JFP **12** (2): 133–158, March 2002.
DOI: 10.1017/S0956796801004233 Printed in the United Kingdom

Type-checking multi-parameter type classes

DOMINIC DUGGAN

Department of Computer Science, Stevens Institute of Technology, Castle Point on the Hudson, Hoboken, NJ 07030, USA

JOHN OPHEL

Department of Computer Science and Computer Engineering, La Trobe University, Bundoora, Victoria 3083, Australia

Abstract

Type classes are a novel combination of parametric polymorphism and constrained types. Although most implementations restrict type classes to be single-parameter, the generalization to multi-parameter type classes has gained increasing attention. A problem with multi-parameter type classes is the increased possibilities they introduce for ambiguity in inferred types, impacting their usefulness in many practical situations. A new type-checking strategy, *domain-driven unifying resolution*, is identified as an approach to solve these problems. Domain-driven unifying resolution is simple, efficient, and practically useful. However, even with severe restrictions on instance definitions, it is not possible to guarantee that type-checking with unifying resolution terminates. This is in contrast with the naive generalization of single parameter resolution strategies. Domain-driven unifying resolution is guaranteed to terminate if the type class constraints are satisfiable; however satisfiability is undecidable even with severe restrictions on instance definitions. These results shed some light on ambiguity problems with multi-parameter type classes.

Capsule Review

Haskell's type classes have proven to be a useful and powerful extension to conventional parametric polymorphism, providing a disciplined form of *ad hoc* polymorphism, or overloading. One problem in their use, however, is the possibility of type *ambiguity*, i.e. the type inference algorithm is not complete. This problem is particularly acute when *multi-parameter* type classes are introduced, which is a popular extension currently supported by several Haskell implementations. This paper proposes a new algorithm that resolves overloading and avoids ambiguous types under certain well-defined, although undecidable, conditions.

1 Introduction

Parametric overloading refers to a form of constrained genericity, based on the combination of parametric polymorphism and overloading (Kaes, 1988). Parametric polymorphism allows a function to abstract over the types of some of its arguments, with the actual type arguments implicitly supplied at the call sites for the function. Parametric overloading generalizes this by allowing a function to abstract over the overloaded operations used in its definition (at least, those occurrences which depend on undetermined types), with the actual implementations implicitly constructed at the call site. For example, if the `double` function is defined by:

```
double x = x + x
```

then the type of `double` reflects that its use depends on there being an implementation of addition available for any arguments to which `double` is applied. This mechanism is realized in the Haskell language through the use of type classes (Hall *et al.,* 1996). A type class consists of a declaration of the operations which inhabit the class, and a collection of instance declarations which provide implementations of that class for various types. Uses of overloaded operations give rise to overloading (class) constraints in polymorphic types. For example:

```
class Plus α where (+) :: α → α → α
instance Plus Int where (+) = integerPlus
instance Plus α ⇒ Plus (Vector α) where x + y = ...
```

Polymorphic types have the form $\forall\overline{\alpha}.C \Rightarrow \tau$, where C is a *context* of class constraints restricting the possible instantiations of the type variables. For example, the type of `double` is $\forall\alpha$.`Plus` $\alpha \Rightarrow \alpha \rightarrow \alpha$. We refer to the expression `Plus` α in this type as a *type constraint*, and we refer to `Plus` in this expression as the *type predicate* of the constraint.

Type-checking for Haskell type classes is difficult in general. Wadler & Blott (1989) presented a very general type system; the actual Haskell type system incorporates a restricted version of this, in which only single-parameter type classes are allowed (class declarations can only declare classes of the form $C(\alpha)$). Even for this restricted case, Volpano & Smith (1991) showed that type-checking is undecidable without further restrictions. For the case of the single-parameter Haskell restrictions, several algorithms have been developed (Nipkow & Snelting, 1991; Chen *et al.*, 1992; Nipkow & Prehofer, 1993).

The situation for type-checking with multi-parameter type classes is less clear. Nevertheless it is worth considering the addition of multi-parameter type classes to Haskell, as evidenced by some of the applications that have been cited:

Linear Algebra (Cormack & Wright, 1990): linear algebra operations such as matrix and vector multiplication and addition can be succinctly coded using multi-parameter type classes, for example:

```
class Plus α β γ where (+) :: α → β → γ
class Times α β γ where (*) :: α → β → γ
instance Times Int Int Int where ...
instance Times Int Float Float where ...
instance (Times α β γ, Plus γ γ γ) ⇒
    Times (Matrix α) (Matrix β) (Matrix γ) where ...
instance (Times α β γ, Plus γ γ γ) ⇒ Times (Vector α) (Vector β) γ where ...
instance Times α β γ ⇒ Times α (Matrix β) (Matrix γ) where ...
```

Collection Types (Jones, 2000): a class for collection types can be specified using multi-parameter type classes:

```
class Collects α β where
   empty :: α
   member :: β → α → Bool
   insert :: β → α → α
```

https://doi.org/10.1017/S0956796801004233 Published online by Cambridge University Press

Instances can then be specified with types:

```
instance Eq α ⇒ Collects [α] α where ...
instance Eq α ⇒ Collects (α → Bool) α where ...
instance Collects BitSet Char where ...
instance (Hashable α, Collects α β) ⇒ Collects (Array Int β) α where ...
```

Record Field Selectors (Peyton-Jones et al., *1997):* various approaches have been suggested for encoding records using datatypes and type classes. Consider, for example, the definition of a record type:

```
Employee = EMPLOYEE{name::String address::String age::Int}
```

This expands to the definition of the record representation as a datatype. Record field selectors are defined as overloaded operations, that dispatch based on the record type (Jones, 1992):

```
data Employee = EMPLOYEE String String Int
class Name α β where name :: α → β
instance Name Employee String where name(EMPLOYEE x y z) = x
class Address α β where address :: α → β
instance Address Employee String where address(EMPLOYEE x y z) = y
class Age α β where age :: α → β
instance Age Employee String where age(EMPLOYEE x y z) = z
```

Monad Transformers (Liang et al.*, 1995):* monad transformers were proposed as a way of structuring modular software systems. These rely on multi-parameter type classes for their definition, for example:

```
class (Monad β, Monad (α β)) ⇒ MonadT α β where ...
instance (Monad β, Monad (StateT α β)) ⇒ MonadT (StateT α) β where ...
class Monad β ⇒ EnvMonad α β where ...
instance Monad β ⇒ EnvMonad α (EnvT α β) where ...
```

Peyton-Jones *et al.* (1997) provide various other examples of monadic definitions that can be expressed succinctly using multi-parameter type classes. In contrast to the other examples provided here, monadic definitions and monad transformers do not give rise to the problems addressed by this article. We include the example of monad transformers because it is an interesting illustration of the usefulness of multi-parameter type classes. Although for simplicity and economy we do not consider type constructor classes, our results generalize straightforwardly to this extension.

The Gofer language pioneered an implementation of multi-parameter type classes (Jones, 1991), although early experiments reported negative experience because of the problems addressed in this article. More recently the GHC and HUGS Haskell compilers have incorporated multi-parameter type classes as experimental extensions, with the intention that these eventually be incorporated in the Haskell 2 language design. Some descriptions of the rationale underlying these implementations are provided by Peyton-Jones *et al.* (1997) and Peyton-Jones (1998).

The issue addressed by this article, that has hindered the usefulness of multi-parameter type classes, is the increased potential they introduce for ambiguous

https://doi.org/10.1017/S0956796801004233 Published online by Cambridge University Press

typing. 'Ambiguity' refers to the situation where there is a free type variable in the overload constraints accumulated during type inference, with no occurrence of that type variable in the inferred type, and therefore no possibility (apparently) that that type variable can be resolved further. For example, with the operations `read :: Read` $\alpha \Rightarrow$ `String` $\rightarrow \alpha$ and `show :: Show` $\alpha \Rightarrow \alpha \rightarrow$ `String`, the expression `show (read "123")` is ambiguous: there is no way to determine how the string is read and then written (for example, is it converted to `Int`, or read as a string of digits?). This ambiguity is signalled by the type `(Read` α`, Show` α`)` $\Rightarrow$ `String`. Since there is no unique 'most general' translation of a program with ambiguity, type-checking fails when it arises.

In section 2, we demonstrate with practical examples how multi-parameter type classes can give rise to the increased potential for ambiguity, impacting the usefulness of multi-parameter type classes for some compelling examples. In section 3 we provide a type system for multi-parameter type classes, and we consider a restriction on multi-parameter type classes, the *overlapping restriction*, that is necessary in its own right. With this restriction, we are able in section 4 to provide an overload resolution strategy, *domain-driven unifying resolution*, that solves the ambiguity problems with multi-parameter type classes (for the applications we are interested in). Domain-driven unifying resolution is verified to be correct, and is guaranteed to terminate if the program is well-typed.

The overlapping restriction (in its weak form) states that in a class declaration:

`class` $C\ \alpha_1\ \ldots\ \alpha_m\ \beta_1\ \ldots\ \beta_n$ `where ...`

the instantiations of the type parameters $\alpha_1, \ldots, \alpha_m$ should uniquely determine the instantiations of the type parameters $\beta_1, \ldots, \beta_n$, for some m and n declared by the programmer. For example, for the `Times` class given earlier, set $m = 2$ and $n = 1$; then the first two type parameters to the `Times` type predicate (representing the domain types of the multiplication operator) determine the third type parameter (representing the range type of the operator). This example is similar to the C++ and Java restrictions on overloading function names. The overlapping restriction allows the programmer the flexibility to choose the amount of determination, and this information is in turn used by the overload resolution algorithm to resolve ambiguity problems.

