Parametric Type Classes
(Extended Abstract)

Kung Chen, Paul Hudak, Martin Odersky*

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

ACM Conf. on LISP and Functional Programming, June 1992

Abstract

We propose a generalization to Haskell's type classes where a class can have type parameters besides the placeholder variable. We show that this generalization is essential to represent container classes with overloaded data constructor and selector operations. We also show that the resulting type system has principal types and present unification and type reconstruction algorithms.

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).

*This research was supported by DARPA through ONR contracts N00014-90-C-0024 and N00014-91-J-4043.

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`). 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 Section 7 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.

Our Contributions

To solve these problems, we introduce the notion of *parametric type classes* as a significant generalization of Haskell's type classes. Our contributions can be summarized as follows:

1. Parametric type classes can have type arguments in addition to the constrained type variable, and thus are able to express classes such as `Sequence` defined earlier.
2. 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`.
3. Parametric type classes are a conservative extension of Haskell's type system: If all classes are parameterless, the two systems are equivalent.
4. We prove that our system is decidable, and provide an effective type inference algorithm.
5. As a concrete demonstration of the power and practicality of the system, we formulate classes *monad* and *monad0* that allow us to generalize the concept of list comprehensions to monads. This is done using the standard translation rules for list comprehensions; no special syntax is needed.

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.

The rest of this paper 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 discusses type reconstruction and unification. Section 6 explains when a type scheme is ambiguous. Section 7 applies our system in defining monads as parametric classes. Section 8 conclues.

