Electronic Notes in Theoretical Computer Science 59 No. 2 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume59.html 29 pages

# Named Instances for Haskell Type Classes

Wolfram Kahl [1]

*Federal Armed Forces University Munich*
*Department of Computer Science, Institute for Software Technology*
*85577 Neubiberg, Germany*

Jan Scheffczyk [2]

*Federal Armed Forces University Munich*
*Werner-Heisenberg-Weg 102, App. 404*
*85579 Neubiberg, Germany*

**Abstract**

Although the functional programming language Haskell has a powerful type class system, users frequently run into situations where they would like to be able to define or adapt instances of type classes only *after* the remainder of a component has been produced. However, Haskell's type class system essentially only allows late binding of type class constraints on free type variables, and not on uses of type class members at variable-free types.

In the current paper we propose a language extension that enhances the late binding capabilities of Haskell type classes, and provides more flexible means for type class instantiation. The latter is achieved via *named instances* that do not participate in automatic context reduction, but can only be used for late binding. By combining this capability with the automatic aspects of the Haskell type class system, we arrive at an essentially conservative extension that greatly improves flexibility of programming using type classes and opens up new structuring principles for Haskell library design.

We exemplify our extension through the sketch of some applications and show how our approach could be used to explain or subsume other language features as for example implicit parameters. We present a typed $\lambda$-calculus for our extension and provide a working prototype type checker on the basis of Mark Jones' "Typing Haskell in Haskell".

[1] Email: kahl@ist.unibw-muenchen.de
[2] Email: jan.scheffczyk@gmx.net

©2001 Published by Elsevier Science B. V.

# 1 Introduction

One of the major success stories of Haskell is its type class system. Haskell's type classes allow a certain kind of ad-hoc polymorphism, and also enhance parameterisation of programs by allowing late binding of their members. In terms of implementations, this means that the dictionary that contains all the members of a certain instance of a class is supplied as a parameter in a late stage. However, this is not always possible, and so we find in the standard library pairs of functions like the following:

```
nub    :: (Eq a)                => [a] -> [a]
nubBy  :: (a -> a -> Bool)      -> [a] -> [a]
sort   :: (Ord a)               => [a] -> [a]
sortBy :: (a -> a -> Ordering) -> [a] -> [a]
```

The motivation of this design is that currently Haskell allows only one instance of a given class for a given type, and provides quite a few standard instances, so it is not possible to have, for example,

- an instance `Eq Integer` that considers two integers as equal if they are equivalent modulo the 38th Mersenne prime,
- an instance `Ord String` that ignores case, or,
- given an expression type `Expr` with `instance Show Expr` producing plain text output, an additional instance `Show Expr` producing TEX output.

Therefore, case-insensitive sorting of strings (that one does not want to wrap in a `newtype` constructor[3]) has to resort to the function `sortBy`. In such simple cases, this may not be a serious problem. But frequently one designs a component around larger classes, only to notice later that the ad-hoc polymorphism provided by type classes does not easily allow ad-hoc instantiations, and the component has to be re-factored along the lines of the `-By` pattern.

In this paper we propose a language extension that allows such ad-hoc instantiations. The central idea is to allow *named instances* besides the anonymous instances of current standard Haskell, and to provide a mechanism for explicit *instance supply*. For example, TEX output of expressions might be provided for via the following instance of `Show` for `Expr`:

```
instance ExprShowTeX :: Show Expr where
  show (Power e1 e2) = '{' : show e1 ++ "}^{" ++ show e2 ++ "}"
  ...
```

[3] With the definitions

```
newtype CIString = CI {unCI :: String}
instance Eq  CIString where compare = caseInsensitiveEqual
instance Ord CIString where compare = caseInsensitiveCompare
```

one would have `sortBy caseInsensitiveCompare = map unCI . sort . map CI`. The wrappers between the `newtype` isomorphisms and the application are in this case simple `map`s. In general, these wrappers can be harder to produce on the fly.

To preserve late binding possibilities, we have to suppress "context reduction" even in the presence of anonymous standard instances, so an application built on `Expr` might now have the following type:

```
calculator :: Show Expr => IO ()
```

For instantiation, we now have two possibilities: Since the type of `main` has an empty context, we can *force context reduction* via the following definition:

```
main = calculator
```

In this case, the anonymous `Show Expr` instance will be used for all occurrences of `show :: Show Expr ⇒ Expr → String`, and expressions are output (presumably) in their plain text format.

With our preliminary syntax for *explicit instance supply* we can force expressions to be output in their TEX format instead:

```
main = calculator # ExprShowTeX
```

In short, our extension unites the following features:

- the (dynamic) semantics of the current type class and instance system is preserved,
- some inferred types have larger contexts, and
- "later" binding of class members via explicit instance supply is possible.

One understanding of our extension is that we make some of the power of the target language of dictionary translations available to users, without burdening them significantly more with technical details than conventional Haskell classes do.

Another understanding also provides valuable guidance to our design. It is folklore that the Haskell type class system may be explained as an extremely restricted subset of ML module systems like the generative module system of SML [15,14], and Leroy's applicative module system [12] with manifest types [11], implemented in OCaml. In his proposal of *parameterised signatures* [6], Jones presents a nice overview of the differences between the SML and OCaml module systems, and also shows how parameterised signatures are closer to OCaml's system — the most relevant differences being the following:

- Parameterised signatures and the structures typed with parameterised signatures do not contain type components except as parameters.
- Parameterised signatures can be understood as easily producing polymorphic modules, which OCaml does not support.
- The ML module system issues of sharing and generativity are bypassed by relegating them to type checking.
- structures in the parameterised signature context are first-class values.

The system we propose in this paper is rather close to Jones' parameterised signatures, but does not go all the way to accept structures as first-class

citizens. Instead, we preserve compatibility with the automatic aspects of the Haskell 98 type class system.

Let us briefly list the parallels between our understanding of Haskell type classes on the one hand, and the OCaml module system and parameterised signatures on the other hand.

- Simple Haskell 98 classes correspond to OCaml signatures, containing the class members as `value` entries, and the argument type variable as an abstract `type` entry. (Haskell 98 modules in addition feature non-abstract type entries.)

  Abstract type entries in OCaml module types have two kinds of application: The first is for hiding implementations and occurs in the construct `ConcreteModule : AbstractModuleType`, which produces a module with a hidden implementation for the abstract type entry; in Haskell, this has a parallel only in the conventional module system in the shape of abstract naming of algebraic datatypes in export lists.

  The second application is for opening up instantiation possibilities via the `with`-clause of the OCaml module language (which introduces "module constraints"), as in `AbstractModuleType with t = int`; this instantiation corresponds to the instantiation of class argument type variables via instance declarations.
- Haskell instance declarations without type class constraints[4] declare structures for the signature corresponding to the associated class, but where the type entry corresponding to the class parameter has been turned, via a `with`-clause, into a manifest type according to its instantiation by the `instance` declaration.
- Haskell classes with superclasses correspond to signature inclusion.
- Multiple abstract type entries in an OCaml module type correspond to the parameters of a multiparameter class in Haskell — these are no problem *per se*, only for automatic type inference of overloaded functions.
- Haskell instance declarations with type class constraints declare functors for which the argument types are induced by the constraints and the result type is the signature corresponding to the instantiated class, but where the type entries corresponding to the class parameters have been turned into manifest types according to their instantiation by the `instance` declaration.