This approach can be said to subsume other approaches, particularly the approach of parametric type classes (Chen *et al.,* 1992). The latter requires class declarations to have the form:

`class` $C\ \alpha\ \beta_1\ \ldots\ \beta_n$ `where ...`

where the collection type α determines the element types $\beta_1, \ldots, \beta_n$. Parametric type classes are an extended version of single-parameter type classes; all instance types are parametric in the type parameters $\beta_1, \ldots, \beta_n$, and are indexed by the single type parameter α. One contrast between this and the overlapping restriction for multi-parameter type classes is given by the fact that overload resolution is guaranteed to terminate for the former, whereas it may fail to terminate for the latter. This is so even for a multi-parameter type class that superficially resembles a parametric type class. We elaborate on this in the next section.

https://doi.org/10.1017/S0956796801004233 Published online by Cambridge University Press

It should be emphasized that our results do not make the closed world assumption that all instance types are known. Domain-driven unifying overload resolution does not over-aggressively resolve against the known instances, without considering the possibility that further instances might be added. The overlapping restriction (which is necessary in any case for coherence) ensures that if it is possible to resolve a constraint against an instance because of unifying domain types, then it will not be possible later to define another instance that might have alternatively been used in resolution.

It should also be emphasized that the overlapping restriction does not rule out interesting instance types. As explained in section 7, the restriction allows the programmer to choose her own trade-off between the amount of restriction on the forms of instance types, and the level of unifying resolution used during type inference. The more restricted are the instance types, the greater the degree of unifying resolution that is possible.

The linear algebra example raises an interesting point about overlapping instances:

```
instance (Times α β γ, Plus γ γ γ) ⇒
    Times (Matrix α) (Matrix β) (Matrix γ) where ...
instance Times α β γ ⇒ Times α (Matrix β) (Matrix γ) where ...
```

Our overlapping restriction prevents the expression of these instance types, because the first instance type is a substitutive instance of the second instance type. In our type system, we resolve this ambiguity by adding *negative context constraints* to the type system:

```
instance (Times α β γ, Plus γ γ γ) ⇒
    Times (Matrix α) (Matrix β) (Matrix γ) where ...
instance (Times α β γ, α ≠ (Matrix δ)) ⇒
    Times α (Matrix β) (Matrix γ) where ...
```

The negative context constraint `α ≠ (Matrix δ)` ensures that α cannot be instantiated to a type with `Matrix` as its outermost type constructor.

Our results shed some light on the implications of attempting to solve the problems with ambiguity with multi-parameter type classes. In section 4 we show that overload resolution may fail to terminate for a program that is not typable, even with strong restrictions on the forms of (multi-parameter) overload instance types. Furthermore, although typability ensures termination, in section 5 we show that typability is undecidable. These results suggest that an attempt to solve the ambiguity problems with multi-parameter type classes may have to deal with a potentially non-terminating overload resolution algorithm, for example by using a depth bound during overload resolution. Section 6 considers related work, while section 7 provides our conclusions.

2 Difficulties with simple overload resolution

In ML it is not possible to add an integer to a real; the integer must be cast by the user to a real so that real addition can be inferred and used. With multi-parameter type classes, mixed-mode arithmetic can be defined by the definition of suitable instances of +. This is achieved by the following definitions:

https://doi.org/10.1017/S0956796801004233 Published online by Cambridge University Press

```
class Plus α β γ where (+) :: α → β → γ
instance Plus Int Int Int where ...
instance Plus Int Float Float where ...
instance Plus Float Int Float where ...
instance Plus Float Float Float where ...
```

Then for example we can have the expression:

```
3 + 4.5 + 4.8 + 4 :: Float
```

The GHC and HUGS implementations of Haskell support multi-parameter templates. With the overload resolution strategies implemented in these compilers, generalizing the Haskell single-parameter overload resolution strategy, the constraints in the above example cannot be resolved. Using the overload resolution strategies in these implementations, and implementing mixed-mode arithmetic with multi-parameter templates, the type of the above expression is computed as:

```
(Plus Int Float α, Plus α Float β, Plus β Int Float) ⇒ Float
```

At the top level, this leads to a compile-time type error due to unresolved overloading.

The Haskell language uses *matching overload resolution*. Consider for example:

```
class Eq α where (==) :: α → α → Bool
instance (Eq α, Eq β) ⇒ Eq(α,β) where (==) = ...
f x y = ((x,y) == (x,y))
```

`f` is given the type `Eq(α,β) ⇒ (α,β) → Bool`. Haskell overload resolution resolves this constraint with the instance for pairs, giving the type `(Eq α, Eq β) ⇒ (α,β) → Bool`. Overload resolution is matching in the sense that overload constraints are only matched against the available instance types; no free variables in the overload constraints are instantiated as a result of overload resolution.

It is important to emphasize that even if the user casts the final result type in a mixed-mode expression, matching resolution may not be able to resolve any constraints. For the above example, it is necessary in GHC and HUGS for the user to supply the result type of all intermediate expressions:

```
(((((3 + 4.5) :: Float) + 4.8) :: Float) + 4) :: Float
```

Alternative approaches to implementing mixed-mode arithmetic have been proposed. The Haskell language specification requires that numeric literals be overloaded, with overload resolution determining whether a numeric literal is, for example, an integer or a float. This approach does not generalize because (a) it does not do anything to help in the case where non-literal values are being used, and (b) it is unrealistic to expect the language to provide special syntax and special treatment for arbitrary literals of any type.

Another alternative solution is to add subtyping to the language, and make `Int` a subtype of `Float`. Then there is no need for mixed-mode arithmetic. This is one possible interpretation of how the GHC and HUGS compilers implement the Haskell specification. These compilers implicitly coerce literals between numeric types. Every numeric literal has an implicit coercion `fromInteger` of type `Int → α` applied to it. Overload resolution resolves such a coercion to the desired result

https://doi.org/10.1017/S0956796801004233 Published online by Cambridge University Press

type. Although a solution in this situation, it should be obvious that this is just an instance of a more general problem with multi-parameter type classes. Consider for example the definition of matrix multiplication (from the previous section).

```
instance (Times α β γ, Plus γ γ γ) ⇒
    Times (Matrix α) (Matrix β) (Matrix γ) where ...
```

Then multiplying a matrix of integers by itself introduces the constraint `Times (Matrix Int) (Matrix Int) α` that is not resolved by matching overload resolution. In this case, the solution of declaring matrices a subtype of vectors does not work so well: coercing a matrix to a vector may be much more expensive than converting an integer to floating point.

As another example, consider the definition of a record type from the previous section, and the definition of the name field selector:

```
data Employee = EMPLOYEE String String Int
class Name α β where name :: α → β
instance Name Employee String where name(EMPLOYEE x y z) = x
```

Then define the following variables:

```
joeInfo = EMPLOYEE "Joe" "New York" 35
janeInfo = EMPLOYEE "Jane" "Paris" 24
```

Type-checking the expression

```
name joeInfo == name janeInfo
```

gives rise to the overload constraints and inferred type:

```
(Eq α, Name Employee α) ⇒ Bool
```

The latter constraint comes from the use of the `name` overloaded operation, applied to an argument of type `Employee` and returning a result of undetermined type α. Although the only instance of `name` defined for `Employee` arguments returns a result of type `String`, there is no contextual information in the above expression to reveal that the result has type `String`. The second constraint comes from the application of equality to the results of the applications of `name`.

The problem is that this expression leads to a compile-time error due to ambiguity. As with the earlier example, the only solution to this problem is to give type annotations for the intermediate expressions:

```
((name joeInfo) :: String) == (name janeInfo)
```

It should be noted that there are alternative ways of representing records using single-parameter type classes, that do not have this ambiguity problem. We repeat this example because it is cited by Peyton-Jones *et al.* (1997), a popular reference on multi-parameter type classes, as an illustration of ambiguity with multi-parameter type classes.

Finally, the example of a collection class is also cited by Haskell language designers as an example of ambiguity due to multi-parameter type classes (Jones, 2000). For example, the expression `(empty::[Int])` gives rise to the ambiguous constraint `(Collects [Int] α)`. Even if the `empty` operator is removed from the class, there are still problems:

https://doi.org/10.1017/S0956796801004233 Published online by Cambridge University Press

```
f x y = insert x . insert y
g     = f True 'a'
```

This gives g the type

$$\texttt{(Collects } \alpha \texttt{ Bool, Collects } \alpha \texttt{ Char)} \Rightarrow \alpha \rightarrow \alpha$$

The problem is that g incorrectly attempts to insert a boolean and a character into the same collection, but this type error is masked by unresolved overloading constraints.

These problems can be solved by using type constructor classes, instead of multi-parameter type classes (Jones, 1993). However this is at the cost that we can only define the first instance for the collections class given in the previous section. Another alternative approach is to use parametric type classes (Chen *et al.,* 1992). As explained at the end of this section, this example is really an example of a single-parameter parametric type class, which not surprisingly can be expressed using multi-parameter type classes.

Haskell incorporates a built-in default resolution mechanism (to overcome problems with the overloading of literals), and it might be considered that this approach can be generalized to solve the problems with matching resolution. Such a strategy could consist of ordering the available instances, and unifying an overload constraint with the instance types in that order until a unifying instance was found. For single-parameter type classes, there are already problems if programmer-defined defaults are allowed. Suppose for example `Plus` and `Times` are defined as single-parameter type classes, with a programmer-defined default instance of type `Int` for `Plus` and of type `Float` for `Times`. Then the constraint set

$$\texttt{Plus } \alpha\texttt{, Times } \alpha$$

can either be resolved by instantiating α to `Int` (resolving the first constraint against the default) or by instantiating α to `Float` (resolving the second constraint against the default). The resolution of the overloaded operations depends on the order of execution of the type-checker, so in effect the type-checker is the true specification of the type system.

