Type Inference for Parametric Type Classes

Kung Chen, Martin Odersky, Paul Hudak

Research Report YALEU/DCS/RR-900
June, 1991

This work is supported by Darpa grant N00014-91-J-4043

Type Inference for Parametric Type Classes

Kung Chen, Martin Odersky, Paul Hudak

Department of Computer Science
Yale University
Box 2158 Yale Station
New Haven, CT 06520

June 11, 1992

Abstract

Haskell's type classes permit the definition of overloaded operators in a rigorous and (fairly) general manner that integrates well with the underlying Hindley-Milner type system. As a result, operators that are monomorphic in other typed languages can be given a more general type. Most notably missing, however, are overloaded functions for data selection and construction. Such overloaded functions are quite useful, but the current Haskell type system is not expressive enough to support them.

We introduce the notion of ***parametric type classes*** as a significant generalization of Haskell's type classes. A parametric type class is a class that has type parameters in addition to the placeholder variable which is always present in a class declaration. Haskell's type classes are special instances of parametric type classes with just a placeholder but no parameters. We show that this generalization is essential to represent container classes with overloaded data constructor and selector operations. Furthermore, through a simple encoding scheme, we show that parametric type classes are able to capture the notion of "type constructor variables", thus permitting the definition of overloaded operators such as `map`.

The underlying type system supporting our proposed generalization is an extension of Hindley-Milner type system with parametric type classes. The range of type variables are bounded by constraints. Rules for satisfiability and entailment of these constraints are given by a context-constrained instance theory that is separate from the inference rules of the type system. The decoupling of the instance theory from the type inference system makes our system more modular than previous work. We prove that the resulting type system is decidable, and provide an effective type inference algorithm to compute the principal types for well-typed expressions.

1 Introduction

Haskell's *type classes* provide a structured way to introduce overloaded functions, and are perhaps the most innovative (and somewhat controversial) aspect of the language design [HJW91]. Type classes permit the definition of overloaded operators in a rigorous and (fairly) general manner that integrates well with the underlying Hindley-Milner type system. As a result, operators that are monomorphic in other typed languages can be given a more general type. Examples include the numeric operators, reading and writing of arbitrary datatypes, and comparison operators such as equality, ordering, etc.

Haskell's type classes have proven to be quite useful. Most notably missing, however, are overloaded functions for data selection and construction. Such overloaded functions are quite useful, but the current Haskell type system is not expressive enough to support them (of course, no other language that we know if is capable of supporting them in a type-safe way either).

A Motivating Example

As a simple example, consider the concept of a *sequence*: a linearly ordered collection of elements, all of the same type. There are at least two reasonable implementations of sequences, linked lists and vectors. There is an efficiency tradeoff in choosing one of these representations: lists support the efficient addition of new elements, whereas vectors support efficient random (including parallel) access. Currently the choice between representations is made at the programming language level. Most functional languages provide lists as the "core" data structure (often with special syntax to support them), relegating arrays to somewhat of a second-class status. Other languages, such as Sisal and Nial, reverse this choice and provide special syntax for arrays instead of lists (this often reflects their bias toward parallel and/or scientific computation).

Of course, it is possible to design a language which places equal emphasis on both "container structures". However, a naive approach faces the problem that every function on sequences has to be implemented twice, once for lists and once for arrays. The obvious cure for this name-space pollution and duplicated code is *overloading*. In our context, that means specifying the notion of a sequence as a *type class* with (at least) lists and vectors as instance types. Using Haskell-like notation, this would amount to the following declarations:

```
class Sequence a s
where cons :: a -> s -> s
      nth  :: s -> Int -> a
      len  :: s -> Int

instance Sequence a (List a)
where cons  =  (:)
      nth   =  (!)
      len   =  (#)

instance Sequence a (Vector a)
where cons  =  vecCons
      nth   =  vecNth
      len   =  vecLen
```

This defines the overloaded constructor `cons`, overloaded indexing selector `nth`, and a length function `len`. (Note the resemblance to a "container class" in object-oriented programming.)

The only problem with this code is that it is not valid Haskell, since Haskell's type classes are permitted to constrain only one type, thus ruling out a declaration such as "`class Sequence a s`". In essence, this restriction forces overloaded constructors and selectors to be monomorphic (which makes them fairly useless).

Even if this restriction did not exist, there is another problem with the current type class mechanism, which can be demonstrated through the typing of `len`:

```
Sequence a s => s -> Int
```

Even if multi-argument type classes were allowed, this qualified type would still not be valid Haskell since it is *ambiguous*: Type variable `a` occurs in the context (`Sequence a s`), but not in the type-part proper (`s->Int`). [1] Ambiguous types need to be rejected, since they have several, possibly conflicting, implementations.

A related, but harder, problem arises if we extend our example to include an overloaded *map* function. Having such a function is attractive, since together with `join` and `filter`, it allows us to generalize (i.e. overload) the notion of a "list comprehension" to include *all* instances of `Sequence`, not just lists. In [CHO92], we elaborate on this, extending it further to comprehensions for arbitrary instances of class *monad*, such as bags and lists. This seems quite natural since, after all, the domain of sets is where the "comprehension" notation came from. However, a problem becomes evident as soon as we attempt to give a type for `map`.

```
map: (Sequence a sa, Sequence b sb) => (a -> b) -> sa -> sb.
```

This type is too general, since it would admit also implementations that take one sequence type (e.g. a list) and return another (e.g. a vector). Generality is costly in this context since it again leads to ambiguity. For instance, the function composition (`map f . map g`) would be ambiguous; the type of `map g`, which does not appear in the type of the enclosing expression, can be either a list or a vector.

What is needed is some way to specify that `map` returns the same kind of sequence as its argument, but with a possibly different element type. A nice way to notate this type would be:

```
map: Sequence (s a) => (a -> b) -> s a -> s b
```

where `s` is a variable which ranges over type constructors instead of types. To accommodate this, `Sequence` should now be viewed as a type constructor class instead of a type class. However, because the instance relationships are now expressed at the functor-level, there is the danger (as has been conjectured in [Lil91]) that second order unification is needed to reconstruct types, thus rendering the system undecidable.

To solve these problems, we introduce the notion of *parametric type classes* as a significant generalization of Haskell's type classes. Parametric type classes can have type arguments in addition to

[1] Satish Thatte has pointed out that the ambiguity problem can be circumvented in this case by having the `len` operation in a different class from the `cons` and `nth` operations. This is true in principle, but it would severely restrict the way operators can be grouped.

the constrained type variable, and thus are able to express classes such as `Sequence` defined earlier. Moreover, through a simple encoding scheme, we show that parametric type classes are able to capture the notion of "type constructor variables," thus permitting the definition of overloaded operators such as `map`. As a matter of fact, parametric type classes are a conservative extension of Haskell's type system: If all classes are parameterless, the two systems are equivalent. Most importantly, we prove that the resulting type system is decidable, and provide an effective type inference algorithm to compute the principal types for well-typed expressions.

The rest of this report is organized as follows: Section 2 introduces parametric type classes. Section 3 presents them formally, in a non-deterministic type system. Section 4 presents an equivalent syntax-directed system that bridges the gap between the non-deterministic system and a type reconstruction algorithm. Section 5 introduces context-preserving unification and presents such a unification algorithm to be used in the type inference algorithm. Section 6 presents a simple algorithm to compute principal types for our type system and proves its soundness and completeness. Section 7 explains when a type scheme is ambiguous and how operators such as `map` can be typed in our system. Section 8 describes related work. Section 9 concludes.

2 Parametric Type Classes

A parametric type class is a class that has type parameters in addition to the placeholder variable which is always present in a class declaration. To distinguish between placeholder and type parameters, we write the placeholder in front of the class, separated by an infix (::). For instance:

```
class t :: Eq where
class s :: Sequence a where
```

The first definition introduces a class without parameters; in Haskell this would be written `class Eq t`. The second definition defines a type class `Sequence` with one parameter; this cannot be expressed in standard Haskell. The infix (::) notation is also used in instance declarations and contexts. The two instance declarations of `Sequence` presented in the last section would now be written:

```
inst List a   :: Sequence a where ...
inst Vector a :: Sequence a where ...
```

In an instance declaration, of form `T :: Sequence a`, say, the type `T` must not be a variable. Furthermore, if two types `T1` and `T2` are both declared to be instances of `Sequence`, then their top-level type constructors must be different. Thus, the instance declarations given above are both valid. On the other hand,

```
inst a :: Sequence (List a)
```

would violate the first restriction, and

```
inst List Int  :: Sequence Int
inst List Char :: Sequence Char
```

Type variables	α			
Type constructors	κ			
Class constructors	c			
Types	τ	$::=$	$() \mid \kappa\,\tau \mid \alpha \mid \tau_1 \times \tau_2 \mid \tau_1 \rightarrow \tau_2$	
Type schemes	σ	$::=$	$\forall\alpha::\Gamma.\sigma \mid \tau$	
Type classes	γ	$::=$	$c\,\tau$	
Class sets	Γ	$::=$	$\{c_1\,\tau_1, \ldots, c_n\,\tau_n\}$	$(n \geq 0, c_i$ pairwise disjoint$)$
Contexts	C	$::=$	$\{\alpha_1::\Gamma_1, \ldots, \alpha_n::\Gamma_n\}$	$(n \geq 0)$
Expressions	e	$::=$	$x \mid e\;e' \mid \lambda x.e \mid$ **let** $x = e'$ **in** e	
Programs	p	$::=$	**class** $\alpha::\gamma$ **where** $x:\sigma$ **in** p	
		$\mid$	**inst** $C \Rightarrow \tau::\gamma$ **where** $x = e$ **in** p	
		$\mid$	e	

Figure 1: Abstract Syntax of Mini-Haskell+

would violate the second restriction. Effectively, these restrictions ensure that in a proof of an instance relationship every step is determined by the class name and the type in placeholder position. The class parameter types, on the other hand, depend on the placeholder type.

One consequence of these restrictions is that there is at most one way to deduce that a type is an instance of a class. This is necessary to guarantee *coherence*. It is not sufficient, since types might be ambiguous; see Section 7 for a discussion. Another consequence is that sets of instance predicates are now subject to a *consistency* criterion: If we have both `T :: Sequence a` and `T :: Sequence b` then we must have `a = b`. That is, `a = b` is a logical consequence of the two instance predicates and the restrictions on instance declarations. The type reconstruction algorithm enforces consistency in this situation by unifying `a` and `b`.

Enforcing consistency early helps in keeping types small. Otherwise, we could get many superfluous instance constraints in types. As an example, consider the composition `(tl . tl)`, where `tl` is typed `(s :: Sequence a) => s -> s`. Without the consistency requirement, the most general type for the composition would be `(s :: Sequence a, s :: Sequence b) => s -> s`. Composing `tl` n times would yield a type with n Sequence constraints, all but one being superfluous.

3 The Type System of Parametric Classes

This section presents our type system formally. We first define the abstract syntax of classes and types in the context of a small example language. We then explain formally what it means for a type to be an instance of a class. Based on these definitions, we define a non-deterministic type system with the same six rules as in [DM82], but with parametric type classes added. We claim that, in spite of its added generality, the system is actually simpler than previously published type systems for standard Haskell.

$$C \Vdash \alpha :: \gamma \quad (\alpha :: \{\ldots\gamma\ldots\} \in C)$$

$$\frac{C \Vdash C'}{C \Vdash \tau :: \gamma} \quad (\ulcorner \textbf{inst } C' \Rightarrow \tau :: \gamma \urcorner \in Ds)$$

$$\frac{C \Vdash \tau :: \gamma_1 \quad \ldots \quad C \Vdash \tau :: \gamma_n}{C \Vdash \tau :: \{\gamma_1, \ldots, \gamma_n\}} \quad (n \geq 0)$$

$$\frac{C \Vdash \tau_1 :: \Gamma_1 \quad \ldots \quad C \Vdash \tau_n :: \Gamma_n}{C \Vdash \{\tau_1 :: \Gamma_1, \ldots, \tau_n :: \Gamma_n\}} \quad (n \geq 0)$$

Figure 2: Inference Rules for Entailment

Syntax

The example language is a variant of Mini-Haskell [NS91], augmented with parameterized type classes. Its abstract syntax and types are shown in Figure 1. A parametric type class γ in this syntax has the form $c\,\tau$, where c is a class constructor, corresponding to a class in Haskell, and τ is a type. Classes with several parameters are encoded using tuple types, e.g. $c\,(\alpha, \beta)$. Parameterless classes are encoded using the unit type, e.g. $Eq\,()$.

The instance relationship between a type and a type class is denoted by an infix (::); the predicate $\tau' :: c\,\tau$ reads τ' is an instance of $c\,\tau$. We use the same infix notation to stipulate that a type be an instance of each member of a set of classes. In particular, bound variables in a type scheme are constrained by a (possibly empty) set of classes. The informal meaning of this is that, in $\forall \alpha :: \Gamma.\sigma$, α may only be instantiated to types that are instances of every class in Γ.

One simplification with respect to standard Haskell concerns the absence of a hierarchy on classes. The subclass/superclass relationship is instead modeled by class sets Γ. Consider for instance the class $Eq\,()$ of equality types in Haskell and its subclass $Ord\,()$ of ordered types. We can always represent $Ord\,()$ as a set of two classes, $\{Eq\,(), Ord'\,()\}$, where Ord' contains only operations $(<, \leq)$, which are defined in Ord but not in Eq. Translating all classes in a program in this way, we end up with sets over a flat domain of classes. This shows that we can without loss of generality disregard class hierarchy in the abstract syntax.

Instance Theories

In this subsection, we make precise when a type τ is an instance of a class set Γ, a fact which we will express $\tau :: \Gamma$. Clearly, the instance relation depends on the instance declarations Ds in a program. We let these declarations generate a theory whose sentences are instance judgments of the form $C \Vdash \tau :: \gamma$. An instance judgment is true in the theory iff it can be deduced using the inference rules in Figure 2.

Context

In these rules the *context* C is a set of instance assumptions $\alpha :: \Gamma$ (all α's in C are disjoint), where the class set Γ is subject to the consistency criterion mentioned in the previous section. When convenient, we will also regard a context as a finite mapping from type variables to class sets, i.e. $C\alpha = \Gamma$ iff $\alpha :: \Gamma \in C$. Thus the domain of C, $dom(C)$, is defined as the set of type variables α such that $\alpha :: \Gamma \in C$. As type classes can now contain parameters, we define the ***region*** of a context C,

$$reg(C) = \bigcup_{\alpha \in dom(C)} fv(C\alpha)$$

and the *closure* of C over a set of type variables, Δ, written $C^*(\Delta)$, as the least fixpoint of the equation

$$C^*(\Delta) = \Delta \cup C^*(reg(C|_\Delta)).$$

Here fv is the standard function that maps every type (scheme) to the set of free type variables that occur in it. By natural extension, it can be applied to class and set of classes.

We say C_1 is *contained* in C_2, written $C_1 \preceq C_2$, if $dom(C_1) \subseteq dom(C_2)$ and $C_1\alpha \subseteq C_2\alpha$ for each $\alpha \in dom(C_1)$. We write $C_1 \uplus C_2$ for the disjoint union of two contexts and $C\backslash_\alpha$ for restriction of a context C to all type variables in its domain other than α. As a consequence of the consistency criterion, the set union of two contexts is not well-defined but for *compatible* contexs. Two contexts C_1 and C_2 are called compatible, written $C_1 \bowtie C_2$, if for any class constructor c and type variable α, $\alpha \in dom(C_1) \cap dom(C_2)$, we have $\tau = \tau'$ whenever $c\ \tau \in C_1\alpha$ and $c\ \tau' \in C_2\alpha$.

A context C is called *closed* if $C^*(dom(C)) = dom(C)$, or, equivalently, $reg(C) \subseteq dom(C)$. A context C is called *acyclic* if all the type variables α, $\alpha \in dom(C)$, can be topologically sorted according to the order: $\alpha < \beta$ if $\alpha \in fv(C\beta)$. We shall restrict our discussion to only closed acyclic contexts in the remainder of the report.

Finally, we shall say that an object Π is *covered* by a context C if $fv(\Pi) \subseteq dom(C)$.

Constrained Substitution

A substitution is a map from type variables to types. The domain of a substitution S, $dom(S)$, is the set of type variables α such that $S\alpha \neq \alpha$. The region of a substitution can be defined just as the region of a context. As usual, the *composition* of substitutions S and R is denoted by $S \circ R$, or simply by juxtaposition SR.

In the following, we shall apply substitutions not only to types, but also to (sets of) classes and (sets of) instance predicates. On all of these, substitution is defined pointwise, i.e. it is a homomorphism on sets, class constructor application and (::). Since a context is a special form of an instance predicate set, substitutions can be applied to contexts. However, the result of such a substitution is in general not a context, as the left hand side α of an instance predicate $\alpha :: \Gamma$ can be mapped to a non-variable type. Our typing rules, on the other hand, require contexts instead of general predicate sets. Thus, we need a means to find a context that is a conservative approximation to a predicate set. We use the following definitions:

Definition. A *constrained substitution* is a pair (S, C) where S is a substitution and C is a closed context such that $C = SC$.

Definition. A constrained substitution (S, C) *preserves* a constrained substitution (S_0, C_0) if there is a substitution R such that $S = R \circ S_0$, C covers RC_0, and $C \Vdash RC_0$. We write in this case $(S, C) \preceq (S_0, C_0)$.

It is easy to show that $\preceq$ is a preorder.