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
```

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 6 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.

For lack of space, we refer the reader to [COH92] for detailed proofs of the results presented in this and the following sections.

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 $c\,\tau$.

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 section, 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 $C \Vdash \tau ::$

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	$::=$	$\{c_1\,\tau_1, ..., c_n\,\tau_n\}$	$(n \geq 0, c_i$ pairwise disjoint)
Contexts	C	$::=$	$\{\alpha_1 :: \Gamma_1, \ldots, \alpha_n :: \Gamma_n\}$	$(n \geq 0)$
Expressions	e	$::=$	$x \mid e\ e' \mid \lambda x . e \mid \textbf{let}\ x = e'\ \textbf{in}\ e$	
Programs	p	$::=$	$\textbf{class}\ \alpha :: \gamma\ \textbf{where}\ x : \sigma\ \textbf{in}\ p$	
		$\mid$	$\textbf{inst}\ C \Rightarrow \tau :: \gamma\ \textbf{where}\ x = e\ \textbf{in}\ p$	
		$\mid$	e	

Figure 1: Abstract Syntax of Mini-Haskell+

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

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

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

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

Figure 2: Inference Rules for Entailment

γ. 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 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

$$C^*(\Delta) = \Delta \cup C(C^*(\Delta)).$$

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. 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 paper.

Constrained Substitution

In the following, we will apply variable 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 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 $S = R \circ S_0$ 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 **inst** $C \Rightarrow \alpha :: c\,\tau$.

(b) For every pair of type and class constructor (κ, c), there is at most one instance declaration of the form **inst** $C \Rightarrow \kappa\,\tau' :: c\,\tau$. 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.

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 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 and moved into the type scheme $\forall \alpha :: \Gamma.\sigma$.

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 $S\tau = \tau'$, α'_j is not free in σ, and $C \uplus \{\alpha'_j :: \Gamma'_j\} \Vdash S\alpha_i :: S\Gamma_i$. We write in this case, $\sigma' \preceq_C \sigma$.

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.

The following property is a direct consequence of the definition.

Lemma 3.2 If $\sigma' \preceq_C \sigma$ and $C \preceq C'$ then $\sigma' \preceq_{C'} \sigma$.

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

Lemma 3.3 If $\sigma' \preceq_C \sigma$ and $C' \Vdash SC$ then $S\sigma' \preceq_{C'} S\sigma$.

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

(var)	$A, C \vdash x : \sigma \qquad (x : \sigma \in A)$	
$(\forall\!-\!elim)$	$\dfrac{A, C \vdash e : \forall\alpha :: \Gamma.\sigma \qquad C \Vdash \tau :: \Gamma}{A, C \vdash e : [\alpha \mapsto \tau]\,\sigma}$	
$(\forall\!-\!intro)$	$\dfrac{A, C.\alpha :: \Gamma \vdash e : \sigma}{A, C \vdash e : \forall\alpha :: \Gamma.\sigma}$	$(\alpha \notin fv\ A \cup reg\ C)$
$(\lambda\!-\!elim)$	$\dfrac{A, C \vdash e : \tau' \to \tau \qquad A, C \vdash e' : \tau'}{A, C \vdash e\ e' : \tau}$	
$(\lambda\!-\!intro)$	$\dfrac{A.x : \tau', C \vdash e : \tau}{A, C \vdash \lambda x.e : \tau' \to \tau}$	
(let)	$\dfrac{A, C \vdash e' : \sigma \qquad A.x : \sigma, C \vdash e : \tau}{A, C \vdash \mathbf{let}\ x = e'\ \mathbf{in}\ e : \tau}$	

Figure 3: Typing Rules for Expressions

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

Figure 4: Typing Rules for Declarations

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

$$\begin{array}{l} gen\ (\sigma, A, C) = \\ \quad \mathbf{if}\ \exists\alpha \in dom(C)\backslash(fv\ A \cup reg\ C)\ \mathbf{then} \\ \qquad gen\ (\forall\alpha :: C\alpha.\sigma, A, C\backslash_\alpha) \\ \quad \mathbf{else}\ (\sigma, C) \end{array}$$

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.

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.

Lemma 4.1 (Substitution lemma) If $A, C \vdash' e : \tau$ and $C' \Vdash SC$ then $SA, C' \vdash' e : S\tau$.

This result assures us that typing derivations are preserved under constrained substitution.

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

Lemma 4.2 If $A, C \vdash' e : \tau$ and $C \preceq C'$ then $A, C' \vdash' e : \tau$.

Lemma 4.3 If $A.x : \sigma, C \vdash' e : \tau$ and $\sigma \preceq_C \sigma'$ then $A.x : \sigma', C \vdash' e : \tau$.

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

Theorem 4.4 If $A, C \vdash' e : \tau$ then $A, C \vdash e : \tau$.

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

Figure 5: Determinstic Typing Rules for Expressions

Theorem 4.5 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 $(\sigma', C'') = gen(\tau, A, C')$.

5 Unification and Type Reconstruction

This section discusses type reconstruction. As usual, type reconstruction relies on unification, and we will first work out what kind of unification is needed for parametric type classes. We then go on 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 the last section 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.

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 $mgn\ \tau\ \Gamma\ (S_0, C_0)$ computes a most general normalizer of $C_0 \cup \{\tau :: \Gamma\}$, provided one exists.

Theorem 5.1 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.

Type Reconstruction

An algorithm for type reconstruction is shown in Figure 7.[1] 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 begin by showing that *tp* is indeed a constrained substitution transformer.

Lemma 5.2 Let (S, C) be a constrained substitution and $(\tau, S', C') = tp(e, A, S, C)$, then (S', C') is a constrained substitution.

Hence we will omit the requirement of constrained substitution from now on.

Lemma 5.3 If $tp(e, A, S, C) = (\tau, S', C')$ then $(S', C') \preceq (S, C)$.