For multi-parameter type classes the situation is even worse: even if there is only a single type class, with a single default for that type class, the resolution of constraints may depend upon the order in which the constraints are considered. For example:

```
class Foo α β where foo :: α → β
instance Foo Int Char
instance Foo Char Int
```

with the constraints

$$\texttt{Foo } \alpha\ \beta\texttt{, Foo } \beta\ \alpha.$$

If the first instance is selected as the default for resolving the first constraint, β is instantiated to `Char`, so the second constraint is resolved against the second instance. Suppose, on the other hand, the second constraint is resolved against the default instance, then the first constraint is resolved against the second instance.

https://doi.org/10.1017/S0956796801004233 Published online by Cambridge University Press

In this paper, we propose an overload resolution strategy which we refer to as *domain-driven unifying overload resolution.* This resolution strategy addresses the issues discussed in this section regarding overload resolution and multi-parameter type classes. Recalling the type with unresolved constraints from the start of this section:

```
(Plus Int Float α, Plus α Float β, Plus β Int Float) ⇒ Float
```

With unifying overload resolution, it can be recognized that there is a single instance satisfying the first type constraint, with type `Plus Int Float Float`. Overload resolution therefore instantiates α to `Float`. The second type constraint is now `Plus Float Float` β, for which again there is only a single satisfying instance. Therefore, overload resolution instantiates β to `Float`. Overload resolution is unifying in the sense that overload constraints are not *matched* against instance types, they are *unified* with instance types, and in the result free type variables in the overload constraints can be instantiated.

The problematic expression `(empty::[Int])` for the collections class leads to the constraint `(Collects [Int] α)`. If the overlapping restriction is used to specify that the first argument to the type predicate specifies the second, then this predicate can be resolved against the first instance type for `Collects`, instantiating α to `Int`. The problematic definition g, which hides a type error with the type `(Collects α Bool, Collects α Char) ⇒ α → α`, is avoided again due to the overlapping restriction. Since both type predicates have the same first type argument, and the first type argument determines the second type argument, the overload resolution algorithm attempts to unify these type predicates, leading to a unification failure.

In our type system, the instance declarations for the `Collects` class must be modified:

```
instance Eq α ⇒ Collects [α] α where ...
instance (Eq α, Boolean β) ⇒ Collects (α → β) α where ...
instance Collects BitSet Char where ...
instance (Hashable α, Collects α β, Integer γ) ⇒
         Collects (Array γ β) α where ...
```

In the second and fourth instance types, `Boolean` and `Integer` are classes that play the rôle of characteristic predicates in the type system for the corresponding types:

```
class Boolean α where toBool :: α → Bool  fromBool :: Bool → α
instance Boolean Bool where toBool x = x  fromBool y = y
class Integer α where ...
instance Integer Int where ...
```

This treatment is necessary because, unlike GHC and HUGS, we only allow type arguments of depth zero or one in the arguments to the type predicate in an instance declaration. The reason for this is given in section 6.

There is the apparent danger here of ambiguous types, for example, the expression `(toBool (fromBool true))` has the apparently ambiguous type `(Boolean α) ⇒ Bool`. With the overlapping restriction, it is possible to specify that the first zero arguments to a type predicate determine all of the other type arguments (as

https://doi.org/10.1017/S0956796801004233 Published online by Cambridge University Press

explained more fully in section 7). This means in effect that there can only be one instance declaration for such a class, and any type constraint for that class can be resolved against that single instance. This 'trick' can be applied to the `Integer` and `Boolean` type classes above to ensure that they do not give rise to ambiguous types.

We can perform this modification one more time, on the third instance type:

```
class Character α where ...
instance Character Char where ...
instance Character α ⇒ Collects BitSet α where ...
```

At this point we have four instance types for `Collects`, indexed by the first type argument and parametric in the second type argument. These all constitute instances of a parametric type class (Chen *et al.,* 1992), where parametric type classes in turn are extended single-parameter type classes. We refer to parametric type classes as single-parameter because the instances are indexed by a single type.

It is no surprise that parametric type classes form a subset of multi-parameter type classes, and unifying overload resolution can be seen as a generalization of the resolution strategy for parametric type classes. Parametric type class resolution has a step similar to Step (2) of unifying overload resolution (Definition 4.1), that catches type errors earlier by combining type class constraints; the type of `g` above is an example of this. On the other hand, overload resolution for parametric type classes uses matching overload resolution to match the index type argument of a parametric type predicate against a 'set of types' expression, parameterized by the element types, describing the set of instances available for that type predicate (Chen *et al.,* 1992). The element type parameters are instantiated as part of this matching process. A matching step decomposes a collection type, constrained by a type class, to a collection of constrained element types. The constraints for an element type include both type class constraints and constraints relating it to type parameters; in the latter case, the element type is unified with a type parameter that constrains it. In contrast, unifying overload resolution unifies all of the type arguments of a type predicate against the corresponding type arguments in a resolvable instance type, once such an instance has been uniquely identified. There are several subtleties in the sufficiency condition for identifying such an instance.

For the special single-parameter case of parametric type classes, unifying overload resolution is known to terminate. As mentioned in section 1, for the more interesting case where an instance may be indexed by several types, it is easy to show that unifying resolution does not terminate. This is so even for the case of a binary type class `(C α β)` where the first type argument α determines the second type argument β, i.e. for a multi-parameter type class that superficially resembles a parametric type class. An example is provided in section 4.

3 Type system and type checking

3.1 Syntax of types

Types in our type system are described by the following abstract syntax:

https://doi.org/10.1017/S0956796801004233 Published online by Cambridge University Press

Mono Types	τ	::=	$\alpha \mid \mathtt{t}(\tau_1, \ldots, \tau_n)$
Instance Type	ρ	::=	$\forall\overline{\alpha_n}.\{\kappa_1, \ldots, \kappa_m\} \Rightarrow c(\tau_1, \ldots, \tau_k)$
Context Constraint	κ	::=	$c(\tau_1, \ldots, \tau_k) \mid \alpha \neq \mathtt{t}(\alpha_1, \ldots, \alpha_n)$
Constraint Set	C	::=	$\{\} \mid \{c(\tau_1, \ldots, \tau_k)\} \mid C_1 \cup C_2$
Poly Types	σ	::=	$\forall\overline{\alpha_n}.C \Rightarrow \tau$

The type expressions $\mathtt{t}(\tau_1, \ldots, \tau_n)$ include the arrow type $(\tau_1 \to \tau_2)$, so $\to$ is an infix type constructor. We assume for simplicity that each class has exactly one overloaded operator, and an overloaded operator is defined in exactly one class. We assume given an environment A_0 of class definitions of the form $(c ::_o^k (x : \forall\overline{\alpha_m}.\tau))$ and class instance types ρ, one of the former for each overloaded symbol. In a class definition, c is the name of the class, with single operator x, and $k \leqslant m$ as explained in the next subsection.

There is a subtle but important distinction between instance types and polytypes: the context for the former includes both class constraints $c(\overline{\tau})$ and negative constraints $\alpha \neq \mathtt{t}(\overline{\tau})$, whereas the context of a polytype only includes class constraints. Negative constraints are only used to distinguish overlapping instance types, and a class constraint is not resolved against an instance type (during type-checking) unless the negative constraints for that instance type are satisfied by the substitution unifying the instance type with the class constraint. Therefore, negative constraints never appear in polymorphic types.

$TV(\ldots)$ denotes the free type variables of an expression (monotype, instance type, context constraint, constraint set, polytype):

$$
\begin{aligned}
TV(\alpha) &= \{\alpha\} \\
TV(\mathtt{t}(\tau_1, \ldots, \tau_n)) &= TV(\tau_1) \cup \cdots \cup TV(\tau_n) \\
TV(\forall\overline{\alpha_n}.C \Rightarrow \tau) &= TV(C) \cup TV(\tau) - \{\overline{\alpha_n}\} \\
TV(\{\}) &= \{\} \\
TV(C_1 \cup C_2) &= TV(C_1) \cup TV(C_2) \\
TV(\{c(\tau_1, \ldots, \tau_k)\}) &= TV(\tau_1) \cup \cdots \cup TV(\tau_k) \\
TV(\forall\overline{\alpha_n}.\{\overline{\kappa_m}\} \Rightarrow c(\overline{\tau_k})) &= \bigcup\{\overline{TV(\kappa_m)}\} \cup \bigcup\{\overline{TV(\tau_k)}\} - \{\overline{\alpha_n}\} \\
TV(c(\tau_1, \ldots, \tau_k)) &= TV(\tau_1) \cup \cdots \cup TV(\tau_k) \\
TV(\alpha \neq \mathtt{t}(\alpha_1, \ldots, \alpha_n)) &= \{\alpha\}
\end{aligned}
$$

The following restriction on instance types is a generalization of the Haskell restrictions (for single-parameter type classes) that allows the examples provided in section 1.

Definition 3.1 (Instance Restriction)
An instance definition must satisfy the following: the instance type has the form

$$(c_1(\overline{\beta^1}), c_2(\overline{\beta^2}), \ldots, c_k(\overline{\beta^k}), \gamma_1 \neq \mathtt{t}_1(\overline{\delta^1}), \ldots, \gamma_m \neq \mathtt{t}_m(\overline{\delta^m})) \Rightarrow c(\overline{\tau_n})$$

where:

1. $\{\overline{\beta^1}\} \cup \{\overline{\beta^2}\} \cup \cdots \cup \{\overline{\beta^k}\} \cup \{\gamma_1, \ldots, \gamma_m\} \subseteq TV(c(\overline{\tau_n}))$.
2. For each $i = 1, \ldots, n$, τ_i is either of the form $\mathtt{t}(\overline{\alpha})$ or α.

https://doi.org/10.1017/S0956796801004233 Published online by Cambridge University Press

The motivation for negative context constraints $\alpha \neq \mathtt{t}(\beta_1, \ldots, \beta_k)$ is given in section 1, and repeated in the next subsection.