Definition. A constrained substitution (S, C) is *most general* among those constrained substitutions that satisfy some requirement $\mathcal{R}$ if (S, C) satisfies $\mathcal{R}$, and, for any (S', C') that satisfies $\mathcal{R}$, $(S', C') \preceq (S, C)$.

Definition. A constrained substitution (S, C) is a *normalizer* of an instance predicate set P if $C \Vdash SP$.

To ensure the principal type property of our type system with parametric classes, we have to place the following requirements on the entailment relation $\Vdash$:

- **monotonicity:** for any contexts C and C', if $C' \preceq C$ then $C \Vdash C'$.
- **transitivity under substitution:** for any substitution S, contexts C and C', predicate set P, if $C \Vdash SC'$ and $C' \Vdash P$ then $C \Vdash SP$.
- **most general normalizers:** If a predicate set P has a normalizer then it has a most general normalizer.

From the viewpoint of type reconstruction, the first two requirements are needed to ensure that once established entailments are not falsified by later substitutions or additions to contexts. They follow directly from the inference rules in Figure 2. The last requirement ensures that there is a most general solution to an entailment constraint. To establish existence of most general normalizers, we have to place two restrictions on the instance declarations in a program:

(a) There is no instance declaration of the form $\ulcorner$**inst** $C \Rightarrow \alpha :: c\,\tau\urcorner$.

(b) For every pair of type and class constructor (κ, c), there is at most one instance declaration of the form $\ulcorner$**inst** $C \Rightarrow \kappa\,\tau' :: c\,\tau\urcorner$. Furthermore, τ' must be the unit type, or a possible empty tuple of *distinct* type variables and both $dom(C)$ and $fv(\tau)$ are contained in $fv(\tau')$.

Restriction (a) is part of current Haskell, and restriction (b) is a direct generalization of current Haskell's restriction to incorporate parametric type classes.

Theorem 3.1 If the instance declarations Ds of a program satisfy the restrictions (a) and (b), then $\Vdash$ admits most general normalizers.

Proof: See Section 5. ■

Typing Rules

Given an entailment relation $\Vdash$ between contexts and instance predicates, we now formalize a theory of typing judgments. Typing judgments are of the form $A, C \vdash e : \sigma$, where A is an assumption set of type predicates $x : \sigma$ (all x disjoint), C is a context, and e is an expression or a program. A typing judgment $A, C \vdash e : \sigma$ holds in the theory iff it can be deduced using the inference rules in Figures 3 and 4.

The rules in Figure 3 form a non-deterministic type system for expressions, along the lines of of the standard Hindley/Milner system [DM82]. One notable difference between this system and the

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

$$(\forall\!-elim) \qquad \frac{A, C \vdash e : \forall\alpha :: \Gamma.\sigma \qquad C \Vdash \tau :: \Gamma}{A, C \vdash e : [\alpha \mapsto \tau]\,\sigma}$$

$$(\forall\!-intro) \qquad \frac{A, C.\alpha :: \Gamma \vdash e : \sigma}{A, C \vdash e : \forall\alpha :: \Gamma.\sigma} \quad (\alpha \notin fv\ A \cup reg\ C)$$

$$(\lambda\!-elim) \qquad \frac{A, C \vdash e : \tau' \rightarrow \tau \qquad A, C \vdash e' : \tau'}{A, C \vdash e\ e' : \tau}$$

$$(\lambda\!-intro) \qquad \frac{A.x : \tau', C \vdash e : \tau}{A, C \vdash \lambda x.e : \tau' \rightarrow \tau}$$

$$(let) \qquad \frac{A, C \vdash e' : \sigma \qquad A.x : \sigma, C \vdash e : \tau}{A, C \vdash \mathbf{let}\ x = e'\ \mathbf{in}\ e : \tau}$$

Figure 3: Typing Rules for Expressions

$$(class) \qquad \frac{A.x : \forall_{fv\ \gamma}\forall\alpha :: \{\gamma\}.\sigma, C \vdash p : \tau}{A, C \vdash \mathbf{class}\ \alpha :: \gamma\ \mathbf{where}\ x : \sigma\ \mathbf{in}\ p : \tau}$$

$$(inst) \qquad \frac{A, C \vdash x : \forall\alpha :: \{\gamma\}.\sigma \qquad A, C \vdash e : [\alpha \mapsto \tau']\sigma \qquad A, C \vdash p : \tau}{A, C \vdash \mathbf{inst}\ C' \Rightarrow \tau' :: \gamma\ \mathbf{where}\ x = e\ \mathbf{in}\ p : \tau}$$

Figure 4: Typing Rules for Declarations

standard Hindley/Milner system is that the bound variable in a type scheme $\forall\alpha :: \Gamma.\sigma$ can be instantiated to a type τ only if we know from the context that $\tau :: \Gamma$ (rule $\forall\!-elim$). The second difference concerns rule ($\forall\!-intro$), where the instance predicate on the generalized variable α is "discharged" from the context to form the type scheme $\forall\alpha :: \Gamma.\sigma$. Here $fv(A)$ is the set of type variables that occur free in A, i.e., $\bigcup fv(\sigma)$ for each $(x : \sigma) \in A$.

One may obsevre that a signle context C is used in deriving both the type scheme for the **let**-definition and the type for the **let**-body. We could use two different contexts, but the following lemma justifies our use of a single context C in deriving the type for a **let**-expression. Let $bv(\sigma)$ be the set of bounded variables that occur in σ. We have:

Lemma 3.2 Given a typing judgement $A, C' \vdash e : \sigma$ and a context C such that $C' \preceq C$ and $dom(C) \cap bv(\sigma) = \emptyset$, we can construct another derivation for $A, C \vdash e : \sigma$.

Proof: By induction on the structure of $A, C' \vdash e : \sigma$. The only nontrivial case is when the last rule applied is ($\forall\!-intro$).

We have a derivation of the form :

$$\frac{A, C'.\alpha :: \Gamma \ \vdash \ e : \sigma'}{A, C' \ \vdash \ e : \forall \alpha :: \Gamma . \sigma'} \quad (\alpha \notin fv\ A \cup reg\ C')$$

Let $\sigma = \forall \alpha :: \Gamma . \sigma'$. It is clear that $dom(C.\alpha :: \Gamma) \cap bv(\sigma') = \emptyset$ given $dom(C) \cap bv(\sigma) = \emptyset$. Moreover, since $C' \preceq C$, we have $C'.\alpha :: \Gamma \preceq C.\alpha :: \Gamma$. Hence by induction we can constrcut the following derivation:

$$\cfrac{\cfrac{A, C'.\alpha :: \Gamma \ \vdash \ e : \sigma'}{A, C.\alpha :: \Gamma \ \vdash \ e : \sigma'} \ (induction)}{A, C \ \vdash \ e : \forall \alpha :: \Gamma . \sigma'} \ (let)$$

■

The rules in Figure 4 extend this system from expressions to programs. In rule (*class*), the overloaded identifier x is added to the assumption set. Rule (*inst*) expresses a compatibility requirement between an overloaded identifier and its instance expressions. These rules have to be taken in conjunction with the requirements (a), (b) on instance declarations listed in the last subsection. We say a program $p = Ds\ e$ has type scheme σ, iff Ds satisfies these requirements and generates an entailment relation $\Vdash$, and $A_0, \{\} \ \vdash \ p : \sigma$, for some given closed initial assumption set A_0.

The Instance Relation and Principal Type Schemes

A useful fact about Hindley/Milner type system is that when an expression e has a type, there is a *principal type scheme* which captures the set of all other types derivable for e thruogh the notion of *generic instances*. The remainder of this section introduces the definitions of generic instance and principal type schemes in our system.

Definition. A type scheme $\sigma' = \forall \alpha'_j :: \Gamma'_j . \tau'$ is a *generic instance* of a type scheme $\sigma = \forall \alpha_i :: \Gamma_i . \tau$ under a context C, if there exists a substitution S on $\{\alpha_i\}$ such that

1. α'_j is not free in σ,
2. $S\tau = \tau'$, and
3. $C \uplus \{\alpha'_j :: \Gamma'_j\} \Vdash S\alpha_i :: S\Gamma_i$.

We write in this case, $\sigma' \preceq_C \sigma$, and we drop the subscript in $\preceq_C$ if $C = \{\}$.

The definiton of $\preceq_C$ is an extension of the ordering relation defined in [DM82]. The only new requirement on instance entailment is needed for the extension with parametric type classes. It is easy to see that $\preceq_C$ defines a preorder on the set of type schemes.

We obtain a decision procedure for $\preceq_C$ by performing a match to establish (2), followed by a test of the validity of (3). We shall see in Section 5 how the latter can be done using the normalization function defined there.

Two type schemes σ and σ' are called *syntactically equivalent*, written $\sigma = \sigma'$, iff they are instances of each other, i.e., $\sigma \preceq \sigma'$ and $\sigma' \preceq \sigma$.

The following properties of ($\preceq_C$) are easily established:

Lemma 3.3 Let $(\sigma, C_1) = gen(\tau, A, C)$ and $(\sigma', C_1) = gen(\tau, A, C \uplus C')$. Then $\sigma' \preceq \sigma$ and $\sigma \preceq_{C'} \sigma'$.

Proof: Follows immediately from the definition. ■

Lemma 3.4 If $\sigma' \preceq_C \sigma$ and $C \preceq C'$ then $\sigma' \preceq_{C'} \sigma$.

Proof: Follows immediately from the definition, and noting that $\Vdash$ is monotonic. ■

The next lemma shows that the ordering on type schemes is preserved by constrained substitutions.

Lemma 3.5 If $\sigma' \preceq_C \sigma$ and $C' \Vdash SC$ then $S\sigma' \preceq_{C'} S\sigma$.

Proof: Let $\sigma = \forall\alpha_i :: \Gamma_i.\tau$ and $\sigma' = \forall\beta_j :: \Gamma'_j.\tau'$. By definition, there are types τ_i forming a substitution R, $R = [\alpha_i \mapsto \tau_i]$, such that

1. β_j are not free in σ,
2. $R\tau = \tau'$, and
3. $C \uplus \{\beta_j :: \Gamma'_j\} \Vdash R\{\alpha_i :: \Gamma_i\}$.

Without loss of generality we can assume that neither α_i nor β_j are involved in S. Hence

$$S\sigma = \forall\alpha_i :: S\Gamma_i.S\tau \quad \text{and} \quad S\sigma' = \forall\beta_j :: S\Gamma'_j.S\tau'.$$

Now let $R' = [\alpha_i \mapsto S\tau_i]$. We show that $S\sigma' \preceq_{C'} S\sigma$ through R'.

$$\begin{aligned} R'(S\tau) &= [\alpha_i \mapsto S\tau_i]S\tau \\ &= S([\alpha_i \mapsto \tau_i]\tau) \\ &= S\tau'. \end{aligned}$$

Moreover, since $C' \Vdash SC$ and β_j are not involved in S, we have

$$C' \uplus \{\beta_j :: S\Gamma'_j\} \Vdash S(C \uplus \{\beta_j :: \Gamma'_j\}).$$

By the transitivity under substitution of $\Vdash$ on (3), we get

$$C' \uplus \{\beta_j :: S\Gamma'_j\} \Vdash S(R\{\alpha_i :: \Gamma_i\}).$$

Using a similar argument, we have

$$R'(\{\alpha_i :: S\Gamma_i\}) = S(R\{\alpha_i :: \Gamma_i\}),$$

so

$$C' \uplus \{\beta_j :: S\Gamma'_j\} \Vdash R'(\{\alpha_i :: S\Gamma_i\}).$$

Finally, since β_j are not involved in S and do not occur free in σ, neither will they occur free in $S\sigma$. This completes the proof. ■

$$
\begin{array}{lll}
(\mathit{var}') & A, C \vdash' x : \tau \quad (x : \sigma \in A,\ \tau \preceq_C \sigma) & \\[2ex]
(\lambda-\mathit{elim}') & \dfrac{A, C \vdash' e : \tau' \to \tau \qquad A, C \vdash' e' : \tau'}{A, C \vdash' e\, e' : \tau} & \\[3ex]
(\lambda-\mathit{intro}') & \dfrac{A.x{:}\tau', C \vdash' e : \tau}{A, C \vdash' \lambda x.e : \tau' \to \tau} & \\[3ex]
(\mathit{let}') & \dfrac{A, C' \vdash' e' : \tau' \qquad A.x{:}\sigma, C \vdash' e : \tau}{A, C \vdash' \mathbf{let}\ x = e'\ \mathbf{in}\ e : \tau} & (\sigma, C'') = \mathit{gen}(\tau', A, C'),\ C'' \preceq C
\end{array}
$$

Figure 5: Determinstic Typing Rules for Expressions

We can easily extend the definition of instance relation to type asumption sets: For type assumption sets A' and A, $A' \preceq_C A$ if $\mathit{dom}(A') = \mathit{dom}(A)$ and $A'(x) \preceq_C A(x)$ for all x, $x \in \mathit{dom}(A)$.

With the definiton of ordering on type schemes, we can define the notion of *principal type schemes* in our system.

Definition. Given A, C, and e, we call σ a *principal type scheme* for e under A and C iff $A, C \vdash e : \sigma$, and for every σ', if $A, C \vdash e : \sigma'$ then $\sigma' \preceq_C \sigma$.

We shall develop an algorithm to compute principal type schemes in the following sections.

4 A Deterministic Type Inference System

We present a deterministic type inference system in this section. Compared to the typing rules in Section 3, the rules here are so formulated that the typing derivation for a given term e is uniquely detrmined by the syntactic structure of e, and hence are better suited to use in a type inference algorithm. We show that the system is equivalent to the previous one in terms of expressiveness and, in addition, has all the nice properties toward the construction of a type reconstruction algorithm.

Deterministic Typing Rules

The typings rules for the deterministic system are given in Figure 5. The rules $\forall-$*intro* and $\forall-$*elim* have been removed and typing judgements are now of the form $A, C \vdash' e : \tau$ where τ ranges over the set of type expressions as opposed to type schemes in the typing judgements of Section 3. Other major differences are that rule (var') instantiates a type scheme to a type according to the definition of generic instance and rule (let') use the generalization function, *gen*, to introduce type schemes.

The function *gen* takes as arguments a type scheme, an assumption set, and a context, and returns a generalized type scheme and a discharged context. It is defined by

```
gen $(\sigma, A, C) =$
    if $\exists\alpha \in dom(C)\backslash(fv\ A \cup reg\ C)$ then
        gen $(\forall\alpha :: C\alpha.\sigma, A, C_{\backslash\alpha})$
    else $(\sigma, C)$
```

In other words, instance assumptions in the given context, except those constraining type variables in the assumption set, are discharged and moved to form a more general type scheme in an order so that type variables are properly quantified. This is formalized in the following lemma:

Lemma 4.1 Let C be a closed and acyclic context that covers both type τ and type assumption set A. If $gen(\tau, A, C) = (\sigma, C')$ where $\sigma = \forall\alpha_i :: C\alpha_i.\tau$, then $\{\alpha_i\} \uplus C^*(fv\ A) = dom(C)$. In other words, $dom(C') = C^*(fv\ A)$.

Proof: Follows immediately from *gen*'s definition. ■

It is clear that the *gen* function is derived from the rule $(\forall-intro)$ used in the non-deterministic typing system. Hence we have the follwoing lemma:

Lemma 4.2 If $A, C \vdash e{:}\tau$ and $(\sigma, C') = gen(\tau, A, C)$ then $A, C' \vdash e : \sigma$.

Proof: By applying $(\forall-intro)$ repeatedly on $A, C \vdash e{:}\tau$.

Note that both the $(\forall-intro)$ rule and the *gen* function generalize a type scheme by discharging the instance assumptions contained in the context. Clearly, to derive a more genral typing, one would presume the context to constrain as many type variables as used in the inference process. This can always be achieved by assuming an empty class set for those unconstrained type variables and thus making them part of the context. Consequently, as a convention, in writing $A, C \vdash' e : \tau$, we shall assume that both A and τ are covered by C.

The following lemma shows that the set of free type variables in the assumption set is, roughly speaking, preserved under constrained substitution.

Lemma 4.3 Given a type assumption set A, contexts C and D, and a substitution S such that A is covered by C and SC is covered by D, if $D \Vdash SC$ then $\bigcup\{fv(S\alpha) \mid \alpha \in C^*(fv\ A)\} = D^*(fv\ SA)$.

Proof: We prove $\bigcup\{fv(S\alpha) \mid \alpha \in C^*(fv\ A)\} \subseteq D^*(fv\ SA)$ only; the proof for the other direction is quite similar and thus omitted.

If $\alpha \in (C)^*(fv\ A)$ then $\alpha \in C^k(fv\ A)$ for some k, $k \geq 0$. Thus an induction on k suffices:

$\mathbf{k = 0}$: Clearly, if $\alpha \in fv(A)$ then $fv(S\alpha) \subseteq fv(SA) \subseteq D^*(fv\ SA)$.

$\mathbf{k = n + 1}$: Suppose that $\beta \in fv(C\alpha)$ for some $\alpha \in C^n(fv\ A)$. By induction, $fv(S\alpha) \subseteq D^*(fv\ SA)$. There are four possibile cases depending on the effects of S on α and β:

- If $\alpha \notin dom(S)$ and $\beta \notin dom(S)$, then clearly $\beta \in fv(D\alpha)$ since $D \Vdash SC$. So, by induction, we have $\alpha \in D^*(fv\ SA)$ and hence $fv(S\beta) \subseteq D^*(fv\ SA)$.