But, since a class may be instantiated only once for a particular type, in Haskell 98 there can be only *one* module in scope for every instantiation of a declared module type. (The "overlapping instances" allowed in some implementations do not significantly improve the situation with respect to ad-hoc instantiation.)

[4] In the Haskell 98 report, and in large part of the literature, type class constraints are called "contexts" — to avoid confusion with other kinds of contexts we are going to use the term *constraint* throughout.

According to these analogies, there are several choices for nomenclature. Talking about dictionaries and dictionary types seems to us too much implementation oriented, and introduces additional difficulties when talking about types representing functions between dictionaries. In OCaml, the name *module type* is used both for signatures, that is, "dictionary types", and for functor types, that is "types representing functions between dictionaries", therefore we shall use the term *module type* with the same meaning. The third module type constructor of OCaml, the `with` clause introducing module type constraints, is reflected by instantiation of "signature parameters", i.e., class arguments.

OCaml also uses the term "module" both for structures and for functors, so we feel that this is most appropriate in this paper, too. In this way, everything that has a module type is a module. For non-functorial modules we may use the term *structure*; this corresponds to its usage in the context of parameterised signatures. In addition, we may subsume both instances and conventional Haskell modules under the term "structure" — conventional Haskell modules with appropriate elements can also be used for instance supply, see Sect. 3.

The named instances we introduce in this paper can then be considered as lightweight modules, and Haskell constraints indicate parameterisation that can be put to use by explicitly supplying appropriate parameter instances. Therefore, our extension eliminates the constraint that there can be at most one module for every module type; we also introduce explicit functor application and even higher-order functors. The most important aspects of the OCaml module system not covered by our extension include the following:

- Opaque types in interfaces, where not only the implementation of a type, but even its identity are hidden, can, as in parameterised signatures, be mimicked only via free type variables as class constraint arguments that occur only once, see [6, Sect. 2.1] for details.
- We did not consider nested modules at all. Very limited forms arise naturally, just by virtue of the fact that Haskell instances have to be defined inside traditional Haskell modules, but we do not consider module types specifying module entries in signatures.

Since we strive for maximal compatibility with Haskell (98), we also inherit a Haskell feature that cannot be found in ML: a Haskell function that carries a constraint in its type may be considered as an *anonymous one-element functor*. This feature poses quite a few difficulties for a coherent module-system-inspired extension of Haskell's type class system, but it also gives additional power to applications of such a system. Therefore, a coherent treatment of these *module type constraints* is the main challenge in any effort to extend the use of the Haskell class system towards what amounts to module system capabilities that cannot be found in Haskell's traditional module system.

Our current solution, which is driven by the desire to change the current language design of Haskell as little as possible, is therefore the main contri-

bution of this paper. To better express our points, we use some new syntax; however, since throughout our extensions in this paper, we give proximity to Haskell 98 syntax higher priority than other considerations, we consider the concrete syntax of our extensions still open.

Through the desire to maximise compatibility and harmonious interaction with the Haskell 98 class system, we end up with a system that includes some rather complicated technical details, although its basic ideas are in fact simple. Therefore, we give ample space to introducing the features of our proposed extension and the motivations behind our design decisions in an informal manner. We start with introducing the simplest aspect, *named instances*, in Sections 2 and 3. The next level are functors, discussed in Sect. 4. Module type constraints as type qualifications are discussed in some detail in Sect. 5, and it turns out that rather fine-grained distinctions are necessary. Since our extensions give users more control over the satisfaction of constraints than the one-instance-per-type approach of Haskell 98, this means that we have to do *less* constraint reduction (Sect. 6). We then discuss aspects of how subclass relationships might translate into "module subtyping" in Sect. 7.

An interesting side-effect of our view of constraints is that the implicit parameters of Lewis *et al.* [13] are partly subsumed as "anonymous one-element functors with one-element structures as arguments"; we show this in Sect. 8.

In Sect. 9, we sketch a typed $\lambda$-calculus featuring module type constraints, and in Sect. 10 we briefly describe a prototype implementation of the type-inference aspects, based on Jones' "Typing Haskell in Haskell" [7].

## 2 Named Instances

As seen in the introduction, the first step towards enabling really late binding of class members even at types for which there are predefined instances is the ability to define and *name* non-standard instances for later reuse. Therefore, in our extended Haskell, we may provide names for instances, and, following the module analogy, understand these *named instance declarations* as declaring structures of the module type defined by the associated class declaration. For syntactic convenience, we let these names share the name space of Haskell module names, thus named instances must not use names that are also used as conventional module names. (This decision also opens up additional flexibility that will be exemplified in the next section.)

Let us now have a closer look at the named instance declaration from the introduction.

```
instance ExprShowTeX :: Show Expr where
  show (Power e1 e2) = '{' : show e1 ++ "}^{" ++ show e2 ++ "}"
  ...
```

It brings the module name "`ExprShowTeX`" into scope, and binds it to the structure determined by the body of the instance declaration (together with

default definitions from the class — for sake of simplicity we shall completely ignore the question of such default definitions in the sequel).

Since "`ExprShowTeX`" is a module name, we may use the qualified identifier `ExprShowTeX.show`, which has the type `Expr → String`. The module name `ExprShowTeX` itself is considered to have the module type `Show Expr`, which we write "`ExprShowTeX :: Show Expr`".

Note that such a *named instance declaration* does *not* introduce a "real" instance for the class `Show` that would be available for automatic insertion wherever `show` is used at the type `Expr → String`.

The class member `show` still has type `(Show a) ⇒ a → String`, containing the *constraint* "`(Show a)`". This means it is parameterised over possible instances for `Show`, or, more precisely, over structures of module type `Show a`. In Haskell 98, there is no way to explicitly instantiate such parameters: it all happens implicitly, via anonymous instances and constraint reduction.

With named instances, we can have more than one instance in scope for the same type, and we introduce a possibility to explicitly supply instances as parameters. We use the infix operator "`#`" as application to instance parameters, or, as we shall say from now on, to module parameters. Therefore, "`#`" is the elimination form for types constructed with "⇒", in the same way as standard application is the elimination form for types constructed with "→". So we may write "`show # ExprShowTeX`", and this has type `Expr → String`. Essentially, this means that "`#`" corresponds to dictionary application. However, in dictionary translations it is necessary to make all references to dictionaries explicit, while in our system, much of the implicit character of the Haskell class system is preserved.

Let us now turn to another very simple example which hints at possible uses of named instances for structuring algebraic libraries, and which, at the same time, also illustrates some further effects. Consider the following class declaration for monoids:

```
class Monoid m where
  unit :: m
  comp :: m -> m -> m
```

Imagine further a complex body of utilities built around this class, including, as a simple example:

```
complist :: Monoid m => [m] -> m
complist = foldl comp unit
```

Desiring to use all this in integers, the user of this class in standard Haskell now faces the problem of indicating which of the two well-known monoids on integers to use for defining *the* instance of `Monoid` for `Integer`. For the other monoid, the standard escape route is `newtype`, which allows to define a different instance for the same class on a *different* type which, however, has the *same* implementation. Jones perceives the use of `newtype` declarations to

"clutter up programs" [8]; this stems from the fact that they lead to artificial distinctions on essentially the *same* type.