A type environment is a pair (A, C), where $A = A_0 \cup A'$ and A' is a set of pairs of the form $(x : \sigma)$, corresponding to types for non-overloaded symbols. C is a set of constraints $c(\bar{\tau})$ on implicit operator parameters in polymorphic types, and these constraints are discharged in forming polymorphic types.

The type rules for the mini-language are provided in Fig. 1. We do not consider subclass hierarchies in this type system; they do not materially affect the results.

3.2 Overlapping restriction

At the heart of our overload resolution strategy is the *overlapping restriction* for instance types. We provide two versions of this restriction, the strong and weak overlapping restrictions. The strong restriction is simple to motivate based on the examples in section 2. The weak restriction is more complicated but not as restrictive.

For the *strong overlapping restriction*, we assume that every overloaded operator type is of the form $\tau_1 \to \cdots \to \tau_n \to \tau_0$, where τ_0 is not a function type. We require that for any instance type for an overloaded operation, the domain of the instance type (defined by the types $\tau_1, \ldots, \tau_n$) completely determines the codomain τ_0. For example, suppose we allowed instances of + with types `Int` $\to$ `Float` $\to$ `Int`, `Int` $\to$ `Float` $\to$ `Float` and `Float` $\to$ `Float` $\to$ `Float`. Then in the expression `(1+2.5)+3.1`, there would be two possible resolutions for the first use of +. We avoid this incoherence by disallowing both of the definitions with types `Int` $\to$ `Float` $\to$ `Int` and `Int` $\to$ `Float` $\to$ `Float`. Only one of these can be defined.

On the other hand, we do want to allow overlapping instances of the form:

```
instance (Times α β γ, Plus γ γ γ) ⇒
    Times (Matrix α) (Matrix β) (Matrix γ) where ...
instance Times α β γ ⇒ Times α (Matrix β) (Matrix γ) where ...
```

In this case, the second instance is a 'default' instance that is chosen if the first is not applicable. To incorporate this into our type system, we add *negative context constraints*, of the form $\alpha \neq \mathtt{t}(\alpha_1, \ldots, \alpha_n)$. The above instance types are then provided as:

```
instance (Times α β γ, Plus γ γ γ) ⇒
    Times (Matrix α) (Matrix β) (Matrix γ) where ...
instance (Times α β γ, α ≠ Matrix δ) ⇒
    Times α (Matrix β) (Matrix γ) where ...
```

A substitution is a finite function from type variables to types, homomorphically extended to a function from types to types. Say that a substitution θ satisfies a negative context constraint $\alpha \neq \mathtt{t}(\alpha_1, \ldots, \alpha_n)$ if $\theta(\alpha) \neq \mathtt{t}(\tau_1, \ldots, \tau_n)$ for any $\tau_1, \ldots, \tau_n$. Note that this is *not* $\theta(\alpha) \neq \mathtt{t}(\theta(\alpha_1), \ldots, \theta(\alpha_n))$; this latter condition is too difficult to check for in overload resolution.

The overlapping restriction uses the negative context constraints to allow overlapping instance types provided it does not lead to an ambiguous type system.

https://doi.org/10.1017/S0956796801004233 Published online by Cambridge University Press

Definition 3.2 (Strong Overlapping Restriction)
We assume that every multi-parameter class definition is of the form $(c :: {}^{k}_{o}(x : \forall\overline{\alpha_{k+1}}.\alpha_1 \to \cdots \to \alpha_k \to \alpha_{k+1}))$. For every pair of distinct instance types $(\forall\overline{\alpha}.\ldots \Rightarrow c(\overline{\tau_{k+1}})), (\forall\overline{\beta}.\ldots \Rightarrow c(\overline{\tau'_{k+1}})) \in A_0$, it must be the case that there is no substitution θ satisfying the negative context constraints for the two instance types, such that $\theta(\tau_i) = \theta(\tau'_i)$ for all $i = 1,\ldots,k$.

The definition of a satisfying substitution for an overload context is provided in the next subsection.

Since the strong overlapping restriction is sometimes too strong, we weaken this to a more complicated but weaker restriction.

Definition 3.3 (Weak Overlapping Restriction)
We assume that every multi-parameter class definition is of the form $(c ::^{k}_{o} (x : \forall\overline{\alpha_m}.\tau$, where $FV(\tau) \subseteq \{\overline{\alpha_m}\}$ and $k \leqslant m$. For any instance type $\forall\overline{\alpha}.\ldots \Rightarrow c(\overline{\tau_m})$, we refer to $\tau_1,\ldots,\tau_k$ as the *domain types* of the instance type (by analogy with the strong overlapping restriction). For every pair of distinct instance types $(\forall\overline{\alpha}.\ldots \Rightarrow c(\overline{\tau_m})), (\forall\overline{\beta}.\ldots \Rightarrow c(\overline{\tau'_m})) \in A_0$, it must be the case that there is no substitution θ satisfying the negative context constraints for the two instance types, such that $\theta(\tau_i) = \theta(\tau'_i)$ for all $i = 1,\ldots,k$.

The strong overlapping restriction requires that every multi-parameter type class be of the form:

`class C` α_1 ... α_k α_{k+1} `where f ::` $\alpha_1 \to \cdots \to \alpha_k \to \alpha_{k+1}$

The weak overlapping restriction generalizes the allowable multi-parameter type class declarations to have the form:

`class C` α_1 ... α_k α_{k+1} ... α_m `where f ::` τ

Then there cannot be distinct instances for the `C` class which have similar instantiations for $\alpha_1,\ldots,\alpha_k$; the first k type parameters (the domain types) completely determine the instance (and therefore the range types of the instance).

The strong overlapping restriction rules out some useful operator signatures, for example, `fromInteger :: Num` $\alpha \Rightarrow$ `Int` $\to \alpha$ and `read :: Read` $\alpha \Rightarrow$ `String` $\to \alpha$. The weak overlapping restriction allows these operator signatures, for example $(\texttt{Num} ::^{1}_{o} (\texttt{fromInteger} : \forall\alpha.\texttt{Int} \to \alpha))$. Practically this means that no unifying resolution is possible for these operators. This point is discussed further in section 7.

For obvious reasons of economy, we use a mini-language with simple multi-parameter type classes. There is no obstacle to generalizing these restrictions to full Haskell with type constructor classes. A negative context constraint in such a type system has the form $(\alpha \neq (\texttt{t}\ \beta_1\ \ldots\ \beta_k))$, where α and $(\texttt{t}\ \beta_1\ \ldots\ \beta_k)$ are type expressions with the same kind. It is again important to note that negative context constraints only place a negative condition on the outermost type constructor of any instantiation of α. There are no conceptual problems with allowing negative context constraints of the form $(\alpha \neq (\beta\ \beta_1\ \ldots\ \beta_k))$, with a variable at the head of the right-hand side. However, such an extension is of questionable value. Consider for example attempting to define a generic instance for monad transformers:

https://doi.org/10.1017/S0956796801004233 Published online by Cambridge University Press

```
instance (Monad β, Monad (γ β), γ ≠ (δ α)) ⇒ MonadT γ β where ...
instance (Monad β, Monad (StateT α β)) ⇒ MonadT (StateT α) β where ...
```

A more plausible definition lists the concrete classes that are to be chosen over the default instance:

```
instance (Monad β, Monad (γ β), γ ≠ (StateT α)) ⇒ MonadT γ β where ...
instance (Monad β, Monad (StateT α β)) ⇒ MonadT (StateT α) β where ...
```

The definition of ambiguous typing is not affected by the addition of negative context constraints, since inequality constraints do not appear in polytypes. For instance types $\forall\overline{\alpha_n}.\{\kappa_1,\ldots,\kappa_m\} \Rightarrow c(\overline{\tau_k})$, we require that $\bigcup\{\overline{TV(\kappa_m)}\} \subseteq \bigcup\{\overline{TV(\tau_n)}\}$.

3.3 Instance relation, matching condition and satisfying substitution

Define $A, C \vdash (\forall\overline{\alpha}.\{\overline{c_m(\overline{\tau_m})}, \overline{\gamma_n \neq \tau_n''}\} \Rightarrow \tau) \longrightarrow (\forall\overline{\beta}.\{\overline{c_p'(\overline{\tau_p'})}\} \Rightarrow \tau')$ to be true if there is some θ with domain $\{\overline{\alpha}\}$ such that:

1. $\tau' = \theta(\tau)$.
2. θ satisfies the negative context constraints $\gamma_1 \neq \tau_1'', \ldots, \gamma_n \neq \tau_n''$.
3. Let $C' = C \cup \{c_k'(\overline{\tau_k'}) \mid k = 1,\ldots,p\}$. Then $A, C' \vdash \rho_j \longrightarrow c_j(\overline{\tau_j})$ for some $\rho_j \in A \cup C'$, for $j = 1,\ldots,m$.
4. $\{\overline{\beta}\} \cap TV(A \cup C) = \{\}$.

Condition (1) states that the body of the second instance type is obtained by instantiating the body of the first instance type. Condition (2) states that the negative context constraints are satisfied. Condition (3) states that the constraints in the first polymorphic type can be derived from the constraints in the second polymorphic type; if the latter are satisfiable, then so are the former. Finally, Condition (4) states that the type variables bound in the second polymorphic type are not free in the environment. The relation $A, C \vdash \sigma \longrightarrow \sigma'$ requires the type environment (A, C) because the statement of the relation needs to make assertions about the derivability of type constraints in polymorphic types. The relation $A_0, C \vdash \rho \longrightarrow c(\overline{\tau})$ is defined in an analogous fashion: define $A, C \vdash (\forall\overline{\alpha}.\{\overline{c_m(\overline{\tau_m'})}, \overline{\gamma_n \neq \tau_n''}\} \Rightarrow c(\tau_1,\ldots,\tau_k)) \longrightarrow c(\tau_1''',\ldots,\tau_k''')$ to be true if there is some θ with domain $\{\overline{\alpha}\}$ such that:

1. $\tau_i''' = \theta(\tau_i)$ for $i = 1,\ldots,k$.
2. θ satisfies the negative context constraints $\gamma_1 \neq \tau_1'', \ldots, \gamma_n \neq \tau_n''$.
3. $A, C \vdash \rho_j \longrightarrow c_j(\overline{\tau_j'})$ for some $\rho_j \in A \cup C$, for $j = 1,\ldots,m$.

