash^d e : \tau'$. Then*

$$\psi, C, \Gamma \vdash^W e : \tau$$

for some substitutions ψ, ϕ', constraint C and type τ such that,

$$\psi \leq^{\phi'}_{fv(\Gamma)} \phi \quad C' \vdash^e \phi' C \quad C' \vdash^i \phi'\tau \preceq \tau'$$

Proof. We use induction on the derivation $\vdash^d$. Due to space limitation we only show the two interesting cases.

Case (App) We have the following situation:

$$\frac{C', \phi\Gamma \vdash^d e_1 : \tau'_1 \to \tau'_2 \quad C', \phi\Gamma \vdash^d e_2 : \tau'_1}{C', \phi\Gamma \vdash^d e_1 e_2 : \tau'_2}$$

Application of the induction hypothesis yields

$$\begin{gathered} \psi_1, C_1, \Gamma \vdash^W e_1 : \tau_1 \quad \psi_1 \leq^{\phi'_1}_{fv(\Gamma)} \phi \\ C' \vdash^e \phi'_1 C_1 \quad C' \vdash^i \phi'_1 \tau_1 \preceq \tau'_1 \to \tau'_2 \end{gathered} \tag{A2}$$

and

$$\begin{gathered} \psi_2, C_2, \Gamma \vdash^W e_2 : \tau_2 \quad \psi_2 \leq^{\phi'_2}_{fv(\Gamma)} \phi \\ C' \vdash^e \phi'_2 C_2 \quad C' \vdash^i \phi'_2 \tau_2 \preceq \tau'_1 \end{gathered}$$

We set $\psi' = \psi_1 \sqcup \psi_2$. Then we find that $\psi' \leq^{\phi'}_{fv(\Gamma)} \phi$. We want to apply Lemmas 3, 4. We identify the set U in these lemmas with $fv(\Gamma)$. We assume that type variables introduced in one part of the inference tree do not appear in the other part. Formally, this means that

$$codom(\psi_2) \cap fv(C_1) \subseteq fv(\Gamma)$$

and

$$codom(\psi_1) \cap fv(C_2) \subseteq fv(\Gamma)$$

All preconditions of Lemmas 3, 4 are fulfilled. We can conclude that

$$\begin{gathered} C' \vdash^e (\phi' \circ \psi')C_1 \qquad C' \vdash^i (\phi' \circ \psi')\tau_1 \preceq \tau'_1 \to \tau'_2 \\ C' \vdash^e (\phi' \circ \psi')C_2 \qquad C' \vdash^i (\phi' \circ \psi')\tau_2 \preceq \tau'_1 \end{gathered}$$

We set $D = C_1 \wedge C_2 \wedge (\tau_1 \preceq \tau_2 \to \alpha)$ where α is a fresh type variable. Then we obtain that $C' \vdash^e (\phi' \circ \psi' \circ [\tau'_2/\alpha])D$. We find that $(C', \phi' \circ \psi' \circ [\tau'_2/\alpha])$ is a normal form of (D, ψ'). By assumption HM(X) satisfies the principal constraint property. We obtain that (C, ψ) is the principal normal form of (D, ψ') where $\psi \leq^{\phi''} \phi' \circ \psi' \circ [\tau'_2/\alpha]$. Because (C, ψ) is principal we find that $C' \vdash^e \phi'' C$. W.l.o.g. $(\phi' \circ \psi')\tau'_2 = \tau'_2$. Then, we can conclude that $(\phi' \circ \psi' \circ [\tau'_2/\alpha])_{|fv(\Gamma)} = \phi$. This leads to $\psi \leq^{\phi''}_{fv(\Gamma)} \phi$. Furthermore, it holds that $\phi''(\alpha) = \tau'_2$ because

$$\tau'_2 = \phi' \circ \psi' \circ [\tau'_2/\alpha](\alpha) = \phi'' \circ \psi(\alpha) = \phi''(\alpha)$$

The last reasoning steps holds because α is a new type variable therefore $\alpha \notin dom(\psi)$. Finally, we apply the (App) rule and find

$$\psi_{|fv(\Gamma)}, C, \Gamma \vdash^W e_1 e_2 : \psi(\alpha)$$

which establishes the induction step.