With named instances, this problem is easily solved. We just define:

```
instance AddMonInt  :: Monoid Integer where
  unit = 0
  comp = (+)
instance MultMonInt :: Monoid Integer where
  unit = 1
  comp = (*)
```

A complex application wishing to mix both views on integers just needs to supply the respective instances to different library function invocations:

```
foo :: [[Integer]] -> Integer
foo = (complist # AddMonInt) . map (complist # MultMonInt)
```

Compare this with the Haskell 98 type inferred for the function

```
bar i x = complist $ map complist $
          replicate i $ replicate i $ (x :: Integer)
```

which is `(Monoid Integer)` $\Rightarrow$ `Int` $\rightarrow$ `Integer` $\rightarrow$ `Integer` with only *one* constraint (the application `bar # AddMonInt` will use addition on both levels of lists). Since both instances provided above are named, we assume that there is no anonymous instance of type `Monoid Integer` in scope, so the constraint would be locally unresolvable.

Even apart from the monomorphism restriction, such unresolvable constraints involving non-variable types are illegal in standard Haskell. In our setting however, admitting such constraints makes sense. The delayed constraint may be satisfied at a later stage, for example via the explicit module parameter supply `bar # MultMonInt`.

In contrast with anonymous instances, automatic export does *not* make sense for named instances. Since they share the module name space, they may be exported with `module` entries in export lists. They will be imported in the same way as other module entries — then the instance name is available as an argument to "`#`"-application, and members may be accessed as qualified identifiers. Indeed, there is no reason to forbid using conventional modules as arguments for "`#`"-application, provided they contain appropriately named members — we show an example for this in the next section.

## 3 Named Instances for Multi-Parameter Classes

Multi-parameter classes are recognised as extremely useful, but it is not yet clear how the design choices involving context reduction and type inference should be resolved [16]. With named instances, it is possible to use multi-parameter classes without ever defining anonymous instances for them, so problems of context reduction and type inference can essentially be avoided.

In the following, we therefore allow named instances and forbid anonymous instances for multi-parameter classes, thus reaping a significant portion of the benefits of multi-parameter classes without incurring the usual costs.

Let the following module defining a collection class be given:

```
module Coll where
class Collection c e where
  empty  :: c e
  insert :: e -> c e -> c e
  fold   :: (e -> r -> r) -> r -> c e -> r
```

Let us furthermore assume a stand-alone module that implements sets using some tricky balanced tree implementation and demands the constraint `Ord` on element types:

```
module SetColl(Set(),empty,insert,fold) where

data Set a = EmptySet | TrickyBalancedTree ... a ...

empty  :: Ord a => Set a
empty  = EmptySet

insert :: Ord a => a -> Set a -> Set a
insert = ...

fold   :: Ord a => (a -> r -> r) -> r -> Set a -> r
fold   = ...
```

As in this example, libraries built for our extended language would tend to directly implement appropriate class interfaces. This improves much upon the current practice of unqualified imports that foster the unfortunate tradition of type-indicating names like `foldSet`. Any module that imports `Coll` unqualified will have to do `qualified` import of modules like `SetColl`, because otherwise e.g. `empty` would refer both to the class member `Coll.empty` and to `SetColl.empty`.

Accordingly, consider the following module header:

```
module Main where
import Coll
import qualified SetColl
```

Inside this module, it now makes sense to consider `SetColl` as an instance of `Collection` with the type `forall a . Ord a ⇒ Collection SetColl.Set a`, which is equivalent to the original (anonymous) type of the module `SetColl`. This means that this is a *fully polymorphic instance* that can be used at arbitrary (ordered) element types. The module `SetColl`, considered as a functor in this way, hides the *implementation* of the `Set` datatype constructor,

but it does not hide its *identity*, nor its dependence on types provided by the functor argument — these are the effects that lead to the bypassing of the sharing issue, as discussed by Jones [6].

An application that has been designed to be independent of the implementation of collections might still use collections of different element types, so it can be defined to depend on such a polymorphic instance:

```
app_main :: (forall a . Ord a => Collection c a) => IO ()
```

Given all the above, it is now possible to instantiate this application inside module `Main` in the following way, providing an explicit module type to the conventional Haskell module `SetColl`:

```
import Application(app_main)
main = app_main #
       (SetColl :: forall a . Ord a => Collection SetColl.Set a)
```

We conjecture that for most uses of multi-parameter classes, the resulting need to specify the instance to be used will be only a small burden to the user. Perhaps it may even be a liberation not to have to think about possible overlaps, and being able to specify the intended instance, instead of having to set up a puzzle for the compiler, and hoping that the compiler will solve it correctly, and in finite time.

From the examples in this and the preceding section it should be clear that named instances together with explicit module parameter supply are a natural remedy to the commonly perceived weakness of type classes in Haskell 98 which are, citing [5], only "well suited to overloading, with a single natural implementation for each instance of a particular overloaded operator". For many applications, already this simple extension would be extremely valuable. However, once the basic correspondence between type class concepts and OCaml module concepts is established, further extensions are natural consequences of the introduction of named instances, and are discussed in the next few sections.

## 4 Instance Functors

In module systems, a functor is a function taking modules as arguments, and having other modules as results. In the dictionary translation of classes, instances with constraints are translated into functions between dictionaries. Since dictionaries may directly be viewed as modules, we immediately see that such instance declarations give rise to functors.

The constraints then express the types of the arguments, and the target class of the instance is the result type. On the whole, we get a *functor type*, for which we conveniently use syntax already present in Haskell. In contrast, module types that are not functors are called *atomic*.

Let us consider a facility to show lists not in the standard way with brackets

and commas, but with every element on a separate line:

```
instance ShowListUL :: Show a => Show [a] where
  show xs = unlines (map show xs)
```

Now `ShowListUL` has the *functor type* `Show a` $\Rightarrow$ `Show [a]`. Functor application does of course expect appropriately typed arguments, so we cannot provide `ShowListUL` as an argument to `show`; instead we have to apply the functor to some argument first, and then provide the result to `show`. For this *functor application*, we also use the infix operator "`#`": The functor application "`ShowListUL # ExprShowTeX`" has type `Show [Expr]`, and we may write

```
show # (ShowListUL # ExprShowTeX)
```

which has type `[Expr]` $\rightarrow$ `String`. However, we may not always be working with a fixed, predefined functor argument, so we need *module variables*:

```
showListUL # i = show # (ShowListUL # i)
```

The following type is then automatically inferred:

```
showListUL :: Show a => [a] -> String
```

The case where a functor takes several module arguments at first poses a problem, since in Haskell 98, the types of the following two instances (apart from being named) would be considered as completely equivalent [5]:

```
instance D1 :: (Show a, Show b) => Show (a,b) where ...
instance D2 :: (Show b, Show a) => Show (a,b) where ...
```

If we accepted this, then it would not be possible to give a reasonable semantics to applications like `D1 # ExprShowTeX`. However, since these constraint lists arise in a place where they are written by the programmer, we regard the order of these lists as intentional, so the above is equivalent to specifying curried functor types (we accept the above syntax only for backward compatibility):

```
instance D1 :: Show a => Show b => Show (a,b) where ...
instance D2 :: Show b => Show a => Show (a,b) where ...
```

## 5 Module Type Constraints

We now look in more detail into the problems generated by our view that constraints in the types of functions should be considered not just as class constraints, but as general *module type constraints*. The real problems come from our desire to let these module type constraints be satisfied not only by "module supply" via "`#`", but also by anonymous "default instances" via the conventional class system of Haskell. In effect, we design our extensions in such

[5] In 4.1.4 of the Haskell 98 report, the type generalisation preorder for qualified types is defined and implies that types differing only in constraints that are equal up to permutation have to be considered as equivalent. Another hint in that direction is the mention of "most general instance context" in 4.3.2.

a way that we do not break existing Haskell programs. On the contrary, we enhance the reusability of existing traditional library modules by admitting different methods of module supply for constraints, and by exposing more constraints to user-defined module supply.