This result can be established by a straightforward induction except in the **let**–case. 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**–case in *tp* we would overgeneralize those instance assumptions generated in the previous stages and passed to *tp* as part of the initial context.

[1] 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.

$$
\begin{array}{lcl}
\mathit{mgu} : \tau \rightarrow \tau \rightarrow S \times C \rightarrow S \times C & & \\
\mathit{mgn} : \tau \rightarrow \Gamma \rightarrow S \times C \rightarrow S \times C & & \\[1ex]
\mathit{mgu}\ \tau_1\ \tau_2\ (S, C) & = & \mathit{mgu}'\ (S\tau_1)\ (S\tau_2)\ (S, C) \\[1ex]
\mathit{mgu}'\ \alpha\ \alpha & = & id_{S \times C} \\
\mathit{mgu}'\ \alpha\ \tau\ (S, C)\ \mid\ \alpha \notin \mathit{fv}(\tau) & = & \mathit{mgn}\ \tau\ (C\alpha)\ ([\alpha \mapsto \tau] \circ S, [\alpha \mapsto \tau] C \backslash_{\alpha}) \\
\mathit{mgu}'\ \tau\ \alpha\ (S, C) & = & \mathit{mgu}\ \alpha\ \tau\ (S, C) \\
\mathit{mgu}'\ ()\ () & = & id_{S \times C} \\
\mathit{mgu}'\ \kappa\, \tau\ \kappa\, \tau'\ (S, C) & = & \mathit{mgu}\ \tau\ \tau'\ (S, C) \\
\mathit{mgu}'\ (\tau_1 \times \tau_2)\ (\tau_1' \times \tau_2') & = & (\mathit{mgu}\ \tau_1\ \tau_1') \circ (\mathit{mgu}\ \tau_2\ \tau_2') \\
\mathit{mgu}'\ (\tau_1 \rightarrow \tau_2)\ (\tau_1' \rightarrow \tau_2') & = & (\mathit{mgu}\ \tau_1\ \tau_1') \circ (\mathit{mgu}\ \tau_2\ \tau_2') \\[1ex]
\mathit{mgn}\ \tau\ \{\} & = & id_{S \times C} \\
\mathit{mgn}\ \tau\ \{\gamma\}\ (S, C) & = & \mathit{mgn}'\ (S\tau)\ (S\gamma)\ (S, C) \\
\mathit{mgn}\ \tau\ (\Gamma_1 \cup \Gamma_2) & = & (\mathit{mgn}\ \tau\ \Gamma_1) \circ (\mathit{mgn}\ \tau\ \Gamma_2) \\[1ex]
\mathit{mgn}'\ \alpha\ c\ \tau\ (S, C) & = & \textbf{if}\ \exists \tau'.(c\ \tau' \in C\alpha)\ \textbf{then}\ \mathit{mgu}\ \tau\ \tau'\ (S, C) \\
& & \textbf{else}\ (S, C[\alpha \mapsto C\alpha \cup \{c\ \tau\}]) \\
\multicolumn{3}{l}{\mathit{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' \ =\ \mathit{match}\ \tilde{\tau}'\ \tau' \\
& & \qquad (S'', C'')\ =\ \mathit{mgu}\ \tau\ (S'\tilde{\tau})\ (S, C) \\
& & \qquad \{\tau_1 :: \Gamma_1, \ldots, \tau_n :: \Gamma_n\}\ =\ S'C' \\
& & \textbf{in}\ (\mathit{mgn}\ \tau_1\ \Gamma_1\ (\ \ldots\ (\mathit{mgn}\ \tau_n\ \Gamma_n\ (S'', C'')))) \\[1ex]
\multicolumn{3}{l}{\text{(and similarly for } \rightarrow, \times, ())}
\end{array}
$$

Figure 6: Unification and Normalization Algorithms

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. We define a new generalization function, *tpgen*, which, 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. Thus only those newly generated instance assumptions will be generalized.

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

Theorem 5.4 If $tp(e, A, S, C) = (\tau, S', C')$ then $S'A, C' \ \vdash' e : \tau$.

Together with Theorem 4.4, we have the following soundness result.

Corollary 5.5 (Soundness of tp) If $tp(e, A, S, C) = (\tau, S', C')$ then $S'A, C' \ \vdash\ e : \tau$.

Ultimately, we will state the principal typing result.

Theorem 5.6 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

$$S'A = RSA, \quad C' \Vdash RC, \quad \text{and} \quad \tau' = R\tau.$$

Together with Theorem 4.5, we have the completeness result.

Corollary 5.7 (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

$$S'A = RSA, \quad \text{and} \quad \sigma' \preceq_{C'} R\sigma$$

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

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

Corollary 5.8 Suppose that $tp(e, A, S_0, C_0) = (\tau, S, C)$ and $gen(\tau, SA, C) = (\sigma, C')$. Then σ is a principal type scheme for e under SA and C'.

6 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.

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) $\alpha \notin C_\sigma^*(\mathit{fv}\ \tau)$.

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

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

Figure 7: Type Reconstruction Algorithm

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 implicitely 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[2]

$$
\begin{array}{ll}
\texttt{map} : & \forall a.\forall b.\forall t. \\
 & \forall sa :: \{Sequence\ a,\ TC\ t\}. \\
 & \forall sb :: \{Sequence\ b,\ TC\ t\}.(a \to b) \to sa \to sb
\end{array}
$$

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

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:

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

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

$$(a \rightarrow b) \rightarrow List\ a \rightarrow List\ b$$

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

To keep contexts short, we will use in the next section the similarity relation ($\sim$) directly, instead of its definition in terms of TC.

7 From Monads to Lists

In this section, we show how to use parametric type classes to generalize many of the operations and concepts which were previously restricted to lists. As sketched in the introduction, a first step overloads operations that are common to all implementations of sequence. Some important operations can even be applied in the more general Moand context[Wad90]; hence it makes sense to have "Monad" and "Monad with zero" as superclasses of "Sequence". The following enumeration shows on which levels in the hierarchy some familiar list operations are defined.

Monad: `unit, join, map`, monad comprehensions.

Monad0: `nil, filter`, comprehensions with filters.

Sequence: `cons, hd, tl, reverse, foldl, foldr, (++)`.

The use of monads in functional programming was explored in [Wad90, Wad91]; for a motivation of the concept we refer the reader to the examples given there. The point we want to explore here is how to express monads (and their specializations) in the type system of a programing language such that we can abstract from their concrete implementations. We show how the monad operations can be overloaded, using parametric type classes. This is useful since it allows to define functions over arbitrary Monads, to reuse the same names for operations on different monads, and to generalize list comprehensions without changing their present syntax.

We formulate class `Monad` as follows:

```
  class ma :: Monad a where
    unit :: a -> ma
    bind :: (mb :: Monad b, ma ~ mb)
            => ma -> (a -> mb) -> mb
    map  :: (mb :: Monad b, ma ~ mb)
            => (a -> b) -> ma -> mb
    join :: (mma :: Monad ma, mma ~ ma)
            => mma -> ma
-- Default definitions:
    map f xs  = xs ‘bind (unit . f)
    join xss  = xss ‘bind‘ id
    bind xs f = join (map f xs)
```

This introduces two equivalent formulations of a monad, one in terms of `unit` and `bind`, the other in terms of `unit`, `map` and `join`. The default definitions in the class express one formulation in terms of the other; hence instances can alternatively define `bind` or `map` and `join`. To qualify for a monad, an instance has to satisfy three laws, which are not enforced by the type system. *bind* must be associative, with *unit* as left and right unit:

```
(m ‘bind‘ f) ‘bind‘ g = m ‘bind‘ \x -> f x ‘bind‘ g
\x -> unit x ‘bind‘ f = f
 m ‘bind‘ unit        = m
```

Lists form a monad, as witnessed by the following instance declaration, and a check that monad laws hold:

```
  inst List a :: Monad a where
    unit x         = [x]
    map f [] xs    = []
    map f (x:xs)   = f x : map f xs
    join []        = []
    join (xs::xss) = xs ++ join xss
```

Another example of a monad are "reply"-types, as witnessed by:

```
  data Maybe a = Some a | None

  inst Maybe a :: Monad a where
    unit x            = Some x
    bind (Some x) f   = f x
    bind None f       = None
```

As a consequence, code can now be written that works on lists as well as on reply types or any other monad instance. In particular, we can use the list comprehension notation in each case, by applying the standard translation to *unit*, *map* and *join*:

$$\begin{array}{lcl} [t] & \hat{=} & \mathit{unit}\ t \\ [t \mid g_1, g_2] & \hat{=} & \mathit{join}\,[\,[t \mid g_2] \mid g_1] \\ [t \mid x \leftarrow e] & \hat{=} & \mathit{map}\ (\lambda x.t)\ e \end{array}$$

Here, t and e are terms, x is a variable, and g_1 and g_2 are generators $x \leftarrow e$.

`Monad0` is a subclass of `Monad`. It adds a zero monad, `nil`, and a `filter` function.

```
class (ma :: Monad a) => ma :: Monad0 a where
  nil    :: ma
  filter :: (a -> Bool) -> ma -> ma
```

Monads with zero are the most general type class on which list comprehensions with filters can be defined. The standard translation functions are (p is a filter, i.e. a Boolean term):

$$\begin{array}{lcl} [\,] & \hat{=} & \mathit{nil} \\ [t \mid p] & \hat{=} & \mathit{filter}\ p\ (\mathit{unit}\ t) \end{array}$$

Lists and reply types both have zeros, as witnessed by:

```
instance List a :: Monad0 a where
  nil               = []
  filter p []       = []
  filter p (x:xs)   = if p x then
                        x : filter p xs
                      else filter p xs

instance Maybe a : Monad0 a where
  nil               = None
  filter p None     = None
  filter p (Some x) = if p x then Some x
                      else None
```

As an example of programming with Monads we discuss abstract parsers, adapting and extending an example from [Wad90]. A parser is a function that maps a sequence of input symbols to some output, or to a failure value, if no legal parse exists. If a parse exists, then it will consist of the unused portion of the input stream, plus some application dependent result value, such as a parse tree. If the parser uses backtracking, there might exist several such parses, whereas if it is determinstic, there will be zero or one. We construct in the following a library for determinstic parsers. Such parsers all have type signature:

```
data Parser a = P (String -> Maybe (a, String))
```

The constructor tag `P` is necessary because of the restriction that instances may only be formed of datatypes. Parsers form themselves a monad with zero, as witnessed by the following instance declarations.