Case (Let) We have the following situation:

$$\frac{\begin{array}{c} C_1, \phi\Gamma_x \vdash^d e : \tau \quad (\sigma, C_2) = gen(C_1, \phi\Gamma_x, \tau) \\ C_3, \phi\Gamma_x.x : \sigma \vdash^d e' : \tau' \end{array}}{C_2 \wedge C_3, \phi\Gamma_x \vdash^d \mathsf{let}\ x = e\,\mathsf{in}\ e' : \tau'}$$

Induction hypothesis applied to the left part yields

$$\begin{array}{c} \psi_1, \tilde{C}_1, \Gamma_x \vdash^W e : \tau_1 \quad \psi_1 \leq^{\phi'_1}_{fv(\Gamma_x)} \phi \\ C_1 \vdash^e \phi'_1 \tilde{C}_1 \quad C_1 \vdash^i \phi'_1 \tau_1 \preceq \tau \end{array} \tag{A3}$$

From Lemma 2 and Lemma 1 and A3 we obtain that

$$C_2 \vdash^e \phi'_1 \tilde{C}_2 \qquad \vdash^i \phi'_1 \sigma_1 \preceq \sigma \tag{A4}$$

where $(\sigma_1, \tilde{C}_2) = gen(\tilde{C}_1, \psi\Gamma_x, \tau_1)$. We set $\tilde{\phi} = \phi'_1 \circ \phi$. Then it holds that

$$\vdash^i \tilde{\phi}(\Gamma_x.x : \sigma_1) \preceq \phi\Gamma_x.x : \sigma \tag{A5}$$

because

$$\tilde{\phi}\sigma_1 = (\phi'_1 \circ \phi)\sigma_1 = (\phi'_1 \circ (\phi'_1 \circ \psi_1)_{|fv(\Gamma)})\sigma_1 = \phi'_1 \sigma_1$$

An easy observation yields

$$\tilde{\phi}_{|fv(\Gamma_x)} = \phi \tag{A6}$$

We rewrite the right premise with the stronger type environment in A5 (this fact is stated without proof but can be found in detail in [Sul97]) and find

$$C_3, \tilde{\phi}(\Gamma_x.x : \sigma_1) \vdash^d e' : \tau'$$

Now, we are able to apply the induction hypothesis to the right part and find

$$\begin{array}{c} \psi_2, \tilde{C}_3, \Gamma_x.x : \sigma_1 \vdash^W e' : \tau'_1 \\ \psi_2 \leq^{\phi'_2}_{fv(\Gamma_x) \cup fv(\sigma_1)} \tilde{\phi} \\ C_3 \vdash^e \phi'_2 \tilde{C}_3 \quad C_3 \vdash^i \phi'_2 \tau'_1 \preceq \tau' \end{array} \tag{A7}$$

From A3 we can deduce that

$$\psi_1 \leq^{\phi'_1}_{fv(\Gamma_x) \cup fv(\sigma_1)} \tilde{\phi} \tag{A8}$$

because of A3 and A6 it holds that

$$(\phi'_1 \circ \psi_1)_{|fv(\Gamma_x)} = \phi = \tilde{\phi}_{|fv(\Gamma_x)}$$

and if $\alpha \in fv(\sigma_1)$ we can assume that $\alpha \notin fv(\Gamma_x)$ then we know that

$$\phi(\alpha) = \alpha \quad \psi_1(\alpha) = \alpha$$

We can deduce that

$$\phi'_1 \circ \psi_1(\alpha) = \phi'_1(\alpha) = \tilde{\phi}(\alpha)$$

Then from A7 and A8 we find that the least upper bound of ψ_1 and ψ_2 exists. It holds that