### *5.1 Ordered and Unordered Constraints*

Consider the following type signatures and definitions as given:

```
f  :: Eq a => a -> c -> (a,c)
bs :: Eq b => [b]
g x = f x bs
h x = (\ ff -> ff x bs) f
```

The Haskell 98 interpreter Hugs98 and the compiler GHC-4.08 derive the typings (up to $\alpha$-conversion):

```
g :: (Eq a, Eq b) => a -> (a,[b])
h :: (Eq b, Eq a) => a -> (a,[b])
```

In contrast, the compiler nhc98 derives oppositely ordered constraints — for other similar examples pairs where Hugs and GHC derive different orders, nhc98 derives the same orders.

Of course, `g` and `h` are the same function, and since in Haskell 98 the two typings are considered as equivalent, this is no problem — if explicit type signatures are added, all three systems accept any ordering of constraints.

Since every reasonable typing discipline should obey the subject reduction property — $\beta$-reduction can only lead to a more general type — this example shows that the structure of an expression induces *no natural ordering* for inferred constraints.

A canonical order might be achieved by orientation at the structure of the inferred type, for example preferring `(Eq a, Eq b) => a -> (a,[b])` because of the order of occurrence of the type variables in `a -> (a,[b])`, but this breaks down for constraints on types that do not occur in the raw typing. Therefore, unordered constraints are a natural result of type inference for Haskell 98 expressions, even in their embedding into our extension.

More precisely, it is function application that forces joining of the ordered constraints in the types of the two constituents of the application into the unordered constraint of the type of the whole application. For reasons of compatibility with the Haskell 98 view of constraints, it does not make sense to have functor types in unordered constraints. Therefore, unordered constraints may only contain atomic module types, while ordered constraints may contain even higher-order functors. For a functor type $\mu$ in the ordered constraints of the two constituents of an application we have to apply constraint reduction (see the next section), which is of course *only* possible, if the class environment contains an anonymous default functor of a type $\mu'$ such that $\mu$ and $\mu'$ both map to the same Haskell 98 functor type.

With the unordered polymorphic constraints of the example above, there is no way to allow the user to direct their satisfaction separately, since any structure `M :: Eq t` for any type `t` could satisfy both constraints.

In our extension, however, separate satisfaction of constraints can be enabled even in more general cases by providing explicit module arguments in the same way as functor arguments — this is a result of considering a function with constraints in its type as an implicit functor with an anonymous one-element result signature. So we are able to make module parameterisation explicit, and in analogy to the definition of `showListUL` in Sect. 4, we now may define:

```
k # i # j = (f # i) (bs # j)
```

For this, only the type `Eq a` ⇒ `Eq b` ⇒ `a` → `b` can be inferred, so here we have an *ordered list of constraints*.

Since the two effects may occur together, we have to partition the constraint component of types into what we are, from now on, going to call "ordered constraints" and "unordered constraints". As an example for the coexistence of ordered and unordered constraints, consider the following:

```
k' # i # j  x y = let k0 z = (f # i) z (bs # j) in
                  if x <= y && k0 x <= k0 y then k0 x else k0 y
```

Here we have `Eq a` and `Eq b` in the ordered constraints, since they are associated with the module variables `i` and `j`, and in addition `Ord a` and `Ord b` in the unordered constraints, motivated by the two occurrences of `<=`. Therefore, the following typing is inferred:

```
k' :: Eq a => Eq b => {Ord a, Ord b} => a -> a -> b
```

Note that from now on we shall write unordered constraints with braces `{}`, not with parentheses `()`. In our present design these braces, representing the unordered (Haskell 98) constraint, clearly indicate to which part of the constraint a module type belongs to. Therefore, unlike in Haskell 98, the following types should then be considered as different:

```
ff  ::  Eq a  => [a] -> Bool
ff' :: {Eq a} => [a] -> Bool
```

However, in our investigations this difference would have noticeable consequences together with a certain set of decisions concerning module type subtyping (see Sect. 7) that seems to be a useful compromise and is implemented in the prototype, but will not be discussed any further in the present paper.

Without subtypes, there is a noticeable difference only inside type class definitions:

```
class Foo a where
  foo1 ::  Foo b  => a -> b -> Bool
  foo2 :: {Foo b} => a -> b -> Bool
```

The members of this class have different types:

```
foo1 :: Foo b => {Foo a} => a -> b -> Bool
foo2 :: {Foo a, Foo b}   => a -> b -> Bool
```

As a result, the late binding capabilities of `foo2` are much more restricted than those of `foo1`.

### *5.2 Module Supply*

It is natural to allow satisfaction of module types in unordered constraints where no ambiguity arises, such as in the following (assuming the named instance `FooChar :: Foo Char`):

```
q i = foo2 'c' (i :: Int) # FooChar
```

This has the type `{Foo Int}` $\Rightarrow$ `Int` $\rightarrow$ `Bool`. We may allow this because the type of `FooChar` is not an instance of `Foo Int`.

This "automatic selection" of parameter position by actual parameters may be generalised to the ordered constraints, allowing out-of-order `#`-application to modules. This means that the first type-compatible argument position in the ordered constraints is used, or the *only* type-compatible element of the unordered constraints.

```
f :: Eq Int => Eq Char => Eq [Int] => Eq [Char] =>
     ([Int],[Int]) -> ([Char],[Char]) -> ([Int],[Char])

MyEqLC :: Eq [Char]

-- f # MyEqLC :: Eq Int => Eq Char => Eq [Int] =>
--             ([Int],[Int]) -> ([Char],[Char]) -> ([Int],[Char])
```

The case of several type-compatible elements in the unordered constraints has to be rejected as an unresolvable ambiguity.

This convention — which is in fact nothing more than syntactic sugar — avoids having to use module variables, and thus reduces the syntactic heaviness of supplying parameters to specific parameter positions. Although it may seem somewhat ad-hoc, we consider this usability aspect a strong argument in favour of including this feature — it is also relatively cheap to implement in the type checker and in the formal design, where the details are defined (Sect. 9). For functors, however, out-of-order application is not an option, since it makes module type inference undecidable.

### *5.3 Typing Functors*

Finally, we have to decide how far we take module polymorphism. Consider the following definition:

```
f # k # j x y = ((==) # (k # j)) x y
```

The module variable `k` obviously must have a functor type, but we have no information about its argument type. Thus the inferred type for `f` could be `f :: (? ⇒ Eq a) ⇒ ? ⇒ a → a → Bool` — where "?" might denote a *module type variable*. Since we perceive this as an over-generalisation of very limited use, we tend to exclude module type variables from the syntax, and not to allow definitions that imply constraints with module types in which module type variables occur. The definition above may then be legalised by adding an appropriate type signature.

Functors are polymorphic by nature, but Haskell's first-order type inference makes it impossible to use arguments at polymorphic types. Extensions in Hugs and in GHC include the feature of *rank-2-types* which we adopt for our constraints. We have shown an application of a function with a second-order constraint in Sect. 3; here we show how such a function might be defined:

```
int_component    :: Collection c Integer => IO ()
string_component :: Collection c String  => IO ()

app_main :: (forall a . Ord a => Collection c a) => IO ()
app_main # coll = do int_component    # coll
                     string_component # coll
```

It would of course be more comfortable without explicit instance variables:

```
app_main :: (forall a . Ord a => Collection c a) => IO ()
app_main = do int_component
              string_component
```

This could be made possible by an extension of *forcing constraint reduction*, which is discussed at the end of the next section.