- If $\alpha \notin dom(S)$ and $\beta \in dom(S)$, then clearly $D\alpha = S(C\alpha)$ since $D \Vdash SC$. So, $fv(S\beta) \subseteq fv(D\alpha)$. By induction, we have $\alpha \in D^*(fv\ SA)$ and hence $fv(S\beta) \subseteq D^*(fv\ SA)$.
- If $\alpha \in dom(S)$ and $\beta \notin dom(S)$, then there are two possibile cases according to the structure of $S\alpha$:
  - If $S\alpha = \rho$, *i.e.* $fv(S\alpha) = \{\rho\}$, then $\beta \in fv(D\rho)$ since $D \Vdash SC$. By induction, $\rho \in D^*(fv\ SA)$ and hence $fv(S\beta) \subseteq D^*(fv\ SA)$.
  - If $S\alpha = (\kappa\ \tau)$, then from $D \Vdash SC$ and our requirement on instance declaration, we have $\beta \in fv(\tau)$. But, since $fv(S\alpha) = fv(\tau)$, we have $fv(\tau) \subseteq D^*(fv\ SA)$ by induction and hence $fv(S\beta) \in D^*(fv\ SA)$.
- If $\alpha \in dom(S)$ and $\beta \in dom(S)$, then the four possible cases based on the structures of $S\alpha$ and $S\beta$ can be similarly reasoned as the previous case.

■

The next two lemmas describe the interaction between the *gen* function and constrained substitution. They are evry useful in later proofs.

Lemma 4.4 Let $(\sigma, C') = gen(\tau, A, C)$ and $(\sigma', D') = gen(S\tau, SA, D)$. If $D \Vdash SC$ then $\sigma' \preceq_{D'} S\sigma$.

Proof: Let $\sigma = \forall\alpha_i :: \Gamma_i.\tau$ and $\sigma' = \forall\alpha'_j :: \Gamma'_j.S\tau$. Now let β_i be new type variables, then

$$S\sigma = \forall\beta_i :: S[\alpha_i \mapsto \beta_i]\Gamma_i.S[\alpha_i \mapsto \beta_i]\tau.$$

To show that $\sigma' \preceq_{D'} S\sigma$, we choose the substitution $R = [\beta_i \mapsto S\alpha_i]$. Thus,

$$\begin{aligned} R(S[\alpha_i \mapsto \beta_i]\tau) &= [\beta_i \mapsto S\alpha_i]S([\alpha_i \mapsto \beta_i]\tau) \\ &= S[\beta_i \mapsto \alpha_i][\alpha_i \mapsto \beta_i]\tau \\ &= S\tau \end{aligned}$$

and similarly $R(S[\alpha_i \mapsto \beta_i]\Gamma_i) = S\Gamma_i$.

Now, the next thing to show is that $D' \uplus \{\alpha'_j :: \Gamma'_j\} \Vdash R(\{\beta_i :: S[\alpha_i \mapsto \beta_i]\Gamma_i\})$. But since

$$\begin{aligned} R(\{\beta_i :: S[\alpha_i \mapsto \beta_i]\Gamma_i\}) &= \{S\alpha_i :: S\Gamma_i\} \\ &= S\{\alpha_i :: \Gamma_i\}, \end{aligned}$$

and $D = D' \uplus \{\alpha'_j :: \Gamma'_j\}$ by Lemma 4.1, what we need to show is that $D \Vdash S\{\alpha_i :: \Gamma_i\}$. This follows directly from the given facts that $\{\alpha_i :: \Gamma_i\} \preceq C$ and $D \Vdash SC$.

Finally, since β_i are not free in σ', we have $\sigma' \preceq_{D'} S\sigma$. ■

The following lemma shows that, under certain conditions, the composition of generalization and constrained substitution can be commutative.

Lemma 4.5 Let $(\sigma, C') = gen(\tau, A, C)$. For any context C'' and substitution S such that C'' covers SC' and $C'' \Vdash SC'$, there exist a substitution R and a context D such that

$$RA = SA, \quad D \Vdash RC, \quad S\sigma = \sigma', \quad \text{and} \quad D' \preceq C''$$

where $(\sigma', D') = gen(R\tau, RA, D)$.

Proof: Let $\sigma = \forall\alpha_i :: \Gamma_i.\tau$ and $R = S[\alpha_i \mapsto \beta_i]$ where β_i are new type variables. From Lemma 4.1, none of α_i occurs in A, so we have $RA = SA$.

Next, let $D = R\{\alpha_i :: \Gamma_i\} \uplus C''|_{fv(SC')}$. From Lemma 4.1, $C = \{\alpha_i :: \Gamma_i\} \uplus C'$ and hence $RC' = SC'$. Now, since $C'' \Vdash SC'$, it follows that $D \Vdash RC$.

Third, we shall show that $S\sigma = \sigma'$. As β_i are new, we have $S\sigma = \forall\beta_i :: R\Gamma_i.R\tau$. Suppose that $\sigma' = \forall\rho_j :: \Gamma'_j.R\tau$, we need to show that $\{\beta_i\} = \{\rho_j\}$ to complete the proof. Given D's definition and the fact that β_i are new, it suffices to show that $D' = C''|_{fv(SC')}$. This we prove by considering $fv(S\alpha)$ for $\alpha \in dom(C')$.

From Lemma 4.1, we know that $dom(C') = C^*(fv\ A)$. Moreover, since $D \Vdash RC$ and $RA = SA$, we have $fv(S\alpha) \subseteq D^*(fv\ SA)$ for all $\alpha \in dom(C')$ by Lemma 4.3. In other words, none of the type variables in $fv(SC')$ can be generalized in $gen(R\tau, RA, D)$. Thus, $D' = C''|_{fv(SC')}$ and hence we have $S\sigma = \sigma'$ and $D' \preceq C''$. This completes the proof. ■

Equivalence of the two Systems

We now present a number of useful properties of the deterministic type system. They are useful not only in establishing the congruence of the two type systems, but also in investigating the relation between the type system and the type reconstruction algorithm.

We begin with the substitution lemma, which assures us that typing derivations are preserved under constrained substitution.

Lemma 4.6 (Substitution lemma) If $A, C \vdash' e : \tau$ and $C' \Vdash SC$ then $SA, C' \vdash' e : S\tau$.

Proof: By induction on the structure of the proof $A, C \vdash' e : \tau$. The only nontrivial cases are (var') and (let').

Case(var') : We have a derivation of the form

$$A, C \vdash' x : \tau \quad (x : \sigma \in A,\ \tau \preceq_C \sigma).$$

Without the loss of generality, we assume that S is safe for σ. Hence if $\sigma = \forall\alpha_i :: \Gamma_i.\tau'$ then $S\sigma = \forall\alpha_i :: S\Gamma_i.S\tau'$. Our goal is to find a instantiation substitution J such that $J(S\tau') = S\tau$ and $C' \Vdash J\{\alpha_i :: S\Gamma_i\}$.

Let $I = [\alpha_i \mapsto \tau_i]$ be the instantiation substitution such that $\tau = I\tau'$ and $C \Vdash I\{\alpha_i :: \Gamma_i\}$. Now define $J = S \circ I$. We first show that $J(S\tau') = S(I\tau')$. This we prove by showing that for each α occuring τ', $J(S\alpha) = S(I\alpha)$.

If α is bound in σ, *i.e.*, $\alpha = \alpha_j$ for some j, then

$$J(S\alpha_j) = J\alpha_j = S(I\alpha_j).$$

Otherwise α is free in σ, so

$$J(S\alpha) = S(I(S\alpha)) = S(S\alpha) = S\alpha = S(I\alpha),$$

since S is safe for α_i and α is not in $\{\alpha_i\}$. Hence $J(S\tau') = S\tau$ follows from $\tau = I\tau'$.

By a similar argument we can show that $J(S\Gamma_i) = S(I\Gamma_i)$. So

$$J\{\alpha_i :: S\Gamma_i\} = \{S(I\alpha_i) :: S(I\Gamma_i)\} = S(I\{\alpha_i :: \Gamma_i\}).$$

Then from $C \Vdash I\{\alpha_i :: \Gamma_i\}$ and $C' \Vdash SC$, we can infer that $C' \Vdash S(I\{\alpha_i :: \Gamma_i\})$. Therefore, $S\tau \preceq_{C'} S\sigma$ through J and hence $SA, C' \vdash' x : S\tau$.

Case(*let'*) : We have a derivation of the form

$$\frac{A, C_0 \vdash' e' : \tau \qquad A.x : \sigma, C \vdash' e : \tau'}{A, C \vdash' \textbf{let } x = e' \textbf{ in } e : \tau'} \quad (\sigma, C_1) = gen(\tau, A, C_0), \quad C_1 \preceq C$$

and $C' \Vdash SC$.

Clearly, we have $C' \Vdash SC_1$, since $C_1 \preceq C$. Now, applying Lemma 4.5 to $(\sigma, C_1) = gen(\tau, A, C_0)$, C' and S, we get a substitution R and a context C'' such that

$$RA = SA, \;\; C'' \Vdash RC_0, \;\; S\sigma = \sigma' \;\text{ and }\; \bar{C} \preceq C'$$

where $(\sigma', \bar{C}) = gen(R\tau, SA, C'')$.

Thus we can construct the derivation

$$\frac{\dfrac{\dfrac{A, C_0 \vdash' e' : \tau}{RA, C'' \vdash' e' : R\tau}\;(induction)}{SA, C'' \vdash' e' : R\tau}\;(SA = RA) \qquad \dfrac{\dfrac{A.x : \sigma, C \vdash' e : \tau'}{SA.x : S\sigma, C' \vdash' e : S\tau'}\;(induction)}{SA.x : \sigma', C' \vdash' e : S\tau'}\;(S\sigma = \sigma')}{SA, C' \vdash' \textbf{let } x = e' \textbf{ in } e : S\tau'}\;(let')$$

■

The next two lemmas express a form of monotonicity of typing derivations with respect to the context and the assumption set.

Lemma 4.7 If $A, C \vdash' e : \tau$ and $C \preceq C'$ then $A, C' \vdash' e : \tau$.

Proof: A straightforward induction on the structure of $A, C \vdash' e : \tau$.

Lemma 4.8 If $A', C \vdash' e : \tau$ and $A' \preceq_C A$ then $A, C \vdash' e : \tau$.

Proof: By induction on the structure of $A, C \vdash' e : \tau$. The only nontrivial case is (*let'*):

We have a derivation of the form:

$$\frac{A', C' \vdash' e' : \tau' \qquad A'.x : \sigma', C \vdash' e : \tau}{A', C \vdash' \textbf{let } x = e' \textbf{ in } e : \tau} \quad (\sigma', C'') = gen(\tau', A', C'), C'' \preceq C$$

Let $\sigma' = \forall \alpha_i :: \Gamma_i . \tau$ and $S = [\alpha_i \mapsto \beta_i]$ where β_i are new type variables. Now applying S to $A', C' \vdash' e' : \tau'$, we get $SA', \bar{C} \vdash' e' : S\tau'$ where $\bar{C} = SC'$. Since none of the α_i occurs in A', this isequivalent to $A', \bar{C} \vdash' e' : S\tau'$.

Next, let $D = C \backslash C''$. From Lemma 4.1 we know that $C' = C'' \uplus \{\alpha_i :: \Gamma_i\}$. So, $D \uplus \bar{C} = C \uplus \{\beta_i :: S\Gamma_i\}$ and hence $A' \preceq_{D \uplus \bar{C}} A$ by Lemma 3.4. Furthermore, let $(\sigma, \check{C}) = gen(S\tau', A, D \uplus \bar{C})$. Then, since none of the β_i appear free in A, we have $\sigma = \forall\beta_i :: S\Gamma_i . \forall\rho_j :: \Gamma'_j . S\tau'$ where $\{\rho_j :: \Gamma'_j\} \subseteq C$. So, clearly we have $\sigma' \preceq_C \sigma$ and $\tilde{C} \preceq C$. Thus we can construct the following derivation

$$\dfrac{\dfrac{\dfrac{A', C' \vdash' e' : \tau'}{A', D \uplus \bar{C} \vdash' e' : S\tau'}\;(\textit{Lemma } 4.7)}{A, D \uplus \bar{C} \vdash' e' : S\tau'}\;(\textit{induction}) \qquad \dfrac{A'.x : \sigma', C \vdash' e : \tau}{A.x : \sigma, C \vdash' e : \tau}\;(\textit{induction})}{A, C \vdash' \mathbf{let}\ x = e'\ \mathbf{in}\ e : \tau}\;(\textit{let}')$$

■

Now we can show that the deterministic system $\vdash'$ is equivalent to the non-deterministic system $\vdash$ in the following sense.

First, the deterministic system is sound with respect to the non-deterministic one.

Theorem 4.9 If $A, C \vdash' e : \tau$ then $A, C \vdash e : \tau$.

Proof: By induction on the structure of the proof $A, C \vdash' e : \tau$. The only interesting case is $(\textit{let}')$: We have a derivation of the form

$$\dfrac{A, C' \vdash' e' : \tau' \qquad A.x : \sigma, C \vdash' e : \tau}{A, C \vdash' \mathbf{let}\ x = e'\ \mathbf{in}\ e : \tau} \quad (\sigma, C'') = \textit{gen}(\tau', A, C'),\ \ C'' \preceq C$$

Without loss of generality, we can assume that $\textit{dom}(C') \cap \textit{dom}(C) = \textit{dom}(C'')$, which can be achieved by a suitable renaming of variables in $\textit{dom}(C)$. Hence, $\textit{dom}(C) \cap \textit{bv}(\sigma) = \emptyset$ and we can thus construct the following derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{A, C' \vdash' e' : \tau'}{A, C' \vdash e' : \tau'}\;(\textit{induction})}{A, C'' \vdash e' : \sigma}\;(\textit{Lemma } 4.2)}{A, C \vdash e' : \sigma}\;(\textit{Lemma } 3.2) \qquad \dfrac{A.x : \sigma, C \vdash' e : \tau}{A.x : \sigma, C \vdash e : \tau}\;(\textit{induction})}{A, C \vdash \mathbf{let}\ x = e'\ \mathbf{in}\ e : \tau}\;(\textit{let})$$

■

Second, for each typing derivation in the non-deterministic system we can always find a derivation in the deterministic system such that the inferred type scheme, after generalized by the *gen* function, is more general than the type scheme determined by the non-deterministic one.

Theorem 4.10 If $A, C \vdash e : \sigma$ then there is a context C', and a type τ such that $C \preceq C'$, $A, C' \vdash' e : \tau$ and $\sigma \preceq_C \sigma'$ where $(\sigma', C'') = \textit{gen}(\tau, A, C')$.

Proof: By induction on the structure of $A, P \vdash e : \sigma$.

Case$(\textit{var})$: We have a derivation of the form :

$$A, C \vdash x : \sigma \quad (x : \sigma \in A).$$

Let $\sigma = \forall\alpha_i :: \Gamma_i.\tau'$. Define C' and τ as follows:

$$C' = C \uplus S\{\alpha_i :: \Gamma_i\}, \qquad \tau = S\tau'$$

with S of the form $[\alpha_i \mapsto \beta_i]$ where β_i are new type variables. So, by definition, $C \preceq C'$ and $\tau \preceq_{C'} \sigma$. Thus

$$A, C' \vdash' x : \tau \qquad (x : \sigma \in A,\ \tau \preceq_{C'} \sigma)$$

Finally, since $C \preceq C'$, it is clear that $\sigma \preceq_C \sigma'$ where $(\sigma', C'') = gen(\tau, A, C')$.

Case$(\forall - elim)$: We have a derivation of the form :

$$\frac{A, C \vdash e : \forall\alpha :: \Gamma.\sigma \qquad C \Vdash \tau :: \Gamma}{A, C \vdash e : [\alpha \mapsto \tau]\sigma}$$

By induction,

$$A, C' \vdash' e : \tau'$$

for some τ' and C', with $C \preceq C'$, and $\forall\alpha :: \Gamma.\sigma \preceq_C \sigma'$ where $(\sigma', C'') = gen(\tau', A, C')$. Using the hypothesis $C \Vdash \tau :: \Gamma$ and transitivity of $\preceq_C$, we have

$$[\alpha \mapsto \tau]\sigma \preceq_C \forall\alpha :: \Gamma.\sigma \preceq_C \sigma'.$$

Case$(\forall - intro)$: We have a derivation of the form :

$$\frac{A, C.\alpha :: \Gamma \vdash e : \sigma}{A, C \vdash e : \forall\alpha :: \Gamma.\sigma} \ (\alpha \notin fv(A) \cup reg(C))$$

By induction,

$$A, C' \vdash' e : \tau$$

for some τ and C', with $(C.\alpha :: \Gamma) \preceq C'$ and $\sigma \preceq_{C.\alpha:\Gamma} \sigma'$ where $(\sigma', C'') = gen(\tau, A, C')$.

Since $\alpha \notin fv(A) \cup reg(C)$ and $(\alpha :: \Gamma) \in C'$, we know that $(\alpha :: \Gamma)$ will be generalized in $gen(\tau, A, C')$. It then follows that $\forall\alpha :: \Gamma.\sigma \preceq_C \sigma'$.