The justification for local overload resolution during type inference is given by the following (used in the LET rule in the type rules).

Definition 3.4 (Matching Condition)
A constraint set C satisfies the matching condition provided:

1. For any $c(\overline{\tau}) \in C$, there is some $(\forall\overline{\alpha}.\ldots \Rightarrow c(\overline{\tau'})) \in A_0$ such that $c(\overline{\tau})$ and $c(\overline{\tau'})$ are unifiable, where the unifying substitution satisfies any negative context constraints in the instance type.

https://doi.org/10.1017/S0956796801004233 Published online by Cambridge University Press

$$\frac{(c ::^{k}_{o} (x : \forall\overline{\alpha}.\tau)) \in A \quad \rho \in A \cup C \quad A_0, C \vdash \rho \longrightarrow c(\overline{\tau'})}{A; C \vdash x : \{\overline{\tau'}/\overline{\alpha}\}\tau} \quad \text{(OVar)}$$

$$\frac{A, C \vdash \sigma \longrightarrow \tau \text{ where } (x : \sigma) \in A}{A; C \vdash x : \tau} \quad \text{(Var)}$$

$$\frac{A \cup \{x : \tau_1\}; C \vdash e : \tau_2}{A; C \vdash (\lambda x.e) : \tau_1 \to \tau_2} \quad \text{(Abs)}$$

$$\frac{A; C \vdash e_1 : \tau_2 \to \tau_1 \quad A; C \vdash e_2 : \tau_2}{A; C \vdash (e_1\ e_2) : \tau_1} \quad \text{(App)}$$

$$\frac{\begin{array}{c} A; C_1 \vdash e_1 : \tau_1 \quad \{\overline{\alpha_m}\} = (TV(\tau_1) \cup TV(C_1)) - TV(A) \\ C_1 \text{ satisfies the matching condition} \\ (A \cup \{x : (\forall\overline{\alpha_m}.C_1 \Rightarrow \tau_1)\}); C_2 \vdash e_2 : \tau_2 \end{array}}{A; C_2 \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau'} \quad \text{(Let)}$$

Fig. 1. Type rules.

2. For any $(c ::^{k}_{0} (x : \sigma)) \in A_0$, $c(\overline{\tau_m}) \in C$, there is some $i \in \{1, \ldots, k\}$ such that τ_i is a variable (i.e. it is not the case that all of the domain types are non-variable).
3. Finally, given $(c ::^{k}_{0} (x : \sigma)) \in A_0$, there do not exist distinct $c(\overline{\tau_m}), c(\overline{\tau'_m}) \in C$ such that $\tau_i = \tau'_i$ for $i = 1, \ldots, k$.

The first condition states that every constraint at least unifies with the head of some instance type. The second condition states that there is no unresolved constraint where the outer type constructors of the domain types are determined; this condition forces further resolution of such a constraint. Without this condition, a resolution algorithm that did nothing would be correct according to the type system. But the whole point of unifying resolution is that leaving the constraints unresolved can lead to practical problems. The third condition states that if there are two constraints with the same domain types, then these constraints should be combined (their range types unified).

Define that θ is a *satisfying substitution* for C, denoted $\theta \models C$, if for all $c(\overline{\tau}) \in \theta(C)$, there is some $\rho \in A_0$ such that $A_0, \{\} \vdash \rho \longrightarrow c(\overline{\tau})$. Define $\Theta(C) = \{\theta \mid \theta \models C\}$. Define $(A, C) \longrightarrow (A', C')$ to be:

1. $\Theta(C') \subseteq \Theta(C)$.
2. For all $x, \tau, \theta \in \Theta(C')$, if $\theta(A'); \{\} \vdash x : \tau$ then $\theta(A); \{\} \vdash x : \tau$.

To understand this, consider $A_0 = \{(c ::^{1}_{o} (x : \forall\alpha.\alpha)), c(\texttt{Int})\}$, $A = A' = A_0 \cup \{(y : \beta)\}$, $C = \{\}$ and $C' = \{c(\beta)\}$. (A', C') is an instance of (A, C) since the type of y is constrained to be `Int` in (A', C'). As another example, consider A_0, C, C' as before, and $A = A_0 \cup \{(y : \texttt{Int})\}$, $A' = A_0 \cup \{(y : \beta)\}$. The type of y in A' is an instance of y in A because of the constraint on β in C'. Define $(A, C) \longleftrightarrow (A', C')$ to be: $(A, C) \longrightarrow (A', C')$ and $(A', C') \longrightarrow (A, C)$.

A semantics for the language can be provided in terms of a translation where contexts are realized as 'type dictionaries'. This semantics can be used to verify the soundness of the type system. Since similar semantics have already been presented

https://doi.org/10.1017/S0956796801004233 Published online by Cambridge University Press

$$\frac{(c ::_o^k (x : \forall \overline{\alpha}.\tau)) \in A \quad \theta = \{\overline{\beta}/\overline{\alpha}\} \quad \text{each } \beta_i \text{ is new}}{A \vdash x \Longrightarrow (\theta, \theta(\tau), \{c(\overline{\beta})\})} \quad (\textsc{OVar})$$

$$\frac{A(x) = \forall \overline{\alpha}.C \Rightarrow \tau \quad \theta = \{\overline{\beta}/\overline{\alpha}\} \quad \text{each } \beta_i \text{ is new}}{A \vdash x \Longrightarrow (\theta, \theta(\tau), \theta(C))} \quad (\textsc{Var})$$

$$\frac{A \cup \{(x : \alpha)\} \vdash e \Longrightarrow (\theta, \tau', C) \quad \alpha \text{ new}}{A \vdash (\lambda x.e) \Longrightarrow (\theta, \theta(\alpha) \to \tau', C)} \quad (\textsc{Abs})$$

$$\frac{\begin{array}{c} A \vdash e_1 \Longrightarrow (\theta_1, \tau_1, C_1) \quad \theta_1(A) \vdash e_2 \Longrightarrow (\theta_2, \tau_2, C_2) \\ \theta = \mathit{UNIFY}(\theta_2(\tau_1), \tau_2 \to \alpha) \quad \alpha \text{ new} \end{array}}{A \vdash (e_1\ e_2) \Longrightarrow (\theta \circ \theta_2 \circ \theta_1, \theta(\tau_2), \theta(\theta_2(C_1) \cup C_2))} \quad (\textsc{App})$$

$$\frac{\begin{array}{c} A \vdash e_1 \Longrightarrow (\theta_1, \tau_1, C_1) \quad C_1 \downarrow = (\theta, C_1') \\ \{\overline{\alpha}\} = (\mathit{TV}(C_1') \cup \mathit{TV}(\theta(\tau_1))) - \mathit{TV}(\theta(\theta_1(A))) \\ \theta(\theta_1(A)) \cup \{(x : \forall \overline{\alpha}.C_1' \Rightarrow \theta(\tau_1))\} \vdash e_2 \Longrightarrow (\theta_2, \tau, C_2) \end{array}}{A \vdash (\texttt{let}\ x = e_1\ \texttt{in}\ e_2) \Longrightarrow (\theta_2 \circ \theta \circ \theta_1, \tau_2, C_2)} \quad (\textsc{Let})$$

Fig. 2. Type-checking algorithm.

in the literature, we do not provide any further details. See Hall *et al.* (1996) for a representative example. Although the type rules do not enforce satisfiability, the type system is still sound because type constraints are converted to implicit operator parameters during compilation, suspending the evaluation of the expression with the unresolvable use of overloaded operations. This is similar to the situation with the Haskell type rules, which only enforce a weak form of satisfiability.

Figure 2 provides the type-checking algorithm. The algorithm takes as inputs a type environment A and a program e. The outputs of the algorithm consist of a substitution θ and a type τ. The algorithm also computes a constraint set C, that is discharged in the types of polymorphic definitions.

4 Domain-driven unifying overload resolution

In this section we describe the overload resolution algorithm used in the type inference algorithm in the previous section, and then consider the correctness of type inference.

We assume a function $\mathit{UNIFY}(\tau, \tau')$ that computes the most general unifying substitution of the types τ and τ'. We also assume a function $\mathit{UNIFY\_SET}$ defined by:

$$\begin{array}{rcl} \mathit{UNIFY\_SET}(\{\}) & = & \{\} \\ \mathit{UNIFY\_SET}(\{(\tau, \tau')\} \uplus S) & = & \mathit{UNIFY\_SET}(\theta(S)) \circ \theta, \text{ where } \theta = \mathit{UNIFY}(\tau, \tau') \end{array}$$

where $\uplus$ denotes disjoint union.

Definition 4.1 (Domain-driven unifying overload resolution)
Given an initial set of constraints C, let $C \downarrow = (\theta^r, C^r)$ denote the result of repeated application of the following resolution step to the initial configuration $(\{\}, C)$.