There is no intrinsic restriction of this kind of polymorphism to functors; the above example could be rewritten for `Collection`s defined without constraints, for example via lists.

It would of course be most elegant if we would not have to think about such constraints while designing the application, so we would like to have:

```
app_main :: (forall a . Collection c a) => IO ()
```

However, this cannot be directly applied to `SetColl` in our current system; one would have to supply (transparently through type quantification)

```
instance PolyOrd :: (forall a . Ord a)
main = app_main # (SetColl # PolyOrd)
```

The polymorphic ordering instance might be defined via Hinze's and Peyton Jones' *derivable type classes* [3] or Hinze's fully polymorphic *atomic* instances from [2,1].

Short of requiring such *fully* polymorphic ordering instances, one might also consider polymorphism restricted to ground types generated by a limited set of type constructors, which would allow more compile-time control:

```
app_main :: (forall a BuiltFrom {Int,Maybe,(,)} . Collection c a) ⇒ IO ()
```

Obviously, this area deserves further investigation.

# 6 Constraint Reduction

As we have seen in the previous sections, we must have a closer look at *constraint reduction* and *module type instantiation*. We say that a module type is instantiated if its type variable(s) are made less polymorphic. In contrast, a constraint is reduced if we delete one of its module types. In addition, moving a module type from the unordered to the ordered constraint is also a kind of constraint reduction.

The concept of how constraints are reduced and module types are instantiated directly influences the use and flexibility of module type constraints. The tradeoff to be balanced is between compatibility with Haskell 98 and the desire for maximum flexibility. An eager approach to constraint reduction would enforce ultimate compatibility, but incur a severe loss of flexibility, whereas a "fairly lazy" approach is as flexible as possible, while only compromising compatibility in tolerable ways. *Fully lazy* constraint reduction is not feasible, since it would produce ambiguous constraints and inhibit polymorphic recursion [16].

*Monomorphic Module Types*

As we have seen in Sect. 2, monomorphic module types such as `Monoid Integer` play an essential rôle when using named instances. Therefore, we will not delete a monomorphic module type from constraints, since the user may decide at a "later" stage which structure to choose.

In order to accept Haskell 98 programs, we have to allow constraint reduction through explicit type annotations; this will then eliminate monomorphic module types for which a satisfaction is entailed from anonymous class instances. Thus, adding for example the type signature

```
bar :: Int -> Integer -> Integer
```

is only legal when a *default* instance declaration for `Monoid Integer` is in scope, and then this type signature forces `bar` to have precisely this type. This perfectly reflects the behaviour of Haskell 98.

Note that every compilable Haskell 98 program has at least one explicit type annotation[6] `main :: IO(a)`, which is forced by the Haskell compiler if it is not explicitly given by the programmer — this would be the ultimate point of forcing away delayed constraints by supplying the anonymous default instances from the Haskell 98 class system.

[6] Since the type variable `a` may be instantiated in the process, this is a somewhat special case.

*Ambiguous Constraints*

As seen above, there may be problems when two module types $M_1$ and $M_2$ in an *unordered* constraint are ambiguous. This is the case if there exist substitutions $\theta_1$ and $\theta_2$ such that $\theta_1 M_1 = \theta_2 M_2$. If, according to the definition of Peyton Jones *et al.* [16], $M_1$ *entails* $M_2$, denoted by $M_1 \Vdash M_2$, then there is a default functor $\Phi$ justifying this entailment. So we can delete the constraint $M_2$ and supply $\Phi$ `#` $M_1$ into the original parameter position of $M_2$. This is the *only* case where automatic constraint reduction is left intact.

Of course, explicit type annotations with ordered constraints may be used to prevent this automatic constraint reduction:

```
f :: Eq a => Eq [a] => [a] -> [a] -> Bool
f xs ys = (head xs) == (head ys) || (tail xs) == (tail ys)
```

However, we cannot allow type annotations that change ordered constraints into unordered constraints, because ordered constraints *only* arise from the use of module variables and from explicit type annotations.

Note that the definition of entailment of Peyton Jones *et al.* [16] also includes superclass declarations which we regard as projection functors. Since the problem of ambiguous constraints only involves module types related to the *same* class, we do not need to include superclass projection functors in the definition of entailment used in our context. Note further that with our notion of automatic constraint reduction, "lonely" constraints, as for example `{Eq [a]}`, are always treated as irreducible.

*Forcing Constraint Reduction*

As we have seen in the preceding sections, we can *force* constraint reduction via explicit type annotations. This will be *full* Haskell 98 constraint reduction without exceptions, and it can force away only from elements of the unordered constraints. Of course it needs to have the corresponding `instance` (and `class`) declarations in scope.

## 7 Instance Subtyping and Joint Instance Declarations

According to the above, a named variant of the default `Ord` instance for `Maybe`, with the header `instance OM :: Ord a => Ord (Maybe a)` would have the type `Ord a` $\Rightarrow$ `Ord (Maybe a)`.

Now consider the implications of the fact that `Ord` is defined as a *subclass* of `Eq`. With the usual understanding of subclass relationships, the module type `Ord (Maybe a)` should also be a subtype of the module type `Eq (Maybe a)`. This implies that wherever an instance of type `Eq (Maybe a)` can be used, an instance of type `Ord (Maybe a)` should also be acceptable.

At first sight this seems to be no problem: class declarations for classes with superclasses can be seen as containing *implicit* functor definitions for

projection functors from the signature of the subclass to the signatures of the superclasses. A radical view on this subtyping relation implies that the current definition of the class `Ord` is equivalent to the following expanded version:

```
class Ord a where
  (==), (/=), (<=), (<), (>=), (>) :: a -> a -> Bool
  compare :: a -> a -> Ordering
```

This would then also support *joint instance declarations*, which have recently been proposed in a mailing list discussion. They address mainly the following scenario which is relevant to adaptability of library classes:

Assume a library class declaration

```
class C a where
  m1 :: a
  m2 :: a
```

for which a user defined the following instance:

```
instance C Int where
  m1 = 1
  m2 = 2
```

Now the library undergoes some redesign, and it is decided that splitting the class has advantages, so now we have:

```
class          C1 a where m1 :: a
class C1 a => C  a where m2 :: a
```

But in Haskell 98, this breaks the user's instance definition! Since `C1 Int` is a supertype of `C Int`, the proponents of joint instance definitions propose that the user's instance definition should be considered as legal, because it really defines a structure of type `C Int`, from which a structure of type `C1 Int` may be extracted by the corresponding superclass projection functor.

In contrast, the "conventional" instance definition

```
instance C Int where m2 = 2
```

really defines an anonymous functor of type `C1 Int` $\Rightarrow$ `C Int`, which is used to *construct* a structure of type `C Int` from a previously available structure of type `C1 Int`.