```
inst Parser a :: Monad a where
  unit x      = P (\i -> [(x, i)])
  map f (P p) = P (\i ->
                  [(f x, i') | (x, i') <- p i])
  join (P pp) = P (\i ->
                  [(x, i'') | (P p, i') <- pp i,
                              (x, i'') <- p i'])

inst Parser a :: Monad0 a where
  nil          = P (\i -> [])
  filter b (P p)
               = P (\i ->
                    [(x, i') | (x, i') <- p i,
                               b x])
```

Note that we have overloaded the comprehension notation. The monad comprehensions in the previous two instance declarations work on option types, not lists.

We need two primitive parsers and one more parser combinator:

```
sym         :: Parser Char
sym         =  P p
               where p Nil         = []
                     p (Cons c cs) = [(c, cs)]

lookahead   :: Parser Char
lookahead   =  P p
               where p Nil = []
                     p cs  = [(hd cs, cs)]

(|||)       :: Parser a -> Parser a -> Parser a
P p ||| P q =  P (\i -> case p i of
                          None   => q i
                        | Some x => Some x)
```

A deterministic parser for lambda terms can then be written as follows:

```
data Term = Lambda Term Term
          | Apply Term Term
          | Id Char
          | Error

term     :: Parser Term
term     =  [Lambda x y | '\' <- sym,
                          x <- ident, y <- term]
        ||| [y | x <- aterm, y <- aterms x]

aterm    :: Parser Term
aterm    =  [x | '(' <- sym, x <- aterm']
        ||| ident

aterm'   :: Parser Term
aterm'   =  [x | x <- term, ')' <- sym]
        ||| [Error]

aterms   :: Term -> Parser Term
aterms x =  [z | c <- lookahead,
                 'a' <= c && c <= 'z' || c = '(',
                 y <- aterm,
                 z <- aterms (Apply x y)]
        ||| [x]

ident    :: Parser Term
ident    =  [Id c | c <- sym, 'a' <= c && c <= 'z']
        ||| [Error]
```

The defined parser is determinstic; it never backtracks. Therefore, parse failure has to be treated differently according to whether it occurs at the start of a production, or in the middle. If failure occurs at the start of a production, it signals that another alternative should be tried. Failure in the middle of a production signals a syntax error that is reported by returning an `Error` node.

Note that most of the productions are expressed in terms of monad comprehensions. This time, comprehensions refer to parsers instead of option types or lists. Unlike in [Wad90], monad comprehensions need not be labelled with the monad they refer to; we rely instead on the type system for disambiguation (including programmer defined typings if ambiguities arise otherwise). The monad style gives us a flexible interface between parsing and abstract tree generation. The resulting parser resembles an attribute grammar with both synthesized and inherited attributes (see the definition of `aterm`).

8 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 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.

A point we have not discussed so far is how to implement parametric type classes at run-time. Essentially, a translation scheme into Haskell along the lines of [WB89] can be employed. Additional parameters for type classes translate then into parameters for run-time dictionaries. Such a translation can provide a (transformational) semantics for parametric type classes. Whether it can also provide a good run-time model is debatable. Existing implementations that are based on this translation scheme have been criticized for their run-time performance. We argue that, in principle, the run-time performance of a program with type classes should not be any worse than the performance of a program written in an object-oriented language. Moreover, similar optimization techniques can be used [CU90].

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.

[COH92] Kung Chen, Martin Odersky, and Paul Hudak. Type inference for parametric type classes. Technical Report YALEU/DCS/RR-900, Dept. of Computer Science, Yale University, New Haven, Conn., May 1992.

[CU90] Craig Chambers and David Ungar. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. In *Proc. SIGPLAN '90 Conf. on Programming Language Design and Implementation*, White Plains, New York, June 1990.

[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.

[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.

[Wad90] P. Wadler. Comprehending monads. In *Proc. ACM Conf. on LISP and Functional Programming*, pages 61–78, June 1990.

[Wad91] P. Wadler. Continuing monads, August 1991. Tutorial Notes at FPCA'91.

[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.