Case$(\lambda - elim)$: We have a derivation of the form :

$$\frac{A, C \vdash e : \tau' \to \tau \qquad A, C \vdash e' : \tau'}{A, C \vdash ee' : \tau}$$

By induction,

$$A, C_1 \vdash' e : \nu$$

for some ν and C_1, with $C \preceq C_1$ and $\tau' \to \tau \preceq_C \forall\alpha_i :: \Gamma_i.\nu$ where $(\forall\alpha_i :: \Gamma_i.\nu, C_0) = gen(\nu, A, C_1)$. Therefore there is a substitution R, $R = [\alpha_i \mapsto \tau_i]$, such that

$$C \Vdash R\{\alpha_i :: \Gamma_i\} \quad \text{and} \quad R\nu = \tau' \to \tau.$$

From Lemma 4.1, we know that $dom(C_0) = (C_1)^*(fv\ A) = C^*(fv\ A)$. So, $C_0 \preceq C$ and hence $C \Vdash RC_1$. Then using Lemma 4.6 on R, we can get a derivation

$$RA, C \vdash' e : R\nu.$$

Since none of α_i appear in A this is equivalent to

$$A, C \ \vdash' \ e : \tau' \rightarrow \tau.$$

By a similar argument, we get $C \Vdash SD_1$ and

$$A, C \ \vdash' \ e' : \tau'$$

for some context D_1 and substitution S, $S = [\beta_i \mapsto \nu_i]$.
Therefore, we have $A, C \ \vdash' \ ee' : \tau$. Finally, if $gen(\tau, A, C) = (\sigma', C')$ then obviously $\tau \preceq_C \sigma'$.

Case($\lambda-intro$) : We have a derivation of the form :

$$\frac{A.x{:}\tau', C \ \vdash \ e : \tau}{A, C \ \vdash \ \lambda x.e : \tau' \rightarrow \tau}$$

By induction,

$$A.x{:}\tau', C_1 \ \vdash' \ e : \nu$$

for some ν and C', with $C \preceq C_1$ and $\tau \preceq_C \forall\alpha_i :: \Gamma_i.\nu$ where $(\forall\alpha_i :: \Gamma_i.\nu, C_0) = gen(\nu, A.x{:}\tau', C_1)$.
Therefore, there is a substitution S, $S = [\alpha_i \mapsto \tau_i]$, such that

$$C \Vdash S\{\alpha_i :: \Gamma_i\} \quad \text{and} \quad S\nu = \tau.$$

From $C \preceq C_1$, we know that $C \bowtie C_0$. Furthermore, as $C_1 = \{\alpha_i :: \Gamma_i\} \uplus C_0$ by Lemma 4.1, we have $C \cup C_0 \Vdash SC_1$. Then, using Lemma 4.6 on S, we get a derivation

$$S(A.x{:}\tau'), C \cup C_0 \ \vdash' \ e : S\nu.$$

Since none of α_i appears in $A.x{:}\tau'$ this is equivalent to

$$A.x{:}\tau', C \cup C_0 \ \vdash' \ e : \tau.$$

Hence

$$A, C \cup C_0 \ \vdash' \ \lambda x.e : \tau' \rightarrow \tau.$$

Finally, if $gen(\tau' \rightarrow \tau, A, C \cup C_0) = (\sigma', C')$ then obviously $\tau' \rightarrow \tau \preceq_C \sigma'$.

Case(let) : We have a derivation of the form :

$$\frac{A, C \ \vdash \ e' : \sigma \qquad A.x{:}\sigma, C \ \vdash \ e : \tau}{A, C \ \vdash \ \textbf{let } x = e' \textbf{ in } e : \tau}$$

We shall assume that A, σ, and τ are all covered by C, for we can always place an empty class set on unconstrained type variables.

By induction,

$$A, C_1 \ \vdash' \ e' : \tau_1$$

for some τ_1 and C_1 with $C \preceq C_1$ and $\sigma \preceq_C \sigma_1$ where $(\sigma_1, C_1') = gen(\tau_1, A, C_1)$. Again, by induction,

$$A.x{:}\sigma, C_2 \ \vdash' \ e : \tau_2$$

for some τ_2 and C_2 with $C \preceq C_2$ and $\tau \preceq_C \sigma_2$ where $(\sigma_2, C_2') = gen(\tau_2, A.x:\sigma, C_2)$.

Without loss of generality, we can assume that $dom(C_1) \cap dom(C_2) = dom(C)$. This can always be achieved by a suitable renaming of the variables in C_2. Now, from Lemma 4.1, we have $dom(C_1') = (C_1)^*(fv\ A)$. But obviously $(C_1)^*(fv\ A) = C^*(fv\ A)$. Hence $C_1' \preceq C_2$. So we can construct the following derivation

$$\dfrac{A, C_1 \vdash' e : \tau_1 \qquad \dfrac{A.x:\sigma, C_2 \vdash' e : \tau_2}{A.x:\sigma_1, C_2 \vdash' e : \tau_2}\ (\mathit{Lemma}\ 4.8)}{A, C_2 \vdash' \mathbf{let}\ x = e'\ \mathbf{in}\ e : \tau_2}\ (\mathit{let}')$$

Finally, we need to show that if $gen(\tau_2, A, C_2) = (\sigma_3, C_3')$ then $\tau \preceq_C \sigma_3$. Note that since $fv(\sigma) \subseteq dom(C)$ and $C \preceq C_2$, we have $fv(\sigma) \subseteq dom(C_2)$ and hence $\sigma_2 \preceq_C \sigma_3$ by Lemma 3.3. Thus, $\tau \preceq_C \sigma_3$ follows immediately.

This completes the proof. ■

5 Context-Preserving Unification

Type reconstruction usually relies on unification to compute most general types. One consequence of rule ($\forall$−*elim*) is that the well-known syntactic unification algorithm of Robinson [Rob65] cannot be used since not every substitution of variables to types satisfies the given instance constraints. Nipkow and Snelting have shown that order-sorted unification can be used for reconstructing of types in Haskell [NS91], but it is not clear how to extend their result to parametric type classes. We show in this section that algorithm *mgu*, shown in Figure 6, yields the most general context-preserving unifier of two types.

Function *mgu* takes two types and returns a transformer on constrained substitutions. The application $mgu\ \tau_1\ \tau_2\ (S_0, C_0)$ returns a most general constrained substitution that unifies the types τ_1 and τ_2 and preserves (S_0, C_0), if such a substitution exists. The algorithm is similar to the one of Robinson, except for the case $mgu\ \alpha\ \tau\ (S_0, C_0)$, where α may be substituted to τ only if τ can be shown to be an instance of $C_0\alpha$. This constraint translates to an application of the subsidary function *mgn* to τ and $C\alpha$.

The call $mgn\ \tau\ \Gamma\ (S, C)$ computes a most general (S, C)-preserving normalizer of $\{\tau :: \Gamma\}$, provided one exists. This is accomplished by normalizing $\tau :: \gamma$ for each $\gamma \in \Gamma$ through mgn'. In mgn', it may in turn call $mgu\ \tau\ \tau'$ to solve a subproblem of the form $C.\alpha :: (\{c\,\tau'\} \cup \Gamma') \Vdash \alpha :: c\,\tau$. The unification is required since all class constructors in a class set are pairwise disjoint. Thus, the above entailment has $\tau = \tau'$ as a logical consequence. The other call to *mgu* is made when solving the entailment $C \Vdash \kappa\,\tau' :: c\,\tau$. Since there is at most one instance declaration for each (κ, c) pair, it either fails or finds the proper instance declaration. To instantiate the instance declaration $\ulcorner C' \Rightarrow \kappa\,\tilde{\tau}' :: c\,\tilde{\tau}\urcorner$, the standard pattern matching operation *match* is first called. It takes the pair of types $(\tilde{\tau}', \tau')$ and returns a most general substitution S' such that $S'\tilde{\tau}' = \tau'$. Then, due to the at-most-one restriction, it calls *mgu* to unify τ and $S'\tilde{\tau}$. Finally, it instantiates the declared context C' and recursively normalize it.

Note that in an instance declaration, $\ulcorner C' \Rightarrow \kappa\,\tilde{\tau}' :: c\,\tilde{\tau}\urcorner$, the type $\tilde{\tau}'$ is either a unit type or a tuple of distinct type variables. Besides, both $fv(\tilde{\tau})$ and $dom(C')$ are contained in $fv(\tilde{\tau}')$. Thus, it suffices to use the standard matching operation to instantiate an instance declaration, and no new type variables will be introduced in the normalization process.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{mgu : \tau \to \tau \to S \times C \to S \times C} \\
\multicolumn{3}{l}{mgn : \tau \to \Gamma \to S \times C \to S \times C} \\
mgu\ \tau_1\ \tau_2\ (S, C) & = & mgu'\ (S\tau_1)\ (S\tau_2)\ (S, C) \\
mgu'\ \alpha\ \alpha & = & id_{S \times C} \\
mgu'\ \alpha\ \tau\ (S, C) \mid \alpha \notin fv(\tau) & = & mgn\ \tau\ (C\alpha)\ ([\alpha \mapsto \tau] \circ S, [\alpha \mapsto \tau] C\backslash_{\alpha}) \\
mgu'\ \tau\ \alpha\ (S, C) & = & mgu\ \alpha\ \tau\ (S, C) \\
mgu'\ ()\ () & = & id_{S \times C} \\
mgu'\ \kappa\,\tau\ \kappa\,\tau'\ (S, C) & = & mgu\ \tau\ \tau'\ (S, C) \\
mgu'\ (\tau_1 \times \tau_2)\ (\tau_1' \times \tau_2') & = & (mgu\ \tau_1\ \tau_1') \circ (mgu\ \tau_2\ \tau_2') \\
mgu'\ (\tau_1 \to \tau_2)\ (\tau_1' \to \tau_2') & = & (mgu\ \tau_1\ \tau_1') \circ (mgu\ \tau_2\ \tau_2') \\
mgn\ \tau\ \{\} & = & id_{S \times C} \\
mgn\ \tau\ \{\gamma\}\ (S, C) & = & mgn'\ (S\tau)\ (S\gamma)\ (S, C) \\
mgn\ \tau\ (\Gamma_1 \cup \Gamma_2) & = & (mgn\ \tau\ \Gamma_1) \circ (mgn\ \tau\ \Gamma_2) \\
mgn'\ \alpha\ c\,\tau\ (S, C) & = & \textbf{if } \exists \tau'.(c\,\tau' \in C\alpha)\ \textbf{then } mgu\ \tau\ \tau'\ (S, C) \\
 & & \textbf{else } (S, C[\alpha \mapsto C\alpha \cup \{c\,\tau\}]) \\
\multicolumn{3}{l}{mgn'\ \kappa\,\tau'\ c\,\tau\ (S, C) \mid \exists \ulcorner \textbf{inst } C' \Rightarrow \kappa\,\tilde{\tau}' :: c\,\tilde{\tau} \urcorner \in Ds} \\
 & = & \textbf{let } S' \ = \ match\ \tilde{\tau}'\ \tau' \\
 & & \quad (S'', C'') \ = \ mgu\ \tau\ (S'\tilde{\tau})\ (S, C) \\
 & & \quad \{\tau_1 :: \Gamma_1, \ldots, \tau_n :: \Gamma_n\} \ = \ S'C' \\
 & & \textbf{in } (mgn\ \tau_1\ \Gamma_1\ (\ \ldots\ (mgn\ \tau_n\ \Gamma_n\ (S'', C'')))) \\
\multicolumn{3}{l}{\text{(and similarly for } \to, \times, ())}
\end{array}
$$

Figure 6: Unification and Normalization Algorithms

Consequently, we have the following lemma:

Lemma 5.1 Given a constrained substitution (S, C) and types τ_1, τ_2 such that both $S\tau_1$ and $S\tau_2$ are covered by C, if $mgu\ \tau_1\ \tau_2\ (S, C) = (S', C')$, then there exists a substitution R such that $S' = R \circ S$ and both $dom(R)$ and $reg(R)$ are contained in $C^*(fv(S\tau_1) \cup fv(S\tau_2))$.

Moreover, in the course of computing the unifier, whenever the substitution is extended with $[\alpha \mapsto \tau]$ in mgu', the context constraints on α are discharged from C and all occurrences of α in C are replaced by τ. By doing so we obtain the following lemma:

Lemma 5.2 Given a constrained substitution (S, C) and types τ_1, τ_2 such that both $S\tau_1$ and $S\tau_2$ are covered by C, if $mgu\ \tau_1\ \tau_2\ (S, C) = (S', C')$, then (S', C') is a constrained substitution and $dom(S') \setminus dom(S) = dom(C) \setminus dom(C')$.

In order to prove the termination of the unification algorithm, we need to define some metric functions on sets and types. We write $|s|$ for the number of elements in set s, and, overloading the $|\ |$ operation, define $|\tau|$ to be the number of symbols in τ :

$|\alpha| = 1$
$|()| = 1$
$|\kappa\, \tau| = 1 + |\tau|$
$|(\tau \times \tau')| = 1 + |\tau| + |\tau'|$
$|(\tau \rightarrow \tau')| = 1 + |\tau| + |\tau'|$

We now proceed to prove that call to *mgu* with proper arguments will always terminate.

Lemma 5.3 (Termination of *mgu*) Given a constrained substitution (S, C), types $\bar{\tau}_1$ and $\bar{\tau}_2$ such that both $S\bar{\tau}_1$ and $S\bar{\tau}_2$ are covered by C, then *mgu* $\bar{\tau}_1\ \bar{\tau}_2\ (S, C)$ either fails or terminates.

Proof: We can think of the four functions, *mgu*, *mgu'*, *mgn*, *mgn'*, as mutually recursively defined over the tuple $(\bar{\tau}_1, \bar{\tau}_2, \bar{\tau}_3, \gamma, \Gamma, (S, C))$, where $\bar{\tau}_1$ and $\bar{\tau}_2$ are paired with *mgu* and *mgu'*, $\bar{\tau}_3$ and Γ with *mgn*, and $\bar{\tau}_3$ and γ with *mgn'*. The termination of them can be proved by defining a *degree* over the tuple:

$$(|dom(C)|, |\bar{\tau}_3|, |\Gamma|, |\bar{\tau}_2|, |\bar{\tau}_1|).$$

We order degrees lexicographically and argue that each recursive call reduces the degree.

From Lemma 5.2, we know that C is never enlarged by these functions and whenever S is extended, C will be correspondingly diminished. This fact explains why we put $|dom(C)|$ as the first component of the degree and is crucial to the following argument.

Calls to *mgu* are unfolded to calls to *mgu'*. In *mgu'*, the recursive call to *mgn* diminishes $|dom(C)|$ and the recursive calls to *mgu* are supplied with subcomponents of $\bar{\tau}_1$, $\bar{\tau}_2$ and a possibly diminished C. For instance, the call *mgu'* $(\tau_1 \times \tau_2)\ (\tau_1' \times \tau_2')$ will be expanded to $(mgu\ \tau_1\ \tau_1') \circ (mgu\ \tau_2\ \tau_2')$. Then, we could get from *mgu* $\tau_2\ \tau_2'\ (S, C)$ some (S', C') such that $|S'\tau_1| > |(\tau_1 \times \tau_2)|$ when unfolding *mgu* $\tau_1\ \tau_1'\ (S', C')$. In that case, however, we would have $|dom(C')| < |dom(C)|$ by Lemma 5.2 and hence the degree would still be reduced. So we know that the degree is reduced in each case.

Function *mgn* recursively calls itself with a smaller class set and calls *mgn'* only when a singleton class set is reached. In *mgn'*, the calls to *mgu* are unfolded, and, from the above argument for *mgu'*, we know that the degree will be reduced. Moreover, in recursively normailzing the context of an instance declaration, the series of recursive calls to *mgn* are passed the subcomponents of $\bar{\tau}_3$ and a possibly diminished C, thus the degree will be reduced as the normalization process proceeds.

Since there is no infinite decreasing sequence of tuples of natural number, it follows that *mgu* $\bar{\tau}_1\ \bar{\tau}_2\ (S, C)$ either terminates or fails. ■

Lemma 5.4 (Soundness of *mgu* and *mgn*)

1. If *mgn* $\bar{\tau}_3\ \Gamma\ (S, C) = (S', C')$ then $(S', C') \preceq (S, C)$ and $C' \Vdash S'(\bar{\tau}_3 :: \Gamma)$.
2. If *mgu* $\bar{\tau}_1\ \bar{\tau}_2\ (S, C) = (S', C')$ then $S'\tau_1 = S'\tau_2$ and $(S', C') \preceq (S, C)$.

Proof: We use induction on the degree of the parameters.

If $|dom(C)| = 0$ then there is no type variable involved and so the lemma is vacuously true.

We now proceed to show that the lemma is also true for $|dom(C)| = n + 1$.

1. *mgn* $\bar{\tau}_3\ \Gamma\ (S, C)$:

   Induction base ($S\bar{\tau}_3 = \alpha$), consider *mgn* $\alpha\ \Gamma\ (S, C), \alpha \in dom(C)$:

   (a) $\Gamma = \{\}$: Obvious.

   (b) Consider *mgn'* $\alpha\ (c\,\tau)\ (S, C)$. There are two possibilities:

      i. $\exists \tau'.(c\,\tau' \in C\alpha)$:
      We unfold the call *mgu* $\tau\ \tau'\ (S, C)$. The interesting cases are:

      A. $\tau = \beta$:
      Let $(S', C') =$ *mgn* $\tau'\ (C\beta)\ ([\beta \mapsto \tau'] \circ S, [\beta \mapsto \tau']C\backslash_\beta)$. Then, by induction on $|dom(C\backslash_\beta)|$ we get

      $$C' \Vdash (R' \circ ([\beta \mapsto \tau'])C\backslash_\beta, \quad \text{and} \quad C' \Vdash S'(\tau' :: (C\beta))$$

      where $S' = R' \circ ([\beta \mapsto \tau'] \circ S)$. But, since $S'\beta = R'\tau'$ and $S'(C\beta) = R'(C\beta)$, by combining the above two entailments, we get $C' \Vdash (R' \circ [\beta \mapsto \tau'])C$ and hence $(S', C') \preceq (S, C)$. Moreover, since $\alpha \in dom(C)$, we have $C' \Vdash S'(\alpha :: C\alpha)$. Hence from $(c\,\tau') \in C\alpha$ and $S'(c\,\tau) = S'(c\,\tau')$ it is clear that $C' \Vdash S'(\bar{\tau}_3 :: c\,\tau)$.