In the same way, our named instance from the beginning of this section should have the type `MO :: Eq (Maybe a)` $\Rightarrow$ `Ord a` $\Rightarrow$ `Ord (Maybe a)`. We tend to put `Eq` first since in common understanding no instance of a subclass can be defined before there are instances of all its superclasses. Now this is quite counter-intuitive, but making this type explicit in the instance declaration would fail to indicate that the `Eq` structure of the first argument will end up as the `Eq` component of the result. Furthermore, in the presence of more than one instance at some type we run into the same multiple-inheritance problems as in C++ — just imagine additional class declarations as the following:

```
class Eq a => R a
class (R a, Ord a) => S a
```

Now we could define two joint instances for `R` and `Ord`, equipped with different equalities, and a non-joint implicit functor for `S`:

```
instance R1 :: R   Int where (==) = eq1  -- R1 :: R Int
instance O2 :: Ord Int where (==) = eq2  -- E2 :: Eq int
instance S1 :: S Int          -- S1 :: R Int => Ord Int => S Int
```

As a result, `S1` somehow contains the two different bindings `eq1` and `eq2` for the class member `(==)`, and no sensible automatically defined projection functor is available for using `(S1 # R1 # O2)` of type `S Int` at the subtype `Eq Int`, as for example in the expression "`(==) # (S1 # R1 # O2)`".

The whole topic of subtyping of module types therefore has to be treated with great care.

From this discussion it should be obvious that joint instance definitions are really independent from named instances and module type constraints, but we claim that our system is a good way to explain the issues behind joint instance definitions. This is especially so since default definitions for members in class definitions may be considered as inducing a set of functors that turn different subtypes of the defined class type into the complete class type. How these default definitions are to interact with joint instance definitions is probably much easier to analyse using our functor concept.

This problem is closely related to the problem that the identity of atomic module types (i.e., signatures) in Haskell is defined by *class name*, and not by the *contained signature*. From this point of view, implicit parameters are more honest since they use *anonymous* module types as arguments, and an accumulation of implicit parameter constraints may even be considered as a multiple-member module type. However, there is no way to supply a single multiple-member structure as an argument that instantiates all these parameters. Finding a better way to use "anonymous classes" would therefore be a useful continuation of our present work.

## 8 Implicit Parameters

The Haskell interpreter Hugs [4] provides an experimental extension called "implicit parameters" [13], introducing dynamic bindings. We argue that implicit parameters cover a subset of the possibilities of module type constraints, but are easier to use at least in simple cases.

A translation of the "File IO" example given by Lewis *et al.* [13] into Haskell with module type constraints will naturally use zero-parameter type classes and local instance declarations not discussed in this paper. For good measure, we throw in a joint instance definition, used at a supertype:

```
class StdIn  where stdIn  :: IO Handle
class StdOut where stdOut :: IO Handle
class {StdIn, StdOut} => StdIO

instance StdStdIO where
  stdIn  = stdin
  stdOut = stdout

getLine :: {StdIn } =>           IO String
putStr  :: {StdOut} => String -> IO ()

session :: StdIn => StdOut => IO ()
session = do putStr "What is your name?\n"
             s <- getLine
             putStr ("Hello, " ++ s ++ "!\n")

main = do h <- openFile "foo"
          instance H :: MkStdOut where stdOut = h
          session # StdStdIO # H
```

We feel that, on the one hand, the approach of module type constraints gives much more flexibility and syntactically fits better into Haskell 98. On the other hand, implicit parameters are easier to handle because the programmer can use *functions* to modify them. Therefore, both approaches might "peacefully" coexist in a Haskell environment.

## 9 A Typed $\lambda$-Calculus with Named Instances

In this section we formalise named instances and module type constraints by presenting a type system and a type inference algorithm for a small language corresponding to the relevant extension of a subset of Haskell 98, covering only the central aspects of our extension. We present this as a fairly standard typed $\lambda$-calculus with `let`-polymorphism.

### *9.1 Notation and Utility Functions*

In this section we will introduce syntactical notations (see Fig. 1) and define some basic functions. We follow the common notations of [13,9].

Distinguishing $\lambda$-bound variables ($x$) from `let`-bound variables ($p$) is not really necessary, but makes the reading of formulae easier. Module variables have their own name space (which they share with Haskell 98 modules). Types are constructed from type variables via the function type constructor $\rightarrow$ and other type constructors $\chi$.

Module types are simpler than types in that there is only the functor type constructor $:\Rightarrow$ (associating to the right) for producing non-atomic module types. As noted in Sect. 5, a constraint consists of an ordered part $O$ and an unordered part $U$. The ordered part is a list of module types (and we use Haskell list syntax), while the unordered part is a *set* of *atomic* module types.

Constraints are used to construct *qualified types* $\sigma$ which are then of the shape $O \rhd U \Rightarrow \tau$. For qualified types with empty constraints we just write $\tau$.

A Haskell 98 context $\Gamma$ is a finite partial function associating variables from $\mathsf{Var}$ with either a qualified type ($\mathsf{QType}$) or a type scheme ($\mathsf{TScheme}$), where $\mathsf{Var}$ contains $\lambda$-bound variables and `let`-variables. An additional context $\Delta$ is provided, associating module names and module variables with module types.

| | |
|---|---|
| $\lambda$-variables | $x$ |
| `let` variables | $p$ |
| Terms or expressions | $e, f, t \ ::= \ x \mid p \mid \lambda x.t \mid e\ f \mid \mathtt{let}\ p = e\ \mathtt{in}\ t$ |
| Module variables $\mathsf{MVar}$ | $i, j$ |
| Module expressions | $m \ ::= \ i \mid m_1 \,\#\, m_2$ |
| Type variables | $\alpha$ |
| Type constructors | $\chi$ |
| Types | $\tau \ ::= \ \alpha \mid \tau \to \tau \mid \chi\ \tau_1 \ldots \tau_n$ |
| Class predicate symbol | $\kappa$ |
| Module type variable | $\nu$ |
| Atomic module type | $\kappa\langle\tau\rangle$ |
| Module types $\mathsf{MType}$ | $\mu \ ::= \ \mu^0 \mid \forall\bar{\alpha}.\mu^0 \qquad$ where $\bar{\alpha} \subseteq tvars(\mu^0)$ |
| | $\mu^0 \ ::= \ \mu^1 \mid (\forall\bar{\alpha}.\mu_1^1) :\Rightarrow \mu_2^1 \qquad$ where $\bar{\alpha} \subseteq tvars(\mu_1^1)$ |
| | $\mu^1 \ ::= \ \mu^0 \mid \mu^0 :\Rightarrow \mu^1 \mid \kappa\langle\tau\rangle \mid \nu$ |
| Ordered constraint | $O \ ::= [\mu_1, \ldots, \mu_k]$ |
| Unordered constraint | $U \ ::= \ \{\mu_1, \ldots, \mu_k\}$ |
| Constraint | $O \rhd U$ |
| Qualified type $\mathsf{QType}$ | $\sigma \ ::= \ O \rhd U \Rightarrow \tau$ |
| Type scheme $\mathsf{TScheme}$ | $\eta \ ::= \ \forall\bar{\alpha}.\sigma \qquad$ where $\bar{\alpha} \subseteq tvars(\sigma)$ |
| Substitution | $\theta, \hat{\theta}$ |
| Haskell 98 context | $\Gamma : \mathsf{Var} \twoheadrightarrow (\mathsf{QType} + \mathsf{TScheme})$ |
| Module context | $\Delta : \mathsf{MVar} \twoheadrightarrow \mathsf{MType}$ |

Fig. 1. Syntax

When we write $S_1 \oplus S_2$, this denotes the union $S_1 \cup S_2$ and additionally expresses the fact that $S_1$ and $S_2$ are disjoint. We write $\mathsf{mgu}(\tau_1, \tau_2)$ resp. $\mathsf{mgu}(\mu_1, \mu_2)$ to denote the most general unifier for types $\tau_1$ and $\tau_2$, or module types $\mu_1$ and $\mu_2$, respectively.