Given a pair (θ, C) where θ is a substitution, C a set of constraints. Each step of the resolution algorithm consists of performing one of the following actions:

https://doi.org/10.1017/S0956796801004233 Published online by Cambridge University Press

1. Let A_0 contain the class declaration $(c ::_o^l (x : \forall\overline{\beta_k}.\tau))$ where $l \leqslant k$. Let C contain a constraint of the form $c(\overline{\tau_k})$, where each τ_i is of the form $\mathtt{t}_i(\ldots)$ for some $\mathtt{t}_i$, $i = 1,\ldots,l$. Let A_0 contain an instance type:

$$(c ::_i \forall\overline{\alpha}.\{c_1(\overline{\tau^1}),\ldots,c_m(\overline{\tau^m}),\beta_1 \neq \tau_1'',\ldots,\beta_n \neq \tau_n''\} \Rightarrow c(\overline{\tau_k'}))$$

   Suppose that there is a unifying substitution θ' such that $\theta'(\tau_i) = \theta'(\tau_i')$ for $i = 1,\ldots,l$, satisfying the negative context constraints $\beta_1 \neq \tau_1'',\ldots,\beta_n \neq \tau_n''$, where $\{\overline{\alpha}\}$ are renamed to be new type variables. Let $\theta'' = UNIFY(c(\overline{\tau_j}), c(\overline{\tau_j'}))$. Let C' result from removing the constraint $c(\overline{\tau_k})$ from C, and adding the constraints $c_1(\overline{\theta''(\tau^1)}),\ldots,c_m(\overline{\theta''(\tau^m)})$. Then the algorithm transitions to $(\theta'' \circ \theta, \theta''(C'))$.
2. Let A_0 contain the instance declaration $(c ::_o^l \sigma)$. Let C contain constraints of the form $c(\overline{\tau_k})$ and $c(\overline{\tau_k'})$, such that $\tau_i = \tau_i'$ for $i = 1,\ldots,l$. Let $\theta' = UNIFY\_SET(\{(\tau_j,\tau_j') \mid j = l+1,\ldots,k\})$. Let C' result from removing the constraint $c(\overline{\tau_k})$ from C. Then the algorithm transitions to $(\theta' \circ \theta, \theta'(C'))$.
3. The overload resolution algorithm fails if there is a constraint $c(\overline{\tau}) \in C$ for which there is no overload instance type that unifies with $c(\overline{\tau})$.

To understand the first transition rule, consider:

```
data Employee = EMPLOYEE String String Int
class Name α β where name :: α → β
instance Name Employee String where name (EMPLOYEE x y z) = x
name (EMPLOYEE "Jane" ...) == name (EMPLOYEE "Joe" ...)
```

As already noted in section 2, this gives rise to the overload constraints

```
Eq α, Name Employee α.
```

The first part of the match condition requires that the second constraint be resolved against the single unifying instance, instantiating α with `String`. The first constraint is then `Eq String`, which again can be resolved.

The first transition rule checks that the domain types unify, rather than only checking for matching. The following example illustrates why this is so:

```
class Foo α β where foo :: α → β → ()
instance Foo [α] [α] where foo xs ys = ()
f x y = foo [x] [y]
```

The first transition of overload resolution recognizes that the type constraint `Foo [α] [β]` can be resolved with the instance for lists, equating the types of `x` and `y`, so `f` has the type $\forall\alpha.\alpha \to \alpha \to ()$.

To understand the second transition rule, consider:

```
f x y = (x+y, x+y)
```

Two constraints are generated: `Plus α β γ` and `Plus α β δ`. The second transition rule combines these into one constraint.

Lemma 4.1 (Correctness of Overload Resolution)
If $C \downarrow = (\theta', C')$, then for all $\theta \in \Theta(C)$, there exists $\theta'' \in \Theta(C')$ such that $\theta = \theta'' \circ \theta'$.

https://doi.org/10.1017/S0956796801004233 Published online by Cambridge University Press

Proof
Equivalence of the constraint sets is verified by induction on the transitions of the overload resolution algorithm, using the overlapping restriction to reason that each resolution step does not remove any satisfying instantiations for the overload constraints. □

Using this result, we verify the following by induction on the execution of the type inference algorithm.

Theorem 4.1 (Soundness of Type Inference)
Suppose $A \vdash e \Longrightarrow (\theta, \tau, C)$. Then $\theta(A); C \vdash e : \tau$.

The overload resolution algorithm is sound, since it preserves the set of satisfying substitutions. However overload resolution may not terminate. We have not considered the restrictions that would need to be placed on overload instance types in order to ensure termination. In fact it is difficult to see what kind of restrictions could be imposed to ensure termination, while retaining a useful type system. For example, the following restrictions are too strong, and disallow the examples in section 1:

Primitive Recursion Instance Restriction: Any overload instance type must have the form:

$$\forall\overline{\alpha^1} \cdots \forall\overline{\alpha^k}.\{c(\alpha_i^1, \ldots, \alpha_i^k) \mid i = 1, \ldots, n\} \Rightarrow c(\mathtt{t}_1(\overline{\alpha^1}), \ldots, \mathtt{t}_k(\overline{\alpha^k}))$$

Matching resolution can easily be shown to terminate with this restriction. However, unifying resolution may fail to terminate if the constraints are not satisfiable. Consider, for example:

```
class Foo α β where foo :: α → β → Int
instance Foo Int Float where foo x y = 0
instance Foo α β ⇒ Foo [α] [β] where foo (x:_) (y:_) = foo x y
g x y = (foo [x] y) + (foo [y] x)
```

The allowable instance types for `foo` are

`Int → Float → Int`, `[Int] → [Float] → Int`, `[[Int]] → [[Float]] → Int`, ...

Unifying resolution fails to terminate with the constraints `(Foo [α] β)`, `(Foo [β] α)` resulting from type-checking `g`.

Nevertheless, we have the following result:

Lemma 4.2
Given C and $\theta \models C$. Then the unifying resolution algorithm is guaranteed to terminate.

Proof
For each constraint $c(\overline{\tau}) \in C$, the algorithm builds a partial derivation Π' for $A_0, \{\} \vdash \rho \longrightarrow c(\overline{\theta'(\tau)})$, for some instance type $\rho \in A_0$ and substitution θ'. By the correctness of overload resolution, any derivation Π'' for the constraint must have root judgement $A_0, \{\} \vdash \rho \longrightarrow c(\overline{\theta''(\theta'(\tau))})$ for some θ'', and the partial derivation Π' can be instantiated and completed to Π''. Since any path in $\theta''(\Pi')$ is a prefix of a path in Π'', and all paths in the latter tree are finite, the algorithm cannot loop infinitely. □

https://doi.org/10.1017/S0956796801004233 Published online by Cambridge University Press

Therefore, termination can be assured if we can verify that the constraints are satisfiable. In the next section we show that satisfiability is undecidable for any type system that supports the examples given in section 1.

Theorem 4.2 (Completeness of Type Inference)
Given A, e. Suppose there exists A', C', θ', τ' such that $(\theta'(A), \{\}) \longrightarrow (A', C')$ and $A'; C' \vdash e : \tau'$. Then we have $A \vdash e \Longrightarrow (\theta, \tau, C_1)$, for some θ, τ, C_1 and C_2 such that:

1. $\tau' = \theta''(\tau)$ and $\theta' = \theta'' \circ \theta$ for some θ'';
2. $(\theta'(A), \theta''(C_1)) \longrightarrow (A', C')$

Proof
By induction on the derivation for $A'; C' \vdash e : \tau'$. The proof is essentially the same as the standard completeness proof for ML-style type inference, using the correctness of overload resolution (Lemma 4.1). □

5 Satisfiability is undecidable

In this section we demonstrate via an example how to reduce the Post Correspondence Problem to the problem of checking the satisfiability of a set of multi-parameter overloading constraints. To demonstrate that the construction is applicable in more restricted type systems for multi-parameter type classes, we assume that instance definitions satisfy the following.

Definition 5.1 (Linear Instance Restriction)
An instance definition must satisfy the conditions of Definition 3.1, as well as the following restriction:

3. Any type variable occurring in $TV(c(\overline{\tau_n}))$ has a single occurrence.

The linearity restriction is not necessary for our algorithms. We only include it to demonstrate the wide application of this undecidability result. In fact two of the examples in section 1 (collection classes and monad transformers) fail to satisfy the linearity restriction.

Recall the statement of the PCP: given two sequences of strings $u_1, \ldots, u_m$ and $v_1, \ldots, v_m$, is there some sequence $i_1, \ldots, i_k$ such that $u_{i_1} \ldots u_{i_k} = v_{i_1} \ldots v_{i_k}$? Volpano & Smith, (1991) originally used a reduction to PCP to demonstrate the undecidability of satisfiability with *unrestricted single-parameter type classes*. Their construction uses only single-parameter type classes, but violates Instance Restriction (2) and also the linearity restriction above. Our approach is similar to their construction, but does not violate the Instance Restrictions; therefore our result demonstrates the undecidability of satisfiability with *restricted multi-parameter type classes*. To code PCP as a type-checking problem, we use a class as a predicate: `class` $\texttt{PCP}(\alpha_1, \cdots, \alpha_p, \beta_1, \cdots, \beta_p, \gamma)$ where p is $(\texttt{max}\ \{|\ u_i\ |, |\ v_j\ |\}_{i,j}) + 1$. Intuitively we have

$$\texttt{PCP}(1, 0, \varepsilon, u, 1, 0, 1, v, \gamma) \quad \text{if} \quad u_{i_1} = 10,\ v_{i_1} = 101,\ u_{i_2} \ldots u_{i_k} = u,\ v_{i_2} \ldots v_{i_k} = v$$

for some $u_{i_1}, u_{i_2}, \ldots, u_{i_k}, v_{i_1}, v_{i_2}, \ldots, v_{i_k}$, where ε denotes the empty sequence. Since not every u_i or v_j has length p, we fill up the remaining spaces with the special symbol e

https://doi.org/10.1017/S0956796801004233 Published online by Cambridge University Press

(denoting the empty sequence ε). When a recursive call to PCP computes a sequence of types, the FLAT predicate removes all of the e's from the result. FLAT also builds a 'list' of the non-e symbols, where the list 'cons' operation is represented by the arrow type constructor. So

$$\texttt{PCP}(1, 0, \varepsilon, u, 1, 0, 1, v, \gamma),\ \texttt{FLAT}_3(1, 0, \varepsilon, u, u'),\ \texttt{FLAT}_3(1, 0, 1, v, v')$$

implies that

$$u' = (1 \to 0 \to u) \text{ and } v' = (1 \to 0 \to 1 \to v)$$

The predicates ZERO, ONE and EPSILON are predicates for the singleton sets containing 0, 1 and e, respectively. They are necessary for some of the clauses because the Instance Restrictions do not allow nested type constructors. For the FLAT predicates, the reader may wish to consider these as 'moded logic programs' whose second-to-last and last arguments produce output based on the input of the other arguments. The output of the second-to-last argument consists of the input argument 'list' with all occurrences of e removed.