      B. $\tau = \tau_1 \times \tau_2$ and $\tau' = \tau_1' \times \tau_2'$:
      Let $(S', C') =$ *mgu* $\tau_2\ \tau_2'\ (S, C)$. Then, by induction on $|\tau|$ and $|\tau'|$ using (2), we get

      $$C' \Vdash R'C \quad \text{and} \quad S'\tau_2 = S'\tau_2'$$

      where $S' = R' \circ S$. Now consider the second recursive call *mgu* $\tau_1\ \tau_1'\ (S', C')$: If $S' = S$ then $C' = C$ by Lemma 5.2. So, again, by induction on $|\tau|$ and $|\tau'|$ using (2), let $(S'', C'') =$ *mgu* $\tau_1\ \tau_1'\ (S', C')$, then

      $$C'' \Vdash R''(R'C) \text{ and } S''(\tau_1 \times \tau_2) = S''(\tau_2 \times \tau_2')$$

      where $S'' = R'' \circ S'$, then By an argument similar to the one used in the previous case, we can easily show that $C'' \Vdash S''(\bar{\tau}_3 :: c\,\tau)$. Otherwise, we have $|dom(C')| < |dom(C)|$ by Lemma 5.2 and hence we can get the same result by induction on $|dom(C)|$ using (2).

      ii. Otherwise it is clear that $(S, C[\alpha \mapsto C\alpha \cup \{c\,\tau\}])$ satisfies the requirement.

   (c) $\Gamma = \Gamma_1 \cup \Gamma_2$: By a straightforward induction on Γ and possibly $|dom(C)|$.

   Induction step ($S\bar{\tau}_3 = \kappa\,\tau'$), Consider *mgn'* $(\kappa\,\tau')\ \Gamma\ (S, C)$.

   (a) $\Gamma = \{\}$: Obvious.

   (b) $\Gamma = \{(c\,\tau)\}$: Consider *mgn'* $(\kappa\,\tau')\ (c\ \tau)\ (S, C)$. Unfolding the call to *mgu*, and by a similar inductive argument used in the above, we have $(S'', C'') \preceq (S, C)$ and $S''\tau = S''(S'\bar{\tau})$. Then, since for all i, $|\tau_i'| < |\kappa\,\tau'|$ and $|dom(C'')| \leq |dom(C)|$, we can get the result by a series of straightforward induction.

   (c) $\Gamma = \Gamma_1 \cup \Gamma_2$: By a straightforward induction on Γ and possibly $|dom(C)|$.

2. *mgu* $\tau_1\ \tau_2\ (S, C)$: By a similar induction, and using (1).

■

Lemma 5.5 (Completeness of *mgu* and *mgn*)

1. Given any constrained substitution $(\bar{S}, \bar{C})$, if $(\bar{S}, \bar{C}) \preceq (S, C)$ and $\bar{S}\bar{\tau}_1 = \bar{S}\bar{\tau}_2$, then *mgu* $\bar{\tau}_1\ \bar{\tau}_2\ (S, C)$ computes a constrained substitution (S', C'), with $S'\bar{\tau}_1 = S'\bar{\tau}_2$ and $(\bar{S}, \bar{C}) \preceq (S', C') \preceq (S, C)$.
2. Given any constrained substitution $(\bar{S}, C)$, if $(\bar{S}, \bar{C}) \preceq (S, C)$ and $\bar{C} \Vdash \bar{S}(\bar{\tau}_3 :: \Gamma)$, then *mgn* $\bar{\tau}_3\ \Gamma\ (S, C)$ computes a constrained substitution (S', C'), with $C' \Vdash S'(\bar{\tau}_3 :: \Gamma)$ and $(\bar{S}, \bar{C}) \preceq (S', C') \preceq (S, C)$.

Proof: We use induction on the degree of the parameters.

If $|dom(C)| = 0$, then there is no type variable involved and so the lemma is vacuously true.

We now proceed to to show that the lemma is also true for $|dom(C)| = n + 1$.

1. *mgu* $\bar{\tau}_1\ \bar{\tau}_2\ (S, C)$: By induction on $|S\bar{\tau}_1|$ and $|S\bar{\tau}_2|$.
   (a) *mgu'* $\alpha\ \tau\ (S, C)$:
   Since $\bar{S}$ unifies α and τ, we can write $\bar{S} = \bar{R} \circ ([\alpha \mapsto \tau] \circ S)$. Also, from the assumption $(\bar{S}, \bar{C}) \preceq (S, C)$, we know that
   $$\bar{C} \Vdash \bar{R}(\tau :: C\alpha) \text{ and } \bar{C} \Vdash \bar{R}([\alpha \mapsto \tau]C\backslash\alpha).$$
   Now let $(S', C') = mgn\ \tau\ (C\alpha)\ ([\alpha \mapsto \tau] \circ S, [\alpha \mapsto \tau]C\backslash_\alpha)$. Then, by induction on $|dom(C)|$ using (2), we get
   $$(\bar{S}, \bar{C}) \preceq (S', C') \preceq ([\alpha \mapsto \tau] \circ S, [\alpha \mapsto \tau]C\backslash_\alpha)$$
   where $S' = R' \circ ([\alpha \mapsto \tau] \circ S)$. To complete the proof in this case, we need to show that $C' \Vdash R'(\tau :: C\alpha)$. But this follows directly from (1) of Lemma 5.4. So, we have $C' \Vdash (R' \circ [\alpha \mapsto \tau])C$ and hence $(\bar{S}, \bar{C}) \preceq (S', C') \preceq (S, C)$.
   (b) *mgu'* $(\kappa\ \tau_1)\ (\kappa\ \tau_2)\ (S, C)$: By a straightforward induction.
   (c) *mgu'* $(\tau_1 \times \tau_2)\ (\tau_1' \times \tau_2')\ (S, C)$: By a straightforward induction.
2. *mgn* $\bar{\tau}_3\ \Gamma\ (S, C)$:
   Induction base ($S\bar{\tau}_3 = \alpha$): Consider *mgn* $\alpha\ \Gamma\ (S, C)$:
   (a) $\Gamma = \{\}$: Obvious.
   (b) Consider *mgn'* $\alpha\ (c\ \tau)\ (S, C)$. There are two possibilities:
      i. $\exists\tau'.(c\ \tau' \in C\alpha)$:
      We unfold the call *mgu* $\tau\ \tau'\ (S, C)$. The interesting cases are:
      - $\tau = \beta$:
        In order to induction on $|dom(C)|$, we need to verify that
        $$(\bar{S}, \bar{C}) \preceq ([\beta \mapsto \tau'] \circ S, [\beta \mapsto \tau']C\backslash_\beta) \text{ and } \bar{C} \Vdash \bar{S}(\tau' :: C\beta).$$
        Suppose that $\bar{S} = \bar{R} \circ S$, then, from the assumptions, we have
        $$\bar{C} \Vdash \bar{R}(\alpha :: c\ \beta) \text{ and } \bar{C} \Vdash \bar{R}C.$$

```
$tp\ (x, A, S, C)$                              $=$ $inst\ (S(Ax), S, C)$
$tp\ (e_1\ e_2, A, S, C)$                       $=$ let $(\tau_1, S_1, C_1) = tp\ (e_1, A, S, C)$
                                                        $(\tau_2, S_2, C_2) = tp\ (e_2, A, S_1, C_1)$
                                                        $\alpha$ a fresh type variable
                                                        $(S_3, C_3) = mgu\ \tau_1\ (\tau_2 \rightarrow \alpha)\ (S_2, C_2.\alpha :: \{\})$
                                                    in $(S_3\alpha, S_3, C_3)$
$tp\ (\lambda x.e, A, S, C)$                    $=$ let $\alpha$ a fresh type variable
                                                        $(\tau_1, S_1, C_1) = tp\ (e_1, A.x{:}\alpha, S, C.\alpha :: \{\})$
                                                    in $(S_1\alpha \rightarrow \tau_1, S_1, C_1)$
$tp\ (\textbf{let}\ x = e_1\ \textbf{in}\ e_2, A, S, C)$ $=$ let $(\tau_1, S_1, C_1) = tp\ (e_1, A, S, C)$
                                                        $(\sigma, C_2) = tpgen\ (\tau_1, S_1A, C_1, C)$
                                                    in $tp\ (e_2, A.x{:}\sigma, S_1, C_2)$
where
$inst\ (\forall\alpha :: \Gamma.\sigma, S, C)$  $=$ let $\beta$ a fresh type variable
                                                    in $inst\ ([\alpha \mapsto \beta]\ \sigma, S, C.\beta :: \Gamma)$
$inst\ (\tau, S, C)$                            $=$ $(\tau, S, C)$
$tpgen\ (\sigma, A, C, C')$                     $=$ if $\exists\alpha \in dom(C)\backslash(fv(A) \cup reg(C) \cup dom(C'))$then
                                                        $tpgen\ (\forall\alpha :: C\alpha.\sigma, A, C\backslash_\alpha, C')$
                                                    else $(\sigma, C)$
```

Figure 7: Type Reconstruction Algorithm

6 Type Reconstruction

This section discusses type reconstruction. Having developed the unification algorithm needed for parametric type classes, we can proceed to present a type reconstruction algorithm, and state its soundness and completeness with respect to the inference rules given in Section 3 using those rules in Section 4 and the equivalence result established therein. As a corollary of these results, we obtain a prinicpal type scheme property of our system analogous to the one in [DM82]. The type reconstruction algorithm has been implemented in the Yale Haskell compiler. Its size and complexity compare favorably to the type reconstruction parts of our prior Haskell compiler.

An algorithm for type reconstruction is shown in Figure 7.[2] Function *tp* takes as arguments an expression, an assumption set, and an initial constrained substitution, and returns a type and a final constrained substitution. The function is straightforwardly extended to programs. The remainder of this section establishes the correspondence between *tp* and the type system of Section 4 and, thereby, that of Section 3.

We need the following lemmas to establish the soundness and completeness of our algorithm. We

[2]This is actually a simplification of the real algorithm becuase we can get a cyclic context after the call to unification function and thus violate our restriction on contexts. So what is missing here is a clique-detection algorithm, which is simply a variant of occur checking. We omit it here for simplicity.

Now, since we know that $(c\,\tau') \in C\alpha$ and $\bar{C} \Vdash \bar{R}(\alpha :: c\ \tau')$, then, due to our requirements on instance declarations, we have $\bar{R}\beta = \bar{R}\tau'$. Moreover, as $\beta \in dom(C)$, we have $\bar{C} \Vdash \bar{R}(\beta :: C\beta)$. It then follows that $\bar{C} \Vdash \bar{S}(\tau' :: C\beta)$. Now let $\bar{R} = \tilde{R} \circ [\beta \mapsto \tau']$. Then

$$\bar{S} = \tilde{R} \circ ([\beta \mapsto \tau'] \circ S) \text{ and } \bar{C} \Vdash \tilde{R}([\beta \mapsto \tau']C\backslash_\beta).$$

In other words,

$$(\bar{S}, \bar{C}) \preceq ([\beta \mapsto \tau'] \circ S, [\beta \mapsto \tau']C\backslash_\beta).$$

Next, let $(S', C') = mgn\ \tau'\ (C\beta)\ ([\beta \mapsto \tau'] \circ S, [\beta \mapsto \tau']C\backslash_\beta)$. and $S' = R' \circ ([\beta \mapsto \tau'] \circ S)$. Then by induction on $|dom(C)|$, we get

$$(\bar{S}, \bar{C}) \preceq (S', C'),\quad C' \Vdash (R' \circ [\beta \mapsto \tau'])C\backslash_\beta, \text{ and } C' \Vdash S'(\tau' :: C\beta).$$

But $S'\beta = S'\tau'$, so

$$C' \Vdash (R' \circ [\beta \mapsto \tau'])C.$$

Hence $(S', C') \preceq (S, C)$ and $C' \Vdash R'(\alpha :: (c\ \beta))$.

- $\tau = \tau_1 \times \tau_2$ and $\tau' = \tau_1' \times \tau_2'$:
  Note that if $\bar{S} = \bar{R} \circ S$ then $\bar{R}\tau = \bar{R}\tau'$, since $\bar{C} \Vdash \bar{R}(\alpha :: (c\ \tau))$ and $(c\ \tau') \in C\alpha$. Hence the result follows from a straightforward induction using 1.

ii. Otherwise it is clear that $(S, C[\alpha \mapsto C\alpha \cup \{c\ \tau\}])$ satisfies the requirement.

(c) $\Gamma = \Gamma_1 \cup \Gamma_2$: By a straightforward induction on $|\Gamma|$ and possibly $|dom(C)|$.

Induction step ($S\bar{\tau}_3 = \kappa\ \tau'$): Consider $mgn'\ (\kappa\ \tau')\ \Gamma\ (S, C)$.

(a) $\Gamma = \{\}$: Obvious.

(b) $\Gamma = \{(c\,\tau)\}$: Consider $mgn'\ (\kappa\ \tau')\ (c\ \tau)\ (S, C)$.
Note that if $\bar{S} = \bar{R} \circ S$ then $\bar{R}\tau = \bar{R}(S'\tilde{\tau})$ since $\bar{C} \Vdash \bar{R}((\kappa\ \tau') :: (c\ \tau))$, $\ulcorner$ **inst** $C' \Rightarrow \kappa\,\tilde{\tau}' :: c\,\tilde{\tau}\ \urcorner \in Ds$, and $S' = match\ \tilde{\tau}'\ \tau'$. Thus, by a similar inductive argument for $mgu\ \tau\ (S'\tilde{\tau})\ (S, C)$ using 1, we get $(\bar{S}, \bar{C}) \preceq (S'', C'') \preceq (S, C)$ and $S''\tau = S''(S'\tilde{\tau})$. Then since for all i, $|\tau_i'| < |\kappa\ \tau'|$ and $|dom(C'')| \leq |dom(C)|$, we can get the result by a straightforward induction.

(c) $\Gamma = \Gamma_1 \cup \Gamma_2$: By a straightforward induction.

■

Based on these lemmas, we have the following theorem for our unification algorithm.

Theorem 5.6 Given a constrained substitution (S_0, C_0) and types τ_1, τ_2, if there is a (S_0, C_0)-preserving unifier of τ_1 and τ_2 then $mgu\ \tau_1\ \tau_2\ (S_0, C_0)$ returns a most general such unifier. If there is no such unifier then $mgu\ \tau_1\ \tau_2\ (S_0, C_0)$ fails in a finite number of steps.

begin by showing that *tp* is indeed a constrained substitution transformer.

Lemma 6.1 Given a constrained substitution (S, C), an expression e, and type assumption set A such that SA is covered by C, if $tp(e, A, S, C) = (\tau, S', C')$, then (S', C') is a constrained substitution and $S' = R \circ S$ for some substitution R.

Proof: By a straightforward induction, and using Lemma 5.1 and Lemma 5.2. ■

Hence we shall often omit the requirement of constrained substitution from now on when writing $tp(e, A, S, C)$.

One may notice that we have used a new generalization function in dealing with **let**-expression. The reason for doing so can be stated as follows. Recall the typing rule (let') presented in Section 4. There are two contexts used in the antecedent part of that rule : one for deriving the type of the let-definition and one for the type of the let-body. But only the second one appears in the conclusion part and it is those instance assumptions contained in the first one that are generalized by the *gen* function. While in *tp*, we maintain a single context and pass it through the whole algorithm. If we were to use the *gen* function in the **let**-expression in *tp* we would overgeneralize those instance assumptions generated in the previous stages and passed to *tp* as part of the initial context.

To avoid such overgeneralization, we need to confine the domain of generalization to only those instance assumptions generated while reconstructing the type of the let-definition. The new generalization function, *tpgen*, compared to *gen*, takes an extra context parameter, C', whose instance assumptions will be excluded from generalization. Then, in the algorithm, when doing generalization, we pass the initial context to *tpgen* as the second context argument to restrict the domain of generalization. Hence only those newly generated instance assumptions will be generalized.

The following lemma shows that the context is preserved during the type reconstruction process and thus justifies our scheme.

Lemma 6.2 Given a constrained substitution (S, C), an expression e, and type assumption set A such that SA is covered by C, if $tp(e, A, S, C) = (\tau, S', C')$ then $(S', C') \preceq (S, C)$.

Proof: By induction on the structure of e, but we need some stronger induction hypothesis. Let $S' = R \circ S$. The induction hypothesis is:

1. $dom(C) \subseteq dom(S') \uplus dom(C')$.
2. $fv(\tau) \subseteq dom(C')$ and hence $(C')^*(fv\ \tau) \subseteq dom(C')$.
3. $fv(S'A) \subseteq dom(C')$ and hence $(C')^*(fv\ S'A) \subseteq dom(C')$.
4. RC is covered by C'.
5. If $\rho \in dom(C) \backslash C^*(fv\ SA)$ then $\rho \in dom(C') \backslash (C')^*(fv\ S'A)$, but $\rho \notin dom(R)$, $\rho \notin (C')^*(fv\ \tau)$ and $\rho \notin (C')^*(reg\ R)$.
6. $(S', C') \preceq (S, C)$.