In Sect. 5 we argued that it makes sense to accept module arguments for the first argument position expecting a matching argument type. In order to preserve type-substitutivity, we have to make sure that no earlier position unifies with the argument type. Therefore, we define the partial function $\mathsf{fstmgu}$ that takes as arguments a constraint and a module type $\mu$. In case of success, $\mathsf{fstmgu}$ returns a substitution $\theta$ that instantiates $\mu$, together with the constraint without $\mu$.

- $\mathsf{fstmgu}([\mu_1, \ldots, \mu_{i-1}, \mu_i, \mu_{i+1}, \ldots, \mu_k] \rhd U, \mu) = ([\mu_1, \ldots, \mu_{i-1}, \mu_{i+1}, \ldots, \mu_k] \rhd U, \theta)$ if $\theta = \mathsf{mgu}(\mu, \mu_i)$ and no $\mu_j$ with $j < i$ is unifiable with $\mu$, and

- $\mathsf{fstmgu}(O \rhd U \oplus \{\mu_1\}, \mu) = (O \rhd U, \theta)$ if $\theta = \mathsf{mgu}(\mu, \mu_1)$, and no element of $O$ and no other element of $U$ is unifiable with $\mu$.

In addition, we define $\mathsf{delfst}(O \rhd U, \mu) = O' \rhd U'$ iff $\mathsf{fstmgu}(O \rhd U, \mu) = (O' \rhd U', \mathsf{id})$.

The notation $\mathsf{gen}(\Gamma, \sigma)$ is used when we want to denote the generic type scheme resulting from "generalisation" over the type variables in $\sigma$:

$$\mathsf{gen}(\Gamma, \sigma) = \forall \bar{\alpha}.\sigma \qquad \text{where } \bar{\alpha} = \mathsf{tvars}(\sigma) \setminus \mathsf{tvars}(\Gamma)$$

Substitutions form an upper semilattice with ordering $\preceq$, where $\theta_1 \preceq \theta_2$ iff $\exists \theta' \bullet \theta'\theta_1 = \theta_2$. We write $\theta_1 \sqcup \theta_2$ to denote the least upper bound of two substitutions in this semilattice.

Finally, we need a partial function $\mathsf{join}$ to join two potentially complex constraints into a single unordered constraint, if possible. Therefore, $\mathsf{join}(O_1 \rhd U_1, O_2 \rhd U_2, \Gamma)$ is defined iff all functor types in $O_1$ and $O_2$ are contained in $\Gamma$ (as representants of anonymous default instances), and then its value is the union of $U_1$ and $U_2$ and the set containing all atomic module types from $O_1$ and $O_2$.

### *9.2 Well Typed Terms*

The following type system is an extension of a standard Hindley-Milner type system. What distinguishes it is primarily the presence of the new module context $\Delta$ which keeps track of named instances. We define a term as being well typed if and only if it may be derived by the rules in Fig. 2.
Well-typedness judgements for module expressions, resp. for terms are therefore of the following shapes:

$$\Delta; \Gamma \vdash m : \mu \qquad\qquad \Delta; \Gamma \vdash t : \sigma$$

Named instances are accessed via the rule $(\mathsf{MVar})^{\mathrm{WT}}$. Functor application $(\mathsf{App}_{\#})^{\mathrm{WT}}$ is straight-forward.

The standard $\lambda$-calculus rules $(\mathsf{Var})^{\mathrm{WT}}$, $(\lambda)^{\mathrm{WT}}$, and $(\mathsf{App})^{\mathrm{WT}}$ are "mostly standard", with the following exceptions:

- the $\lambda$-bound variable needs to have an un-constrained type, since otherwise the stratification between the two type systems would be destroyed, and
- application has to join ordered constraints into an unordered constraint since ordering makes no sense here, as seen in Sect. 5.

The next four rules work on the interface between terms and module terms: Module abstraction via the $(\lambda_{\#})^{\mathrm{WT}}$ rules is similar to $\lambda$-abstraction, where polymorphic module types have to be annotated. Note that the $(\lambda_{\#})^{\mathrm{WT}}$ rules always include the module type of the bound module variable into the *ordered* constraint. In order to include polymorphic module types smoothly into our calculus, we must allow the instantiation of their type variables $\bar{\alpha}$ with arbi-

$$(\mathsf{MVar})^{\mathrm{WT}}\quad \frac{\mathtt{instance}\, i : \mu \in \Delta}{\Delta;\Gamma \vdash i : \mu} \qquad (\mathsf{App}_{\#})^{\mathrm{WT}}\quad \frac{\Delta;\Gamma \vdash i : \mu_1 :\Rightarrow \mu \qquad \Delta;\Gamma \vdash j : \mu_1}{\Delta;\Gamma \vdash (i \,\#\, j) : \mu}$$

$$(\mathsf{Var})^{\mathrm{WT}}\quad \frac{e : O \rhd U \Rightarrow \tau \in \Gamma}{\Delta;\Gamma \vdash e : O \rhd U \Rightarrow \tau} \qquad (\lambda)^{\mathrm{WT}}\quad \frac{\Delta;\Gamma, x : \tau_1 \vdash e : O_2 \rhd U_2 \Rightarrow \tau_2}{\Delta;\Gamma \vdash (\lambda x.e) : O_2 \rhd U_2 \Rightarrow \tau_1 \to \tau_2}$$

$$(\mathsf{App})^{\mathrm{WT}}\quad \frac{\begin{array}{l}\Delta;\Gamma \vdash e_1 : O_1 \rhd U_1 \Rightarrow \tau_1 \to \tau \\ \Delta;\Gamma \vdash e_2 : O_2 \rhd U_2 \Rightarrow \tau_1 \qquad U = \mathsf{join}(O_1 \rhd U_1, O_2 \rhd U_2, \Gamma)\end{array}}{\Delta;\Gamma \vdash (e_1\, e_2) : [] \rhd U \Rightarrow \tau}$$

$$(\lambda_{\#,1})^{\mathrm{WT}}\quad \frac{\Delta;\Gamma, i : \mu \vdash e : O \rhd U \Rightarrow \tau \qquad \mu \not\equiv \forall \bar{\alpha}.\mu_0}{\Delta;\Gamma \vdash (\lambda_{\#} i.e) : ([\mu] \# O) \rhd U \Rightarrow \tau}$$

$$(\lambda_{\#,2})^{\mathrm{WT}}\quad \frac{\Delta;\Gamma, i : \mu \vdash e : O \rhd U \Rightarrow \tau}{\Delta;\Gamma \vdash (\lambda_{\#} (i :: \mu).e) : ([\mu] \# O) \rhd U \Rightarrow \tau}$$

$$(\mathsf{Inst}_{\#})^{\mathrm{WT}}\quad \frac{\Delta;\Gamma \vdash i : \forall \bar{\alpha}.\mu}{\Delta;\Gamma \vdash i : [\bar{\tau}/\bar{\alpha}]\mu} \qquad (\#)^{\mathrm{WT}}\quad \frac{\Delta;\Gamma, e : O \rhd U \Rightarrow \tau, \mu \in O \vdash i : \mu}{\Delta;\Gamma \vdash (e \,\#\, i) : \mathsf{delfst}(O \rhd U, \mu) \Rightarrow \tau}$$