The top-level call to the predicate is represented by $\texttt{PCPinit}(w, w, \gamma)$, which gives rise to the constraints

$$\texttt{PCP}(a_1, a_2, a_3, u, b_1, b_2, b_3, v, \delta),$$

$$\texttt{FLAT}_3(a_1, a_2, a_3, u, u'),\ \texttt{FLAT}_3(b_1, b_2, b_3, v, v'),\ u' = w,\ v' = w$$

The calls to PCP and $\texttt{FLAT}_3$ build up a type using 1, 0 and $\to$, and unification at the top-level checks that the sequences of 0's and 1's thereby constructed are equal ($u' = w = v'$).

Theorem 5.1
Satisfiability is undecidable for constraint sets with the Instance Restrictions generalized from the Haskell single-parameter case.

6 Related work

Variations on multi-parameter parametric overloading have been described in the literature. Wadler & Blot (1989) provided a type system that included multi-parameter parametric overloading. Cormack & Wright (1990) provided an overload resolution algorithm, but did not provide a type system. In practice the algorithm may fail to terminate. The algorithm also encounters many order-of-type-checking dependencies, making it fairly incomplete.

The Gofer language (Jones, 1991) is a dialect of Haskell, experimentally extended with multi-parameter type classes (among other interesting extensions). Because multi-parameter type classes are an experimental extension, Gofer does not place any restrictions on overload instance declarations, and so even matching overload resolution is not guaranteed to terminate. The problem of increased opportunities for ambiguity with multi-parameter type classes has been recognized in early experiments with the Gofer extensions (Jones, 1991, 1994).

Peyton-Jones *et al.* (1997) investigate the design alternatives for multi-parameter type classes, and some of these design choices have been realized experimentally in the GHC and HUGS Haskell compilers. Peyton-Jones *et al.* acknowledge that

https://doi.org/10.1017/S0956796801004233 Published online by Cambridge University Press

Question: Is $\lambda x.$(PCPinit x x) well-typed?

```
class (PCPinit α β γ) where PCPinit :: α → β → γ
instance (PCP α1 α2 α3 α4 β1 β2 β3 β4 δ),
         (FLAT3 α1 α2 α3 α4 γ1 γ1'), (FLAT3 β1 β2 β3 β4 γ2 γ2')
         ⇒ (PCPinit γ1 γ2 (t α1 α2 α3 α4 β1 β2 β3 β4 γ1' γ2' δ))

class (PCP α1 α2 α3 α4 β1 β2 β3 β4 γ)
instance (PCP 1 0 e e 1 0 1 e t1)
instance (PCP α1 α2 α3 α4 β1 β2 β3 β4 δ),
         (FLAT3 α1 α2 α3 α4 γ1 γ1'), (FLAT3 β1 β2 β3 β4 γ2 γ2')
         ⇒ (PCP 1 0 e γ1 1 0 1 γ2 (t4 α1 α2 α3 α4 β1 β2 β3 β4 γ1' γ2' δ))
```

These two clauses for PCP correspond to a $u_i = 10$ and $v_i = 101$, for some i.

```
class (EPSILON α)      instance (EPSILON e)
class (ZERO α)         instance (ZERO 0)
class (ONE α)          instance (ONE 1)

class (FLAT3 α1 α2 α3 α4 β γ)
instance (FLAT2 α2 α3 α4 β γ), (ZERO α1)
         ⇒ (FLAT3 0 α2 α3 α4 (α1 → β) (t3,0 γ))
instance (FLAT2 α2 α3 α4 β γ), (ONE α1)
         ⇒ (FLAT3 1 α2 α3 α4 (α1 → β) (t3,1 γ))
instance (FLAT2 α2 α3 α4 β γ) ⇒ (FLAT3 e α2 α3 α4 β (t3,e γ))

class (FLAT2 α1 α2 α3 β γ)
instance (FLAT1 α2 α3 β γ), (ZERO α1) ⇒ (FLAT2 0 α2 α3 (α1 → β) (t2,0 γ))
instance (FLAT1 α2 α3 β γ), (ONE α1) ⇒ (FLAT2 1 α2 α3 (α1 → β) (t2,1 γ))
instance (FLAT1 α2 α3 β γ) ⇒ (FLAT2 e α2 α3 β (t2,e γ))

class (FLAT1 α1 α2 β γ)
instance (COPY α2 β), (ZERO α1) ⇒ (FLAT1 0 α2 (α1 → β) t1,0)
instance (COPY α2 β), (ONE α1) ⇒ (FLAT1 1 α2 (α1 → β) t1,1)
instance (COPY α2 β) ⇒ (FLAT1 e α2 β t1,2)

class (COPY α β)
instance (COPY e e)
instance (COPY 0 0)
instance (COPY 1 1)
instance (COPY α β), (COPY α' β') ⇒ (COPY (α → α') (β → β'))
```

Fig. 3. Example of PCP reduced to satisfiability.

multi-parameter type classes introduce new opportunities for ambiguity in Haskell languages. However they do not propose any solution to the problem.

Peyton-Jones (1998) describes the restrictions on type classes in an experimental extension of GHC with multi-parameter type classes. Our restrictions are more onerous in two respects. GHC allows overlapping instances (with unifying types) provided one type is an instance of another. We add negative context constraints to allow overlapping instances. Negative context constraints are limited to restricting

https://doi.org/10.1017/S0956796801004233 Published online by Cambridge University Press

outermost type constructors in substitutions, so for example the following is allowed by GHC but not by our type system:

```
class C α where ...
instance C [α] where ...
instance C [[Int]] where ...
```

The nearest that can be accomplished in our type system is:

```
class C α where ...
instance α ≠ Int ⇒ C [α] where ...
instance C [Int] where ...
```

The GHC restriction relies on an operational description of matching overload resolution to resolve the ambiguity due to overlapping instance types. This operational description fails with multi-parameter type classes and unifying overload resolution. Consider the type class:

```
class C α β where foo :: α → β
class C α Int where ...
class C Int Int where ...
```

The intention in GHC is that the first instance type is the default, whereas the second instance is used to resolve constraints of the form `C Int Int`. Therefore, if unifying overload resolution is used (with the strong overlapping restriction), the constraint `C` β β should be resolved against the default instance, and β will be instantiated to `Int`. But this contradicts the operational requirement that the constraint `C Int Int` should be resolved against the second instance type. In our type system, the first instance type includes the negative context constraint $\alpha \neq$ `Int`, so unifying resolution fails to resolve `C` β β against this instance type.

The second restriction we place on instance types, that is not imposed by GHC, is that an instance type has the form $\forall\overline{\alpha}.\ldots \Rightarrow c(\overline{\tau})$ where each τ_i is either a variable or of the form $\mathtt{t}(\overline{\beta})$ (Condition (2) of Definition 3.1). GHC places no restriction on the τ_i types. The reason for our restriction is to recognize when a constraint has a single instance against which it can be resolved, even when the constraint domain types do not match the instance type. An example of this is given by the second example after Definition 4.1, where there is an instance type `Foo` [α] [α] and a constraint `Foo` [β] [γ], and both parameters to `Foo` are domain types.

Jones (1995) has developed a framework for formalizing type inference algorithms for type systems with various forms of type constraints. In particular the notion of 'improving rules', the main contribution of Jones' work, has similar motivation to the notion of unifying resolution presented here. It appears likely therefore that the algorithm presented here could be expressed in his formal framework. On the other hand, Jones states:

> Our approach is to leave the task of finding suitable simplifying and improving functions to the designer of specific applications of qualified types... The work described here provides simple correctness criteria for simplifying and improving functions, but it does not provide any further insights into the construction of such functions for specific applications.

Independent of this work, the HUGS and GHC compilers have been released with experimental support for 'functional dependencies' (Jones, 2000). These allow class

https://doi.org/10.1017/S0956796801004233 Published online by Cambridge University Press

declarations to specify that the instantiations of some type parameters determine the instantiations of other type parameters. For example a class for the + operation can be defined by:

```
class Plus α β γ | (α β → γ) where (+) :: α → β → γ
```

On the one hand, this can be seen as a generalization of the overlapping restriction, where the latter corresponds to the functional dependency:

```
class C α1 ... αm β1 ... βn | (α1 ... αm → β1 ... βn) where ...
```

On the other hand, our approach of the overlapping restriction solves the same problems of ambiguity that are the stated motivation for functional dependencies. Our approach has the benefit of being simpler to state and to reason about, and appears to be easier to implement (because we do not have to track functional dependencies during type-checking). Functional dependences are more expressive than the overlapping restriction; it remains to be seen if this extra generality is useful in practice.