Case $e = x$: Suppose that $x : \sigma \in A$ and S is *safe* for σ. Let $\sigma = \forall \alpha_i :: \Gamma_i\ \tau'$ and β_i be new type variables. We have

$$\begin{aligned} tp(x, A, S, C) &= inst(S(Ax), S, C) \\ &= inst(\forall \alpha_i :: S\Gamma_i . S\tau', S, C) \\ &= (J(S\tau'), S, C \uplus C_0) \end{aligned}$$

where $J = [\alpha_i \mapsto \beta_i]$ and $C_0 = \{\beta_i :: J(S\Gamma_i)\}$

So, $\tau = J(S\tau')$, $S' = S$, $C' = C \uplus C_0$, and $R = id$. Therefore

1. Obvious.
2. $$\begin{aligned} fv(\tau) &\subseteq fv(S\sigma) \cup \{\beta_i\} \\ &\subseteq fv(SA) \cup \{\beta_i\} \\ &\subseteq dom(C'). \end{aligned}$$
3. Obvious.
4. Obvious.
5. Obvious.
6. Obvious.

Case $e = \lambda x.e'$: By a straightforward induction.

Case $e = e_1 e_2$: We have

$$\begin{aligned} tp\ (e_1\ e_2, A, S, C) \ = \ \textbf{let}\ (\tau_1, S_1, C_1) &= tp\ (e_1, A, S, C) \\ (\tau_2, S_2, C_2) &= tp\ (e_2, A, S_1, C_1) \\ &\alpha \text{ a fresh type variable} \\ (S_3, C_3) &= mgu\ \tau_1\ (\tau_2 \to \alpha)\ (S_2, C_2.\alpha :: \{\}) \\ \textbf{in}\ (S_3\alpha, S_3, C_3) & \end{aligned}$$

Now in order to use the lemmas for the unification algorithm, we need to show that $fv(S_2\tau_1) \cup fv(S_2\tau_2) \subseteq dom(C_2)$. From the induction hypothesis of (2), we have $fv(\tau_1) \subseteq dom(C_1)$ and $fv(\tau_2) \subseteq dom(C_2)$. But, since both (S_1, C_1) and (S_2, C_2) are constrained substitution, we know that $S_1\tau_1 = \tau_1$ and $S_2\tau_2 = \tau_2$, and hence it suffices to look into $fv(R_2\tau_1)$ only where $S_2 = R_2 \circ S_1$.

Consider type variable β, $\beta \in fv(\tau_1)$. By the induction hypothesis of (6), we know that $C_2 \Vdash R_2 C_1$. Thus, if $\beta \in (C_1)^*(fv\ S_1A)$ then $fv(R_2\beta) \subseteq (C_2)^*(fv\ S_2A)$ according to Lemma 4.3, and hence $fv(R_2\beta) \subseteq dom(C_2)$ by the induction hypothesis of (3). Otherwise, we have $\beta \notin dom(R_2)$ according to the induction hypothesis of (5) and hence $\beta \in dom(C_2)$ by the induction hypothesis of 1. Therefore $fv(S_2\tau_1) \cup fv(S_2\tau_2) \subseteq dom(C_2)$ and hence $(C_2)^*(fv(S_2\tau_1) \cup fv(S_2\tau_2)) \subseteq dom(C_2)$ since C_2 is closed.

Let $S_1 = R_1 \circ S$, $S_2 = R_2 \circ S_1$, and $S_3 = R_3 \circ S_2$. We continue our proof:

1. By induction,

$$dom(C) \subseteq dom\ S_1 \uplus dom\ C_1 \subseteq dom\ S_2 \uplus dom\ C_2$$

From Lemma 5.2, $dom(C_2.\alpha :: \{\}) \setminus dom(C_3) = dom(R_3)$. Thus, $dom(S_2) \uplus dom(C_2) \subseteq dom(S_3) \uplus dom(C_3)$, and hence $dom(C) \subseteq dom(S_3) \uplus dom(C_3)$.

2. We know that $\tau = S_3\alpha$. If $\alpha \in dom(R_3)$ then, from Lemma 5.1, we know that $reg(R_3) \subseteq dom(C_2)$, so $fv(S_3\alpha) \subseteq dom(C_2)$. Moreover, from Lemma 5.2, $dom(C_2.\alpha :: \{\}) \setminus dom(C_3) = dom(R_3)$. But, since $dom(R_3) \cap reg(R_3) = \emptyset$, so we have $fv(S_3\alpha) \subseteq dom(C_3)$. Otherwise, $\alpha \notin dom(R_3)$ and $fv(S_3\alpha) = \alpha$. Hence $\alpha \in dom(C_3)$ as $dom(C_2.\alpha :: \{\}) \setminus dom(C_3) = dom(R_3)$.

3. By induction, $(C_2^*)(fv\ S_2A) \subseteq dom(C_2)$. From Lemma 5.1, $reg(R_3) \subseteq dom(C_2.\alpha :: \{\})$, and hence $fv(S_3A) \subseteq dom(C_2.\alpha :: \{\})$. Moreover, from Lemma 5.2, we know that $dom(C_2\ \alpha :: \{\}) \setminus dom(R_3) = dom(C_3)$. But, since none of the variable in $dom(R_3)$ occurs in $fv(S_3A)$, we have $fv(S_3A) \subseteq dom(C_3)$ and hence $(C_3)^*(fv\ S_3A) \subseteq dom(C_3)$ as C_3 is closed.

4. By induction,

$$fv(R_1C) \subseteq dom(C_1) \quad \text{and} \quad fv(R_2C_1) \subseteq dom(C_2).$$

So we have $fv(R_2R_1C) \subseteq dom(C_2)$. Moreover, from Lemma 5.1 and Lemma 5.2, we know that

$$dom(R_3),\ reg(R_3) \subseteq dom(C_2.\alpha :: \{\}) \quad \text{and} \quad dom(C_2.\alpha :: \{\}) \setminus dom(C_3) = dom(R_3).$$

But since $dom(R_3) \cap reg(R_3) = \emptyset$, it then follows that $fv(R_3R_2R_1C) \subseteq dom(C_3)$.

5. For any ρ, $\rho \in dom(C) \setminus C^*(fv\ SA)$, we have, by induction, $\rho \in dom(C_2) \setminus (C_2)^*(fv\ S_2A)$, but $\rho \notin dom(R_1) \cup (C_1)^*(fv\ \tau_1) \cup (C_1)^*(reg\ R_1)$ and $\rho \notin dom(R_2) \cup (C_2)^*(fv\ \tau_2) \cup (C_2)^*(reg\ R_2)$. Next, from Lemma 5.1, $dom(R_3) \cup reg(R_3) \subseteq (C_2)^*(fv\ R_2\tau_1) \cup (C_2^*)(fv\ \tau_2) \cup \{\alpha\}$. So, we have $\rho \notin dom(R_3) \cup fv(\tau) \cup reg(R_3)$ where $\tau = S_3\alpha$. Moreover, from Lemma 5.2, $dom(C_2.\alpha :: \{\}) \setminus dom(C_3) = dom(R_3)$. In other words, $\rho \in dom(C_3)$, but $\rho \notin fv(S_3A)$. Now, since $dom(C_3) \subseteq dom(C_2)$, clearly we have $\rho \in dom(C_3) \setminus (C_3)^*(fv\ S_3A)$, but $\rho \notin dom(R_3) \cup (C_3)^*(fv\ \tau) \cup (C_3)^*(reg\ R_3)$.

   Finally, $R = R_3 \circ R_2 \circ R_1$. It follows from the argument above that $\rho \notin dom(R) \cup (C_3)^*(fv\ \tau) \cup (C_3)^*(reg\ R)$.

6. By a straightforward induction, and using Lemma 5.4.

Case $e = (\textbf{let}\ x = e_1\ \textbf{in}\ e_2)$: we have

$$\begin{array}{rcl} tp\ (\textbf{let}\ x = e_1\ \textbf{in}\ e_2, A, S, C) & = & \textbf{let}\ (\tau_1, S_1, C_1) = tp\ (e_1, A, S, C) \\ & & \quad\ (\sigma, C_2) = tpgen\ (\tau_1, S_1A, C_1, C) \\ & & \quad\ (\tau_2, S_2, C_3) = tp\ (e_2, A.x{:}\sigma, S_1, C_2) \\ & & \textbf{in}\ \ (\tau_2, S_2, C_3) \end{array}$$

By the induction hypothesis of (3), $(C_1)^*(fv\ S_1A) \subseteq dom(C_1)$. To do induction on the second recursive call, we need to show that $(C_2)^*(fv\ S_1(A.x : \sigma)) \subseteq dom(C_2)$. Suppose that $\sigma = \forall\alpha_i :: \Gamma_i.\tau_1$. Then we have $dom(C_1) \setminus dom(C_2) = \{\alpha_i\}$, $fv(\sigma) \subseteq (C_1^*)(fv\ S_1A)$, and none of the α_i can occur in $(C_1)^*(\mathit{fv}\ S_1A)$ according to *tpgen*'s definition. So, $\mathit{fv}(S_1(A.x:\sigma)) \subseteq \mathit{dom}(C_2)$ and hence $(C_2^*)(\mathit{fv}\ S_1(A.x:\sigma)) \subseteq \mathit{dom}(C_2)$.

Let $S_1 = R_1 \circ S$ and $S_2 = R_2 \circ S_1$. We continue the proof:

1. By induction, $\mathit{dom}(C) \subseteq \mathit{dom}(S_1) \uplus \mathit{dom}(C_1)$ and $\mathit{dom}(C_2) \subseteq \mathit{dom}(S_2) \uplus \mathit{dom}(C_3)$. To complete the proof, it suffices to show that $\mathit{dom}(C) \subseteq \mathit{dom}(S_1) \uplus \mathit{dom}(C_2)$. But, since none of the α_i occurs in $\mathit{dom}(C)$, this is obviously true. Hence $\mathit{dom}(C) \subseteq \mathit{dom}(S_2) \uplus \mathit{dom}(C_3)$ follows directly by induction on the second recursive call.

2. By a straightforward induction.

3. By a straightforward induction.

4. By induction, $\mathit{fv}(R_1C) \subseteq \mathit{dom}(C_1)$ and $\mathit{fv}(R_2C_2) \subseteq \mathit{dom}(C_3)$. To completethe proof, it suffices to show that none of the α_i occurs in $\mathit{fv}(R_1C)$; this in turn can be shown by considering $\mathit{fv}(R_1\alpha)$ for $\alpha \in \mathit{dom}(C)$: If $\alpha \in C^*(\mathit{fv}\ SA)$, then, from Lemma 4.3, $\mathit{fv}(R_1\alpha) \subseteq (C_1)^*(\mathit{fv}\ S_1A)$ and hence none of the α_i occurs in $\mathit{fv}(R_1\alpha)$ in this case. Otherwise, $\alpha \notin \mathit{dom}(R_1)$ according to the induction hypothesis of (5) and hence $R_1\alpha = \alpha$. But, since none of the α_i occurs in $\mathit{dom}(C)$, we know that $\alpha \neq \alpha_i$, for all i. Therefore, $\mathit{fv}(R_1C) \subseteq \mathit{dom}(C_2)$, and hence $\mathit{fv}(R_2R_1C) \subseteq \mathit{dom}(C_3)$.

5. For any ρ, $\rho \in \mathit{dom}(C) \setminus C^*(\mathit{fv}\ SA)$, we have, by induction, $\rho \in \mathit{dom}(C_1) \setminus (C_1)^*(\mathit{fv}\ S_1A)$, but $\rho \notin \mathit{dom}(R_1) \cup (C_1)^*(\mathit{fv}\ \tau_1) \cup (C_1)^*(\mathit{reg}\ R_1)$. By the definition of *tpgen*, we have $\rho \in \mathit{dom}(C_2)$ and $\mathit{dom}(C_2) \subseteq \mathit{dom}(C_1)$. Thus, $\rho \in \mathit{dom}(C_2) \setminus (C_2)^*(\mathit{fv}\ S_1A)$ and the result follows directly by induction on the second recursive call.

6. By induction, $(S_1, C_1) \preceq (S, C)$ and $(S_2, C_3) \preceq S_1, C_2)$. To complete the proof we need to show $(S_1, C_2) \preceq (S, C)$. By an argument similar to the one used in (4), we know that none of the α_i occurs in R_1C. So, $(S_1, C_2) \preceq (S, C)$ and C_3 covers R_2R_1C. It then follows immediately that $(S_2, C_3) \preceq (S, C)$.

■

Now we can proceed to prove the soundness of our algorithm.

Theorem 6.3 If $\mathit{tp}(e, A, S, C) = (\tau, S', C')$ then $S'A, C' \vdash' e : \tau$.

Proof: By induction on the structure of e.

Case $e = x$: Suppose that $x:\sigma \in A$ and S is *safe* for σ. Let $\sigma = \forall\alpha_i :: \Gamma_i\ \tau'$ and β_i be new type variables. We have

$$\begin{aligned} \mathit{tp}(x, A, S, C) &= \mathit{inst}(S(Ax), S, C) \\ &= \mathit{inst}(\forall\alpha_i :: S\Gamma_i.S\tau', S, C) \\ &= (J(S\tau'), S, C \uplus C_0) \end{aligned}$$

where $J = [\alpha_i \mapsto \beta_i]$ and $C_0 = \{\beta_i :: J(S\Gamma_i)\}$

Finally, it is clear that $\sigma \preceq_{C_2} \bar{\sigma}$ by Lemma 3.3 and Lemma 3.4. Then, from $C_3 \Vdash R_2 C_2$, we have, by Lemma 3.5, $R_2\sigma \preceq_{C_3} R_2\bar{\sigma}$. But $S_2\sigma = R_2\sigma$, hence $S_2\sigma \preceq_{C_3} \sigma'$. Therefore, we can construct the following derivation

$$\dfrac{\dfrac{\dfrac{S_1A, C_1 \vdash' e_1 : \tau_1}{R(S_1A), C' \vdash' e_1 : R\tau_1}\ (\mathit{Lemma}\ 4.6)}{S_2A, C' \vdash' e_1 : R\tau_1}\ (\mathit{Lemma}\ 4.5) \qquad \dfrac{S_2A.x\!:\!S_2\sigma, C_3 \vdash' e_2 : \tau_2}{S_2A.x\!:\!\sigma', C_3 \vdash' e_2 : \tau_2}\ (\mathit{Lemma}\ 4.8)}{S_2A, C_3 \vdash' \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau_2}\ (\mathit{let}')$$

This completes the proof. ■

Together with Theorem 4.9, we have the following soundness result.

Corollary 6.4 (Soundness of tp) If $tp(e, A, S, C) = (\tau, S', C')$ then $S'A, C' \vdash e : \tau$.

Ultimately, we will state and prove that tp is able to compute the most general derivable typing for a given term.

Theorem 6.5 Suppose that $S'A, C' \vdash' e : \tau'$ and $(S', C') \preceq (S_0, C_0)$. Then $tp(e, A, S_0, C_0)$ succeeds with (τ, S, C), and there is a substitution R such that

1. $S' = RS$, except on new type variables of $tp(e, A, S_0, C_0)$,
2. $C' \Vdash RC$, and
3. $\tau' = R\tau$.

Proof: Let R_0 be the substitution such that $S' = R_0 \circ S_0$ and $C' \Vdash R_0 C_0$. We prove this theorem, again, by induction on the structure of $S'A, C' \vdash' e : \tau'$.

Case(var') : We have a derivation of the form :

$$S'A, C' \vdash' x : \tau' \quad (x : \sigma \in A,\ \ \tau' \preceq_{C'} S'\sigma).$$

Without loss of generality, we can assume that variables in σ have been suitably renamed so that no name clash will occur in the following proof. Let $\sigma = \forall\alpha_i\!::\!\Gamma_i.\nu$. Then $S'\sigma = \forall\alpha_i\!::\!S'\Gamma_i.S'\nu$.

As $\tau' \preceq_{C'} S'\sigma$, there are types τ_i such that

$$[\alpha_i \mapsto \tau_i](S'\nu) = \tau' \quad \text{and} \quad C' \Vdash \{\tau_i\!::\![\alpha_i \mapsto \tau_i](S'\Gamma_i)\}.$$

On the other hand, let β_i be new type variables, we have

$$tp(x, A, S_0, C_0) = ([\alpha_i \mapsto \beta_i](S_0\nu),\ S_0,\ C_0 \uplus C_1)$$

where $C_1 = \{\beta_i\!::\![\alpha_i \mapsto \beta_i](S_0\Gamma_i)\}$. In other words, $\tau = [\alpha_i \mapsto \beta_i](S_0\nu)$, $S = S_0$, and $C = C_0 \uplus C_1$.

Now let $R = [\beta_i \mapsto \tau_i]R_0$. Then, clearly $S' = RS$ except on the new type variables β_i. Moreover, we have

$$\begin{array}{rcll} R\tau & = & ([\beta_i \mapsto \tau_i]R_0)[\alpha_i \mapsto \beta_i]S_0\nu & \\ & = & ([\beta_i \mapsto \tau_i][\alpha_i \mapsto \beta_i])(R_0S_0)\nu & (R_0 \text{ safe for } \alpha_i) \\ & = & [\alpha_i \mapsto \tau_i]S'\nu & \\ & = & \tau' & \end{array}$$

So, $\tau = J(S\tau')$, $S' = S$, and $C' = C \uplus C_0$. Clearly, $\tau \preceq_{C'} S'\sigma$ and hence

$$S'A, C' \vdash' x : \tau$$

using (var') rule.