$$\psi' \leq^{\phi'}_{fv(\Gamma_x) \cup fv(\sigma)} \phi \tag{A9}$$

where $\psi' = \psi_1 \sqcup \psi_2$. With A6 and from A9 we find that

$$\psi' \leq^{\phi'}_{fv(\Gamma_x)} \phi$$

From A4 and A3 we know that

$$C_2 \vdash^e \phi'_1 \tilde{C}_2 \quad C_3 \vdash^e \phi'_2 \tilde{C}_3 \quad C_3 \vdash^i \phi'_2 \tau'_1 \preceq \tau'$$

As in the (App) case we can conclude from Lemmas 3, 4 that

$$C_2 \vdash^e (\phi' \circ \psi')\tilde{C}_2 \qquad C_3 \vdash^e (\phi' \circ \psi')\tilde{C}_3$$

and

$$C_3 \vdash^i (\phi' \circ \psi')\tau'_1 \preceq \tau'$$

We set $D = \tilde{C}_2 \wedge \tilde{C}_3$. Then we obtain that $(C_2 \wedge C_3, \phi' \circ \psi')$ is a normal form of (D, ψ'). By assumption HM(X) satisfies the principal constraint property. Assume (C, ψ) is the principal normal form of (D, ψ') where $\psi \leq^{\phi''} \phi' \circ \psi'$. Now, we can apply the (Let) rule and find

$$\psi_{|fv(\Gamma)}, C, \Gamma_x \vdash^W \mathsf{let}\ x = e\,\mathsf{in}\ e' : \psi\tau'_1$$

Furthermore, we obtain that

$$C_2 \wedge C_3 \vdash^e \phi'' C \qquad C_2 \wedge C_3 \vdash^i (\phi'' \circ \psi)\tau'_1 \preceq \tau'$$

where $\psi \leq^{\phi''}_{fv(\Gamma_x)} \phi$.

■

Now we have everything at hand to prove completeness of type inference.

Theorem 9. *Given $C', \phi\Gamma \vdash e : \sigma'$ and $\phi\Gamma$ is realizable in C'. Then*

$$\psi, C, \Gamma \vdash^W e : \tau$$

for some substitutions ϕ', ψ, constraint C and type τ such that,

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

where $(\sigma_o, C_o) = gen(C, \psi\Gamma, \tau)$.

Proof. First, we apply Lemma 5 in order to get a derivation in $\vdash^2$. Then, we can apply Lemma 7 (completeness of $\vdash^d$). This yields

$$\begin{array}{lll} \text{(a)} & \sigma' = \tau : & C, \phi\Gamma \vdash^d e : \tau \quad C' \vdash^e C \\ \text{(b)} & \text{otherwise :} & C, \phi\Gamma \vdash^d e : \tau \quad (\sigma_o, C_o) = gen(C, \phi\Gamma, \tau) \\ & & C' \vdash^e C_o \quad C' \vdash^i \sigma_o \preceq \sigma' \end{array} \tag{A10}$$

After that we apply Lemma 8 (completeness of $\vdash^W$) and find

$$\begin{array}{c} \psi, \tilde{C}, \Gamma \vdash^W e : \tilde{\tau} \quad \psi \leq^{\phi'}_{fv(\Gamma)} \phi \\ C \vdash^e \phi' \tilde{C} \quad C \vdash^i \phi' \tilde{\tau} \preceq \tau \end{array}$$

We set $(\sigma_o, C_o) = gen(C, \psi\Gamma, \tau)$. It remains to show

1. $C' \vdash^i \phi' \tilde{\sigma}_o \preceq \sigma'$
2. $C' \vdash^e \phi' \tilde{C}_o$.

This fact follows by application of the Lifting Lemmas 3, 4.

■