At the time of writing, the two compilers take different approaches to the problem of non-termination demonstrated by the example in section 4. The HUGS compiler infers the type

```
g :: (Foo [[[[a]]]] [[[[b]]]], Foo [[[[[b]]]]] [[[a]]]) ⇒
     [[[[b]]]] → [[[a]]] → Int
```

This appears to be due to a depth bound in the HUGS type checker. GHC on the other hand uses 'lazy context reduction', returning the type

```
g :: (Foo [[a]] [b], Foo [[b]] [a]) ⇒ [a] → [b] → Int
```

That is, the constraints are left unresolved. This has the problem that it introduces the possibility that type errors may be masked by unresolved type constraints. As evidenced by the example of the `Collects` class in section 2, this masking of type errors was one of the motivations for functional dependencies to begin with. Eager context resolution is triggered if an explicit type declaration is provided. It is not clear if this eager resolution is guaranteed to terminate; as explained in section 2, unresolved multi-parameter overload constraints can give rise to intermediate free type variables, even with a top-level type signature.

Odersky *et al.* (1995) describe a simplified form of single-parameter parametric overloading. Instance types have the form $\tau_1 \rightarrow \tau_2$, where $\tau_1 = \mathtt{t}(\alpha_1, \ldots, \alpha_n)$ and the type constructor `t` uniquely identifies the instance. Although there is a superficial similarity between this scheme and domain-driven unifying resolution, the two are quite different in motivation and properties:

1. Odersky *et al.* require that an overloaded operator have type $(\alpha \rightarrow \tau)$ where α is the only type parameter in the operator type, and the domain of an instance type $(\tau_1 \rightarrow \tau_2)$ then completely determines the instantiation of the instance type. In particular, all free variables in the instance type are free in τ_1. This is quite different from the overlapping restriction: we require that the free type variables in the domain types *determine* the free type variables

https://doi.org/10.1017/S0956796801004233 Published online by Cambridge University Press

in the range type, but do not necessarily *include* them. For example, the type of matrix multiplication (`Matrix` α) $\rightarrow$ (`Matrix` β) $\rightarrow$ (`Matrix` γ) is not allowed under the restriction of Odersky *et al.* They do not consider multi-parameter type classes at all in their work.

2. The motivation, mechanisms and properties of the two systems are quite different. The motivation for the approach of Odersky *et al.* is to remove the need for passing type dictionaries at run-time. They are also able to remove class declarations, and they are able to use their system to define extensible records. Run-time type dictionaries and class declarations are still essential in our type system. The overlapping restriction for multi-parameter type classes is motivated by coherence problems, and domain-driven unifying resolution is motivated by practical problems with matching resolution for multi-parameter type classes. Termination and satisfiability for unifying resolution with multi-paramete type classes are difficult or impossible to ensure, while the corresponding properties for the system of Odersky *et al.* are reasonably easy to verify.

Dubois *et al.* (1995) have proposed what amounts to an alternative to multi-parameter type classes. Their approach gives up on type-checking completely, and relies instead on link-time abstract interpretation to detect type errors. Interestingly, many of the examples they describe are also examples of the use of multi-parameter type classes. This provides further evidence of the practical usefulness of multi-parameter type classes.

7 Conclusions

We have described an approach to type-checking multi-parameter type classes. Domain-driven unifying resolution is simple, efficient and in practice very useful for some applications of parametric overloading. We have identified a restriction that is useful for multi-parameter parametric overloading, the overlapping restriction. With this restriction, domain-driven unifying resolution is guaranteed to preserve the set of valid resolutions for program overloadings. Domain-driven overload resolution is guaranteed to terminate provided the input overload constraints are satisfiable.

All existing Haskell single-parameter type classes are allowed by the overlapping restriction. With a straightforward generalization of our type system to include type constructor classes, all of the examples provided in section 1 are allowed by the overlapping restriction. The restriction allows the programmer the freedom to choose a trade-off between restrictions on overlapping of instances, and the amount of unifying resolution allowed. Specifically, given a multi-parameter type class declaration $(c ::_o^k (x : \forall \overline{\alpha_m}.\tau))$, then setting $k = m$ only requires that instances of x have non-unifiable types, while also disallowing unifying resolution. On the other hand, setting $k = 0$ only allows a single instance of x: considering Definition 3.3, the k domain types of two distinct instances are trivially unifiable when $k = 0$, so there cannot be two distinct instances for x. For the original problem of ambiguity with multi-parameter type classes, unifying resolution provides a solution to this problem with type class declarations satisfying the overlapping restriction. All of the examples considered in section 2 are of this form.

Our results shed some light on the implications of attempting unifying overload resolution (of any form) to solve the problems with ambiguity with multi-parameter type classes. Our negative results are that:

1. unifying overload resolution may fail to terminate for a program that is not typable, and
2. it is difficult to see what reasonable restrictions could be placed on (multi-parameter) overload instance types, that could ensure termination.

Elaborating on the latter point: without recursive instance types, overload resolution is guaranteed to be decidable (if potentially intractable). Primitive recursion is the simplest form of recursion from recursion theory. The linear algebra example in section 2 does in fact include primitive recursive instance types. Restricting instances to be 'primitive recursive' is actually fairly useless, since it rules out many examples (including some of the instance types in the linear algebra example). Yet even this draconian restriction is insufficient to ensure termination of unifying overload resolution. Any restrictions that eliminate this example must eliminate primitive recursive instance types; it is difficult to see what would be left with such restrictions.

Furthermore, although satisfiability ensures termination for domain-driven overload resolution, we have shown that satisfiability is undecidable, even with a type system that is arguably too restrictive to be useful.

It is not clear if these negative results necessarily rule out the use of unifying overload resolution. For many years, the Gofer language has provided multi-parameter type classes (with matching resolution) without any restrictions, and in practice non-termination has not been a problem. Based on the example involving the primitive recursion instance restriction in Sect. 4, it appears plausible that non-termination of overload resolution could become more of a practical issue with unifying overload resolution. Adding a depth bound or loop check to overload resolution could be a feasible approach to handling a non-terminating overload resolution algorithm. Overload resolution with a depth bound has been an experimental extension in the GHC compiler (Peyton-Jones, 1998). Our work sheds some light on the possibilities in this area.

Acknowledgements

Thanks to Mark Jones, Simon Peyton-Jones and Satish Thatté for helpful correspondence. Thanks to the anonymous reviewers for helping to greatly improve the original presentation.

References

Chen, K., Hudak, P. and Odersky, M. (1992) Parameteric type classes (extended abstract). *Proceedings of ACM Symposium on Lisp and Functional Programming*, pp. 170–181. ACM Press.

Cormack, G. and Wright, A. (1990) Type-dependent parameter inference. *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 127–136. ACM Press.

Dubois, C., Rouaix, F. and Weis, P. (1995) Extensional polymorphism. *Proceedings of ACM Symposium on Principles of Programming Languages*, ACM Press.

Hall, C., Hammond, K., Peyton-Jones, S. and Wadler, P. (1996) Type classes in Haskell. *ACM Trans. Programming Langu. Syst.* **18**(2), 109–138.

Jones, M. (1991) An introduction to Gofer. Available via ftp from `nebula.cs.yale.edu` in `pub/haskell/gofer`.

Jones, M. (1992) Qualified Types: Theory and Practice. PhD thesis, Oxford University Computing Laboratory.

Jones, M. (1993) A system of constructor classes: Overloading and implicit higher-order polymorphism. *Proceedings of ACM Symposium on Functional Programming and Computer Architecture: Lecture Notes in Computer Science 594*, pp. 1–10. Springer-Verlag.

Jones, M. (1994) Re: Multiple parameter classes. E-mail message on Haskell mailing list.

Jones, M. (1995) Simplifying and improving qualified types *Proceedings of ACM Symposium on Functional Programming and Computer Architecture*. ACM Press.

Jones, M. (2000) Type classes with functional dependencies. *European Symposium on Programming: Lecture Notes in Computer Science 1782*. Springer-Verlag.

Kaes, S. (1988) Parametric overloading in polymorphic programming languages. In: D. Sannella, editor, *European Symposium on Programming: Lecture Notes in Computer Science 300*, pp. 131–144. Springer-Verlag.

Liang, S., Hudak, P. and Jones, M. (1995) Monad transformers and modular interpreters. *Proceedings of ACM Symposium on Principles of Programming Languages*, pp. 333–343. ACM Press.

Nipkow, T. and Prehofer, C. (1993) Type reconstruction for type classes. *Proceedings of ACM Symposium on Principles of Programming Languages*, pp. 409–418. ACM Press.

Nipkow, T. and Snelting, G. (1991) Type classes and overloading resolution via order-sorted unification. In: J. Hughes, editor, *Proceedings of ACM Symposium on Functional Programming and Computer Architecture: Lecture Notes in Computer Science 523*, pp. 1–14. Springer-Verlag.

Odersky, M., Wadler, P. and Wehr, M. (1995) A second look at overloading. *Proc. ACM Conf. on Functional Programming and Computer Architecture*, pp. 135–146.

Peyton-Jones, S. (1998) Multi-parameter type classes in GHC. URL: `http://research.microsoft.com/Users/simonpj/Haskell/multi-param.html`.

Peyton-Jones, S., Jones, M. and Meijer, E. (1997) Type classes: an exploration of the design space. *Haskell Workshop*. Amsterdam, the Netherlands.

Volpano, D. and Smith, G. (1991) On the complexity of ML typability with overloading. In: J. Hughes, editor, *Proceedings of ACM Symposium on Functional Programming and Computer Architecture: Lecture Notes in Computer Science 523*, pp. 15–28. Springer-Verlag.

Wadler, P. and Blott, S. (1989) How to make *ad-hoc* polymorphism less *ad-hoc*. In: M. O'Donnell and S. Feldman, editors, *Proceedings of ACM Symposium on Principles of Programming Languages*, pp. 60–76. ACM Press.