Case $e = e_1 e_2$: We have

$$\begin{array}{rcl} tp\,(e_1\ e_2, A, S, C) & = & \textbf{let}\ (\tau_1, S_1, C_1) \ = \ tp\,(e_1, A, S, C) \\ & & \quad\ (\tau_2, S_2, C_2) \ = \ tp\,(e_2, A, S_1, C_1) \\ & & \quad\ \alpha \text{ a fresh type variable} \\ & & \quad\ (S_3, C_3) \ = \ mgu\ \tau_1\ (\tau_2 \rightarrow \alpha)\ (S_2, C_2.\alpha :: \{\}) \\ & & \textbf{in}\ (S_3\alpha, S_3, C_3) \end{array}$$

By induction,

$$S_1A, C_1 \vdash' e_1 : \tau_1 \quad \text{and} \quad S_2A, C_2 \vdash' e_2 : \tau_2.$$

Let $S_1 = R_1 \circ S$, $S_2 = R_2 \circ S_1$, $S_3 = R_2 \circ S_2$. From Lemma 6.2, we have $C_1 \Vdash R_1 C$, $C_2 \Vdash R_2 C_1$. Furthermore, we know that $S_3\tau_1 = S_3(\tau_2 \rightarrow \alpha)$ and $C_3 \Vdash R_3(C_2.\alpha :: \{\})$ from Lemma 5.4. Hence $C_3 \Vdash R_3(R_2 C_1)$ and we can construct the following derivation

$$\dfrac{\dfrac{\dfrac{S_1A, C_1 \vdash' e_1 : \tau_1}{S_3A, C_3 \vdash' e_1 : S_3\tau_1}\ (Lemma\ 4.6)}{S_3A, C_3 \vdash' e_1 : S_3\tau_2 \rightarrow S_3\alpha}\ (Lemma\ 5.4) \qquad \dfrac{S_2A, C_2 \vdash' e_2 : \tau_2}{S_3A, C_3 \vdash' e_2 : S_3\tau_2}\ (Lemma\ 4.6)}{S_3A, C_3 \vdash' e_1 e_2 : S_3\alpha}\ (\lambda - elim')$$

Case $e = \lambda x.e'$: By a straightforward induction.

Case $e = (\textbf{let}\ x = e_1\ \textbf{in}\ e_2)$: We have

$$\begin{array}{rcl} tp\,(\textbf{let}\ x = e_1\ \textbf{in}\ e_2, A, S, C) & = & \textbf{let}\ (\tau_1, S_1, C_1) \ = \ tp\,(e_1, A, S, C) \\ & & \quad\ (\sigma, C_2) \ = \ tpgen\,(\tau_1, S_1A, C_1, C) \\ & & \quad\ (\tau_2, S_2, C_3) \ = \ tp\,(e_2, A.x\!:\!\sigma, S_1, C_2) \\ & & \textbf{in}\ \ (\tau_2, S_2, C_3) \end{array}$$

By induction

$$S_1A, C_1 \vdash' e_1 : \tau_1 \quad \text{and} \quad S_2A.x\!:\!S_2\sigma, C_3 \vdash' e_2 : \tau_2$$

We shall combine these two derivations using Lemma 4.5 to complete the proof.

First, let $(\bar\sigma, \bar C) = gen(\tau_1, S_1A, C_1)$. Then, $\bar\sigma = \forall\alpha_i :: \Gamma_i.\sigma$ for some i, $i \geq 0$, and $\{\alpha_i :: \Gamma_i\} \subseteq C_2$ according to the definitions of gen and $tpgen$. Next, let $S_2 = R_2 \circ S_1$. From Lemma 6.2, $C_3 \Vdash R_2 C_2$. Hence $C_3 \Vdash R_2\bar C$, since $\bar C \preceq C_2$. Now, applying Lemma 4.5 to $\bar\sigma$, R_2, and C_3, we get a substitution R and a context C' such that

$$R(S_1A) = R_2(S_1A), \quad C' \Vdash RC_1, \quad R_2\bar\sigma = \sigma' \ \text{and}\ \ C'' \preceq C_3$$

where $(\sigma', C'') = gen(R\tau_1, R_2(S_1A), C')$.

and $RC = R_0C_0 \uplus RC_1$. But, since

$$\begin{array}{rcl} RC_1 & = & [\beta_i \mapsto \tau_i]R_0(\{\beta_i :: [\alpha_i \mapsto \beta_i]S_0\Gamma_i\}) \\ & = & \{\tau_i :: [\alpha_i \mapsto \tau_i]R_0S_0\Gamma_i\} \\ & = & \{\tau_i :: [\alpha_i \mapsto \tau_i](S'\Gamma_i)\}, \end{array}$$

so we have $C' \Vdash RC$ as well.

Case($\lambda-elim'$) : We have a derivation of the form :

$$\frac{S'A, C' \vdash' e_1 : \tau_1 \rightarrow \tau_2 \qquad S'A, C' \vdash' e_2 : \tau_1}{S'A, C' \vdash' e_1e_2 : \tau_2}$$

By induction, $tp(e_1, A, S_0, C_0) = (\nu_1, S_1, C_1)$ succeeds and there is a substitution R_1 such that

1. $S' = R_1S_1$, except on Δ_1,
2. $C' \Vdash R_1C_1$, and
3. $R_1\nu_1 = \tau_1 \rightarrow \tau_2$.

where Δ_1 is the set of new type variables of $tp(e_1, A, S_0, C_0)$.

Clearly, $S'A = (R_1S_1)A$ and hence $(R_1S_1)A, C' \vdash' e_2 : \tau_1$. So, by induction , $tp(e_2, A, S_1, C_1) = (\nu_2, S_2, C_2)$ succeeds and there is a substitution R_2 such that

1. $R_1S_1 = R_2S_2$, except on Δ_2,
2. $C' \Vdash R_2C_2$, and
3. $R_2\nu_2 = \tau_1$.

where Δ_2 is the set new type variables of $tp(e_2, A, S_1, C_1)$.

Now let α be a new variable and $R' = [\alpha \mapsto \tau_2]R_2S_2$. Then, clearly

$$S' = R' \text{ except on } \Delta_1 \cup \Delta_2 \cup \{\alpha\}$$

and

$$C' \Vdash [\alpha \mapsto \tau_2]R_2(C_2.\alpha :: \{\}).$$

In addition, we show that (R', C') is indeed a $(S_2, C_2.\alpha :: \{\})$-preserving unifier of ν_1 and $\nu_2 \rightarrow \alpha$:

$$\begin{array}{rcll} R'\nu_1 & = & (R_2S_2)\nu_1 & (\alpha \ new) \\ & = & (R_1S_1)\nu_1 & (R_1S_1 = R_2S_2 \ except \ on \ \Delta_2) \\ & = & R_1\nu_1 & (S_1C_1 = C_1) \\ & = & \tau_1 \rightarrow \tau_2 & \\ & = & R_2\nu_2 \rightarrow R'\alpha & \\ & = & R_2(S_2\nu_2) \rightarrow R'\alpha & (S_2C_2 = C_2) \\ & = & R'\nu_2 \rightarrow R'\alpha & (\alpha \ new) \\ & = & R'(\nu_2 \rightarrow \alpha) & \end{array}$$

Thus, from Theorem 5.6, *mgu* ν_1 $(\nu_2 \to \alpha)$ $(S_2, C_2.\alpha :: \{\})$ succeeds with (S_3, C_3) such that $S_3\nu_1 = S_3(\nu_2 \to \alpha)$, $R' = R \circ S_3$ and $C' \Vdash RC_3$ for some substitution R.

Since *mgu* does not introduce any new type variables, we know that $\Delta_1 \cup \Delta_2 \cup \{\alpha\}$ is the set of new type variables of $tp(e_1 e_2, A, S_0, C_0)$. Therefore, $tp(e_1 e_2, A, S_0, C_0)$ succeeds with $(S_3\alpha, S_3, C_3)$ and R is the required substitution:

1. $S' = RS_3$, except on new type variables of $tp(e_1 e_2, A, S_0, C_0)$,
2. $C' \Vdash RC_3$, and
3. $\tau_2 = R'\alpha = R(S_3\alpha)$.

Case($\lambda - intro'$) : We have a derivation of the form :

$$\frac{S'A.x{:}\tau', C' \ \vdash' \ e : \tau}{S'A, C' \ \vdash' \ \lambda x.e : \tau' \to \tau}$$

Let α be a new type variable and $S'' = [\alpha \mapsto \tau'] \circ S'$. Then, $S''(A.x : \alpha), C' \ \vdash' \ e : \tau$ and $(S'', C') \preceq (S_0, C_0.\alpha :: \{\})$. So, by induction, $tp(e, A.x : \alpha, S_0, C_0.\alpha :: \{\}) = (\tau_1, S, C)$ succeeds and there is a substitution R such that

1. $S'' = RS$ except on the new type variables of $tp(e, A.x{:}\alpha, S_0, C_0.\alpha :: \{\})$,
2. $C' \Vdash RC$, and
3. $R\tau_1 = \tau$.

Therefore, $tp(\lambda x.e, A, S_0, C_0)$ succeeds with $(S\alpha \to \tau_1, S, C)$ and R is the required substitution :

1. $S' = RS$ except on new type variables of $tp(\lambda x.e, A, S_0, C_0)$
2. $C' \Vdash RC$, and
3. $R(S\alpha \to \tau_1) = S''\alpha \to \tau = \tau' \to \tau$.

Case(let') : We have a derivation of the form:

$$\frac{S'A, D \ \vdash' \ e_1 : \tau_1 \qquad S'A.x{:}\sigma', C' \ \vdash' \ e_2 : \tau_2}{S'A, C' \ \vdash' \ \mathbf{let}\ x = e_1 \ \mathbf{in}\ e_2 : \tau_2}$$

where $(\sigma', C'') = gen(\tau_1, S'A, D)$ and $C'' \preceq C'$.

We cannot do a direct induction on $S'A, D \ \vdash' \ e_1 : \tau_1$, since this would require $(S', D) \preceq (S_0, C_0)$, which is not true in general. But, we note that $(S, D \cup C')$ can suit our purpose here. Without loss of generality, we can assume that $dom(D) \cap dom(C') = dom(C'')$. So, $D \bowtie C'$, and if $\sigma' = \forall\alpha_i :: \Gamma_i.\tau_1$ then $D \cup C' = \{\alpha_i :: \Gamma_i\} \uplus C'$ and hence $(S', D \cup C') \preceq (S_0, C_0)$.

Moreover, $S'A, D \cup C' \ \vdash' \ e_1 : \tau_1$ by Lemma 4.7. So, by indution, $tp(e_1, A, S_0, C_0) = (\nu_1, S_1, C_1)$ succeeds and there is a substitution R_1 such that

1. $S' = R_1 S_1$ except on new type variables of $tp(e_1, A, S_0, C_0)$,

2. $D \cup C' \Vdash R_1 C_1$, and

3. $R_1 \nu_1 = \tau_1$.

Hence $S'A = (R_1 S_1)A$ and $(R_1 S_1)A.x:\sigma', C' \vdash' e_2 : \tau_2$. Now, in order to apply the induction hypothesis to the second recursive call, we need to show that

$$C' \Vdash R_1 C_2 \quad \text{and} \quad \sigma' \preceq_{C'} R_1 \sigma$$

where $(\sigma, C_2) = tpgen(\nu_1, S_1 A, C_1, C_0)$.

Since $(D \cup C') \setminus C' = \{\alpha_i :: \Gamma_i\}$, we can prove $C' \Vdash R_1 C_2$ by showing that none of the α_i occurs in $R_1 C_2$; this in turn can be proved by analyzing $fv(R_1\alpha)$ for $\alpha \in dom(C_2)$. There are two possibie cases according to *tpgen*'s definition:

- $\alpha \in (C_1)^*(fv\ S_1 A)$: As $D \cup C' \Vdash R_1 C_1$, we have $fv(R_1\alpha) \subseteq (D \cup C')^*(fv\ R_1 S_1 A)$ by Lemma 4.3. But, since $R_1 S_1 A = S'A$ and $(D \cup C')^*(fv\ S'A) = D^*(fv\ S'A)$, so we have $fv(R_1\alpha) \subseteq D^*(fv\ S'A)$ and hence none of the α_i occurs in $fv(R_1\alpha)$ in this case.

- $\alpha \in (dom(C_1) \cap dom(C_0)) \setminus (C_1)^*(fv\ S_1 A)$: Since α is not a new type variable of $tp(e_1, A, S_0, C_0)$, $R_1 S_1 \alpha = S'\alpha$. Moreover, as $\alpha \in dom(C_1)$, from (1) of Lemma 6.2, we have $\alpha \notin dom(S_1)$. But, $S' = R_0 S_0$, so $fv(R_1\alpha) = fv(R_1 S_1 \alpha) = fv(R_0\alpha)$. Finally, from C' covers $R_0 C_0$ and $C' \Vdash R_0 C_0$, it follows that $fv(R_1\alpha) \subseteq dom(C')$ and hence none of the α_i occurs in $fv(R_1\alpha)$ in this case, either.

To show that $\sigma' \preceq_{C'} R_1\sigma$, we first define $(\tilde{\sigma}, \tilde{C}) = gen(\tau_1, S'A, D \cup C')$. Then, clearly $\sigma' \preceq_{C'} \tilde{\sigma}$ and $\tilde{C} \preceq C'$. Next, let $(\bar{\sigma}, \bar{C}) = gen(\nu_1, S_1 A, C_1)$. We observe that R_1 is the substitution that relates these two generalization. So, from Lemma 4.4, we have $\tilde{\sigma} \preceq_{\tilde{C}} R_1\bar{\sigma}$ and hence $\tilde{\sigma} \preceq_{C'} R_1\bar{\sigma}$. Finally, note that $\bar{\sigma} \preceq_{C_2} \sigma$ according to their definitions. So, from $C' \Vdash R_1 C_2$, we have $R_1\bar{\sigma} \preceq_{C'} R_1\sigma$ by Lemma 3.5, and hence $\sigma' \preceq_{C'} R_1\sigma$ by the transitivity of $\preceq_{C'}$.

Now we can proceed to the second recursive call to *tp*. By hypothesis, $S'A.x:\sigma', C' \vdash' e_2 : \tau_2$, but

$$S'A.x:\sigma' \;=\; R_1 S_1 A.x:\sigma' \;\preceq_{C'}\; R_1 S_1 A.x:R_1\sigma \;=\; R_1 S_1(A.x:\sigma),$$

so $R_1 S_1(A.x:\sigma), C' \vdash' e_2 : \tau_2$ by Lemma 4.8. Hence by induction, $tp(e_2, A.x:\sigma, S_1, C_2) = (\nu_2, S_2, C_3)$ succeeds and there is a substitution R_2 such that

1. $R_1 S_1 = R_2 S_2$ except on new type variables of $tp(e_2, A, S_1, C_2)$,

2. $C' \Vdash R_2 C_3$, and

3. $\tau_2 = R_2 \nu_2$.

Therefore, $tp(\textbf{let } x = e_1 \textbf{in } e_2, A, S_0, C_0)$ succeeds with (ν_2, S_2, C_3) and R_2 is the required substitution since $S' = R_2 S_2$ except on new type variables of $tp(\textbf{let } x = e_1 \textbf{in } e_2, A, S_0, C_0)$.

This completes the proof. ■

Together with Theorem 4.10, we have the completeness result.

Corollary 6.6 (Completeness of *tp*) Suppose that $S'A, C' \vdash e : \sigma'$ and $(S', C') \preceq (S_0, C_0)$. Then $tp(e, A, S_0, C_0)$ succeeds with (τ, S, C), and there is a substitution R such that

1. $S' = RS$ except on the new type variables of $tp(e, A, S_0, C_0)$, and
2. $\sigma' \preceq_{C'} R\sigma$

where $(\sigma, \tilde{C}) = gen(\tau, SA, C)$.

Proof: By Theorem 4.10, $S'A, C_1 \vdash' e : \nu$ for some context C_1 that $C' \preceq C_1$ and some type ν such that $\sigma' \preceq_{C'} \sigma_1$ where $(\sigma_1, C_1') = gen(\nu, S'A, C_1)$. Now, since $(S', C') \preceq (S_0, C_0)$, we have $(S', C_1) \preceq (S_0, C_0)$. Therefore, by Theorem 6.5, if $tp(e, A, S_0, C_0) = (\tau, S, C)$, then there is a substitution R such that

1. $S' = RS$ except on new type variables of $tp(e, A, S_0, C_0)$,
2. $C_1 \Vdash RC$, and
3. $\nu = R\tau$.

Clearly, $S'A = SA$. Now let $(\sigma, \tilde{C}) = gen(\tau, SA, C)$. Then, applying Lemma 4.4 on R, we get $\sigma_1 \preceq_{C_1'} R\sigma$. But $C_1' \preceq C'$, so $\sigma' \preceq_{C'} R\sigma$ by Lemma 3.4. ∎

As a corollary, we have the following result for principal type schemes.

Corollary 6.7 Suppose that $dom(C_0) = (C_0)^*(fv\ S_0A)$ and $tp(e, A, S_0, C_0) = (\tau, S, C)$. Then σ is a principal type scheme for e under SA and C' where $(\sigma, C') = gen(\tau, SA, C)$

Proof: If $tp(e, A, S_0, C_0) = (\tau, S, C)$ then $SA, C \vdash e : \tau$ by Theorem 6.3 and Theorem 4.9. Now, since $(\sigma, C') = gen(\tau, SA, C)$, we have $SA, C' \vdash e : \sigma$ by Lemma 4.2. Moreover, from Lemma 6.2, we have $(S, C) \preceq (S_0, C_0)$. In other words, $S = R_0S_0$ and $C \Vdash R_0C_0$ for some substitution R_0. So, in order to apply Corollary 6.6, we need to show that $(S, C') \preceq (S_0, C_0)$. From Lemma 4.1, we have $dom(C') = C^*(fv\ SA)$. So, by Lemma 4.3, we have $fv(R_0\alpha) \subseteq dom(C')$ for all $\alpha \in dom(C_0)$, and hence $C' \Vdash R_0C_0$. Therefore, $(S, C') \preceq (S_0, C_0)$.

Now, suppose that $SA, C' \vdash e : \sigma'$ for some type scheme σ'. By Corollary 6.6, there is a substitution R such that

1. $S = RS$ except on new type variables of $tp(e, A, S_0, C_0)$ and
2. $\sigma' \preceq_{C'} R\sigma$.

Furthermore, from Lemma 4.3, we know that $\bigcup\{fv(S\alpha) \mid \alpha \in (C_0)^*(fv\ S_0A)\} = C^*(fv\ SA)$. Hence type variables in $C^*(fv\ SA)$ are unchanged by R because of (1). Now, since $fv(\sigma) \subseteq C^*(fv\ SA)$, we have $R\sigma = \sigma$. It then follows that σ is a principal type scheme for e under SA and C'. ∎

7 Ambiguity Revisited

As we have seen in the introduction, parametric type classes share with standard type classes the problem that type schemes might be ambiguous.

Definition. Given a type scheme $\sigma = \forall\alpha_i :: \Gamma_i.\tau$, let $C_\sigma = \{\alpha_i :: \Gamma_i\}$ be the generic context of σ and $\tau_\sigma = \tau$ be the type proper of σ. Conversely, if $C = \{\alpha_i :: \Gamma_i\}$ and C covers τ then we shall write $< C, \tau >$ for the type scheme $\forall\alpha_i :: \Gamma_i.\tau$ formed from C and τ.

Definition. A generic type variable α in a type scheme $\sigma = \forall\alpha_i :: \Gamma_i.\tau$ is (weakly) *ambiguous* if (1) $C_\sigma\ \alpha \neq \emptyset$, and (2) $\alpha \notin C^*_\sigma(fv\ \tau)$.

Ambiguous type variables pose an implementation problem. The usual approach to implement overloading polymorphism is to pass extra *dictionary* arguments for every type class in the context of a function signature. Since the constraints on ambiguous variables are non-empty (1), dictionaries need to be passed. But since the ambiguous variable does not occur free in the type (2), it is never instantiated, hence we do not know which dictionaries to pass. Seen from another perspective, any dictionary of an appropriate instance type would do, but we have a problem of coherence: There are several implementations of an expression with possibly different semantics [Jon92a].

The problem is avoided by requiring that the programmer disambiguate expressions if needed, by using explicit type signatures. Conceptually, the ambiguity check takes place after type reconstruction; would it be part of type reconstruction then the principal type property would be lost. In a way, the ambiguity problem shows that sometimes reconstructed types are too general. Every ambiguous type has a substitution instance which is unambiguous (just instantiate ambiguous variables). The trouble is that there is not always a most general, unambiguous type.

Compared to multi-argument type classes, our type system often produces types with less ambiguity. Consider:

```
len :: (sa :: Sequence a) => sa -> Int
```

Seen as a multi-argument type class, a would be ambiguous, since it occurs in a predicate but not in the type itself. Seen as a parametric type class, however, a is not ambiguous: Although it does not occur in the type, it both unconstrained and dependent on sa through `(sa :: Sequence a)`. Hence both (1) and (2) fail.

Ambiguity problems can be further reduced by making use of the following observation: Because of restriction (b) in Section 3, the top-level type constructor of a type uniquely determines the dictionary that needs to be passed. Hence, if two types have the same top-level type constructor (but possibly different type arguments), their dictionaries share the same data constructor (but have possibly different parameters). We can recognize equality of top-level type constructors statically, using the following technique:

We introduce a special "root" class TC, with one type parameter but no operations. Every type is an instance of `TC` by virtue of the following instance declaration (which can be thought of being implicitly generated for every type $\kappa\ \tau$).

$$\textbf{inst}\ \kappa\ \tau :: TC\ (\kappa\ ())$$

Effectively, TC is used to "isolate" the top-level type constructor of a type. That is, if two types are related by a TC constraint, we know that they have the same top-level type constructor. The two types are then called *similar*:

Definition. Given a context C, let *similarity* in C, $(\sim_C)$, be the smallest transitive and symmetric relation such that $C \Vdash \tau_1 :: TC\ \tau_2$ implies $\tau_1 \sim_C \tau_2$.

TC is treated like every other type class during type reconstruction. It is treated specially in the ambiguity check, allowing us to strengthen the ambiguity criterion:

Definition. A generic type variable α in a type scheme σ is *strongly ambiguous* if α is weakly ambiguous in σ, and, for every type τ, $\alpha \sim_{C_\sigma} \tau$ implies that τ is a strongly ambiguous type variable in σ.

The TC technique enables us to type **map** precisely[3]

$$\textbf{map}\ : \forall a.\forall b.\forall t.\forall sa :: \{\mathit{Sequence}\ a,\ TC\ t\}.\forall sb :: \{\mathit{Sequence}\ b,\ TC\ t\}.(a \to b) \to sa \to sb$$

This states that sa and sb are instance types of *Sequence* with element types a and b, and that sa and sb share the same type constructor.

The knowledge that sa and sb have the same type constructor is initially on the meta-level, derived from the form of the compiler-generated instance declarations. We can formalize it in the type system as follows:

Definition. A type scheme $\sigma = \forall \alpha_i :: \Gamma_i.\tau'$ is in *reduced* form if none of the Γ_i contains a class $TC\ (\kappa\ \tau)$, for arbitrary constructor κ and type τ. We use σ_R for type schemes in reduced form.

Definition. Two type schemes σ_1, σ_2 are *equivalent* under a context C, $\sigma_1 \simeq_C \sigma_2$, iff for all reduced type schemes σ_R,

$$\sigma_R \preceq_C \sigma_1 \quad \Leftrightarrow \quad \sigma_R \preceq_C \sigma_2.$$

We extend the definition of generic instance to include equivalence: A type scheme σ_1 is a generic instance of a type scheme σ_2 under a context C if there is a type scheme σ' s.t. $\sigma_1 \simeq_C \sigma'$, and $\sigma' \preceq_C \sigma_2$ according to the definition of $\preceq_C$ in Section 3. This stronger notion of generic instance is important to check user-defined type signatures.

Example: After substituting *List a* for *sa*, the type signature of *map* would become:

$$\forall sb :: \{\mathit{Sequence}\ b,\ TC\ (\mathit{List}\ ())\}.(a \to b) \to \mathit{List}\ a \to sb$$

The usual definition of map for lists, on the other hand, would have type:

$$(a \to b) \to \mathit{List}\ a \to \mathit{List}\ b$$

Equivalence is necessesary to verify that the first type is an instance of the second.

Without loss of generality, we shall assume that all type schemes are closed and thus drop the subscript in $\simeq_C$ in the following discussion.

In order to handle the extended notion of generic instance, we introduce the function r to rewrite a type scheme into a more 'reduced', but equivalent one.

[3]Previously, it has been conjectured that this required second-order unification.

```
$r(\sigma)$ = let $(S, C) = reduce(C_\sigma)$
        in $< C, S(\tau_\sigma) >$
        where $reduce\,(C) =$
                if $\exists(\alpha :: \{TC\ (\kappa\,())\} \cup \Gamma) \in C$ then
                    let $\beta$ a fresh type variable
                    in $mgu\ \alpha\ (\kappa\,\beta)\ (id,\ C\backslash_\alpha.\alpha :: \Gamma.\beta :: \{\})$
                else $(id, C)$
```

As might be expected, the convention on class TC enables us to reduce σ by properly instantiating σ and normalizing its generic context. Furthermore, the set of reduced generic instances of σ is preserved by r, as formalized in the following lemma:

Lemma 7.1 $\sigma \simeq r(\sigma)$.

Proof: Let $\sigma' = r(\sigma)$. We prove that for all reduced type schemes σ_R, if $\sigma_R \preceq \sigma$ then $\sigma_R \preceq \sigma'$. The converse part follows immediately from the definition of $\preceq$.

By the definition of $\preceq$, $C_{\sigma_R} \Vdash S_0 C_\sigma$ and $\tau_{\sigma_R} = S_0 \tau_\sigma$ for some substitution S_0. In particular, for $(\alpha :: \{TC\ (\kappa\,())\} \cup \Gamma) \in C_\sigma$, we have $C_{\sigma_R} \Vdash S_0(\alpha :: \{TC\ (\kappa\,())\} \cup \Gamma)$. So, $S_0 = S' \circ [\alpha \mapsto \kappa\,\nu]$ for some type ν and substitution S' according to our convention on class TC.

Next, let β be a new type variable and $S = S'[\beta \mapsto \nu][\alpha \mapsto \kappa\,\beta]$. Then $S = S_0$ except on β. Moreover, we have $S\alpha = S(\kappa\,\beta)$ and $(S, C_{\sigma_R}) \preceq (id,\ C_\sigma\backslash_\alpha.\alpha :: \Gamma.\beta :: \{\})$. So, if $mgu\ \alpha\ (\kappa\,\beta)\ (id,\ C_\sigma\backslash_\alpha.\alpha :: \Gamma.\beta :: \{\}) = (\bar{S}, \bar{C})$, then, by Lemma 5.5, $S = S'' \circ \bar{S}$ and $C_{\sigma_R} \Vdash S''\bar{C}$ for some substitution S''. But $C_{\sigma'} = \bar{C}$ and $\tau_{\sigma'} = \bar{S}\tau_\sigma$. So, $\tau_{\sigma_R} = S\tau_\sigma = S''\tau_{\sigma'}$ and hence $\sigma_R \preceq \sigma'$. ∎

Moreover, the two notions of equivalent coincide for reduced type schemes:

Lemma 7.2 For any two reduced type schemes σ_R and σ'_R, $\sigma_R = \sigma'_R$ iff $\sigma_R \simeq \sigma'_R$.

Proof: Follows immediately from their definitions. ∎

Now, let $r^*(\sigma)$ be the reduced type scheme obtained by repeatedly applying r to σ. With the above two lemmas, we can easily prove the following theorem:

Theorem 7.3 For any two type schemes σ_1 and σ_2, $\sigma_1 \simeq \sigma_2$ iff $r^*(\sigma_1) = r^*(\sigma_2)$.

Proof: We prove the "if" part; the proof for the converse part is similar and thus omitted.

By Lemma 7.1, $r^*(\sigma_1) \simeq \sigma_1$ and $\sigma_2 \simeq r^*(\sigma_2)$. So, $r^*(\sigma_1) \simeq r^*(\sigma_2)$ and hence $r^*(\sigma_1) = r^*(\sigma_2)$ by Lemma 7.2. ∎

Consequently, by repeatedly applying r to rewrite type schemes to their reduced forms, we get a decision procedure for the extended generic instance relation.

8 Related Work

Wadler and Blott [WB89] introduced type classes and presented an extension of the Hindley-Milner type system that incorporates them. They proposed a new form of type, called a *predicated type*, to specify the types of overloaded functions. A quite similar notion was used under the name of *category* in the Scratchpad II system for symbolic computation [JT81]. Also related are Kaes' work on parametric overloading [Kae88], F-bounded polymorphism in object-oriented programming [CCH+89], and [Rou90]. The type class idea was quickly taken up in the design of Haskell. Its theoretical foundation, however, took some time to develop. The initial approach of [WB89] encoded Haskell's source-level syntax in a type system that was more powerful than Haskell itself, since it could accommodate classes over multiple types. This increased expressiveness can, however, lead to undecidability, as has been investigated by Volpano and Smith [VS91]. Indeed, the system published in [WB89] is apparently undecidable.

The source-level syntax of Haskell, on the other hand, has a sufficient number of static constraints to guarantee decidability. This was shown in [NS91], where Nipkow and Snelting modeled type classes in a three-level system of values, types, and partially ordered sorts. In their system, classes correspond to sorts and types are sorted according the class hierarchy. Order-sorted unification [MGS89] is used to resolve overloading in type reconstruction. The use of an order-sorted approach is mathematically elegant, yet we argue that the ordering relation between classes is a syntactic mechanism and thus not necessary for developing a type system for type classes. Furthermore, it is not obvious how to extend their system to incorporate our proposed extensions.

Work was also done to extend the type class concept to predicates over multiple types. Volpano and Smith [VS91] looked into modifications of the original system in [WB89] to ensure decidability of type reconstruction and to get a sharper notion of well-typed expressions. Jones [Jon91, Jon92b] gave a general framework for *qualified types.* His use of predicate sets is at first sight quite similar to our context-constrained instance theory. The main difference between the two approaches lies in our use of normal forms (Jones does not address this issue) and our distinction between constrained and dependent variables. This distinction allows us to solve the ambiguity problems previously encountered in definitions of container classes. Recently, based on the notion of *constrained types*, Kaes [Kae92] presented a generic inference system of which overloading and subtyping are special instances.

9 Conclusion

We have proposed a generalization of Haskell's type classes to support container classes with overloaded data constructors and selectors. The underlying type system is an extension of the Hindley-Milner type system with parametric type classes. This extension preserves two important properties of the original system, namely decidable typability and principal types. Its type scheme uses bounded quantification whose introduction and elimination depend on a separate context-constrained instance theory. The decoupling of the instance theory from the type inference system makes our system more modular than previous work. We believe that the gained modularity can also be a great aid to implementors.

Compared to other approaches, our type system follows closest in spirit to the design of Haskell's type classes. Indeed, the use of normalized context has its root in the source-level syntax of Haskell. The instant normalization of context constraints matches well with the $C-T$ rule stated in the

Haskell report, which requires that a $C-T$ instance declaration for class C and type T appear either in the module in which C is declared or in the module in which T is declared. From a pragmatic viewpoint, the $C-T$ rule is a reasonable requirement for systems supporting separate compilation to handle globally overloaded operators. But for type systems with general predicate sets like qualified types or constrained types, it seems difficult to find such a proper rule for them without being overly restrictive.

On the other hand, we associate context constraints with individual quantified type variables in forming a type scheme. By doing so, we have restricted the expressiveness of our system. For instance, we exclude *cylic contexts* from our discussion. Therefore, in the future, we plan to investigate the feasibility of having a context set in our type scheme.

Acknowledgement

We wish to thank Mark Jones, Stefan Kaes, Tobias Nipkow, and Satish Thatte for their critical comments and helpful discussions.

References

[CCH$^+$89] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proc. ACM Conf. Functional Programming Languages and Computer Architecture*, pages 273–280, 1989.

[CHO92] Kung Chen, Paul Hudak, and Martin Odersky. Parametric type classes. In *Proc. ACM Conf. on Lisp and Functional Programming.* ACM, June 1992.

[DM82] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. 9th ACM Symp. on Principles of Programming Languages*, pages 207–212, Jan. 1982.

[HJW91] Paul Hudak, Simon Peyton Jones, and Philip L. Wadler. Report on the programming language Haskell: a non-strict, purely functional language, version 1.1. Technical Report YALEU/DCS/RR-777, Dept. of Computer Science, Yale University, New Haven, Conn., August 1991.

[Jon91] Mark P. Jones. Type inference for qualified types. Technical Report PRG-TR-10-91, Oxford University Computing Laboratory, Oxford, UK, 1991.

[Jon92a] Mark P. Jones. Coherence for qualified types. Private communication, March 1992.

[Jon92b] Mark P. Jones. A theory of qualified types. In B. Krieg-Brückner, editor, *Proc. European Sysposium on Programming*, pages 287–306. Springer Verlag, Feburary 1992. LNCS 582.

[JT81] R.D. Jenks and B.M. Trager. A language for computational algebra. In *Proc. ACM Symposium on Symbolic and Algebraic Manipulation*, pages 22–29, 1981.

[Kae88] S. Kaes. Parametric overloading in polymorphic programming languages. In H. Ganzinger, editor, *Proc. 2nd European Symosium on Programming, Lecture Notes in Computer Science, Vol. 300*, pages 131–144, Nancy, France, March 1988. Springer-Verlag.

[Kae92] Stefan Kaes. Type inference in the presence of overloading, subtyping abd recursive types. In *Proc. ACM Conf. on Lisp and Functional Programming.* ACM, June 1992.

[Lil91] Mark D. Lilibridge. A generalization of type classes. distributed to Haskell mailing list, June 1991.

[MGS89] J. Meseguer, J. Goguen, and G. Smolka. Order-sorted unification. *J. Symbolic Computation*, 8:383–413, 1989.

[NS91] T. Nipkow and G. Snelting. Type classes and overloading resolution via order-sorted unification. In J. Hughes, editor, *Proc. Conf. on Functional Programming and Computer Architecture*, pages 15–28. Springer-Verlag, 1991. LNCS 523.

[Rob65] J. Robinson. A machine-oriented logic based on the resolution principle. *J. Assoc. Comput. Mach.*, 12(1):23–41, 1965.

[Rou90] Francois Rouaix. Safe run-time overloading. In *Seventeenth Annual ACM Symp. on Principles of Programming Languages*, pages 355–366, San Franscico, CA, January 1990.

[VS91] Dennis M. Volpano and Geoffery S. Smith. On the complexity of ML typability and overloading. In J. Hughes, editor, *Proceedings of Functional Programming and Computer Architecture*, pages 15–28. Springer-Verlag, 1991. LNCS 523.

[WB89] Phil Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Sixteenth Annual ACM Symp. on Principles of Programming Languages*, pages 60–76. ACM, 1989.