$$(\mathsf{Let})^{\mathrm{WT}}\quad \frac{\begin{array}{l}\Delta;\Gamma \vdash u : O_1 \rhd U_1 \Rightarrow \tau_1 \\ \Delta;\Gamma, p : \eta \vdash t : O_2 \rhd U_2 \Rightarrow \tau_2 \qquad \eta = \mathsf{gen}(\Gamma, O_1 \rhd U_1 \Rightarrow \tau_1)\end{array}}{\Delta;\Gamma \vdash (\mathtt{let}\, p = u\, \mathtt{in}\, t) : O_2 \rhd U_2 \Rightarrow \tau_2}$$

$$(\mathsf{auto})^{\mathrm{WT}}\quad \frac{\begin{array}{ll}\Delta;\Gamma \vdash e : \forall \bar{\alpha}\,.\, O \rhd U \Rightarrow \tau & U_2 = \{\kappa\langle\tau_1\rangle, \ldots, \kappa\langle\tau_n\rangle\} \\ U = U_1 \oplus U_2 \oplus \{\kappa\langle\chi\, \tau_1 \ldots \tau_n\rangle\} & \exists\theta_1, \theta_2\,.\, \theta_1(\kappa\langle\chi\, \tau_1 \ldots \tau_n\rangle) \in \theta_2 U_2 \\ & U_2 \Vdash_\Gamma \kappa\langle\chi\, \tau_1 \ldots \tau_n\rangle\end{array}}{\Delta;\Gamma \vdash e : O \rhd U_1 \oplus U_2 \Rightarrow \tau}$$

$$(\mathsf{force})^{\mathrm{WT}}\quad \frac{\Delta;\Gamma \vdash e : O \rhd (U_1 \oplus U_2 \oplus U_3) \Rightarrow \tau \quad U_1 \cup U_3 \Vdash_\Gamma U_2 \quad U_3 = \{\mu_1, \ldots, \mu_k\}}{\Delta;\Gamma \vdash (e :: (O \# [\mu_1, \ldots, \mu_k]) \rhd U_1 \Rightarrow \tau) : (O \# [\mu_1, \ldots, \mu_k]) \rhd U_1 \Rightarrow \tau}$$

Fig. 2. Well-typedness rules

trary types $\bar{\tau}$ via the $(\mathsf{Inst}_{\#})^{\mathrm{WT}}$ rule. Explicit "dictionary application" $(\#)^{\mathrm{WT}}$ is, as discussed above, not restricted to arguments matching the first argument type of the constraint.

(Non-recursive) `let` bindings are treated as usual; there is no need to add the constraints of $u$ to those of $t$, since they are already taken care of via the presence of $p$.

The last group of rules corresponds to constraint change and constraint reduction. "Automatic" reduction only takes place in ambiguous *unordered* constraints. Note that the rule $(\mathsf{auto})^{\mathrm{WT}}$ only applies if and only if the type variables are polymorphic.

Automatic constraint reduction might be considered as problematic since we may derive *several* different types for *one* term — as Haskell 98 does in a number of cases. As an example consider `f :: {Eq a, Eq [a]} ⇒ [a] → [a]` which has also the type `{Eq a} ⇒ [a] → [a]`. For this reason, type inference can only be complete when the unordered constraint of a qualified type is irreducible. In addition, the rule $(\mathsf{auto})^{\mathrm{WT}}$ may only be applied (repeatedly)

as last steps of the derivation of the type of $u$ in the $(\mathsf{Let})^{\mathrm{WT}}$ rule (that is, top-level in binding groups). Reduction via $(\mathsf{auto})^{\mathrm{WT}}$ is obviously normalising because it strictly reduces the size of the unordered constraint, and because of transitivity of entailment.

Less problematic is constraint change through explicit type annotation, which is denoted by "::" as usual in Haskell. There are two effects possible here. The first is elimination of atomic module types from the unordered constraint if they can be entailed from anonymous instances in $\Gamma$ and the remaining unordered constraint. The second allows to move atomic module types from the unordered into the ordered constraint.

### *9.3 Type Inference Algorithm*

An only slightly more involved set of rules defines a type inference algorithm for our system. In comparison with well-typedness judgements, type inference judgements carry an additional substitution; this substitution and the inferred type are the output of the algorithm, while the two contexts $\Delta$ and $\Gamma$ and the (module) term are its input:

$$\overset{\uparrow}{\theta};\overset{\downarrow}{\Delta};\overset{\downarrow}{\Gamma}\vdash\overset{\downarrow}{i}:\overset{\uparrow}{\mu} \qquad\qquad \overset{\uparrow}{\theta};\overset{\downarrow}{\Delta};\overset{\downarrow}{\Gamma}\vdash\overset{\downarrow}{t}:\overset{\uparrow}{\sigma}$$

We understand a type inference judgement to be a *result* of the algorithm if no further derivation is possible — the rule "(auto)" would otherwise introduce ambiguities. It is understood that type inference is impossible if any of the operations used in a rule invocation is undefined. The rules of the type inference algorithm as shown in Fig. 3 correspond to the rules in the preceding section.

The relationship between the type system and the inference algorithm is made precise by the following two theorems.

**Theorem 9.1 (Soundness)** *For every (module) term judgement resulting from the type inference algorithm, a corresponding well-typedness judgement may be derived:*

$$\theta;\Delta;\Gamma\ \vdash\ i:\mu\ \Longrightarrow\ \theta\Delta;\theta\Gamma\ \vdash\ i:\mu$$
$$\theta;\Delta;\Gamma\ \vdash\ t:O\rhd U\Rightarrow\tau\ \Longrightarrow\ \theta\Delta;\theta\Gamma\ \vdash\ t:O\rhd U\Rightarrow\tau$$

**Theorem 9.2 (Completeness)** *For every well-typedness judgement not involving rank-2 module types, a corresponding judgement may be derived via the type inference algorithm:*

$$\theta\Delta;\theta\Gamma\ \vdash\ i:\mu\ \Longrightarrow\ \exists\theta_1,\theta_2,\mu_1\,.\ \theta_1;\Delta;\Gamma\ \vdash\ i:\mu_1\ \wedge\ \theta\Delta=\theta_2\theta_1\Delta\ \wedge\ \theta\Gamma=\theta_2\theta_1\Gamma\ \wedge\ \mu=\theta_2\mu_1$$

$$(\mathsf{MVar})^{\mathrm{TI}}\quad \frac{\texttt{instance}\, i : \mu \in \Delta}{\mathtt{id}; \Delta; \Gamma \vdash i : \mu} \qquad (\mathsf{Inst}_{\#})^{\mathrm{TI}}\quad \frac{\theta; \Delta; \Gamma \vdash i : \forall \overline{\alpha}.\mu}{\theta; \Delta; \Gamma \vdash i : [\overline{\alpha'}/\overline{\alpha}]\mu}$$

$$(\mathsf{App}_{\#})^{\mathrm{TI}}\quad \frac{\theta_1; \Delta; \Gamma \vdash i : \mu_1 :\Rightarrow \mu \qquad \theta_2; \Delta; \Gamma \vdash j : \mu_2 \qquad \theta = \theta_1 \sqcup \theta_2 \sqcup \mathsf{mgu}(\mu_1, \mu_2)}{\theta; \Delta; \Gamma \vdash (i \,\#\, j) : \theta\mu}$$

$$(\mathsf{Var})^{\mathrm{TI}}\quad \frac{e : O \rhd U \Rightarrow \tau \in \Gamma}{\mathtt{id}; \Delta; \Gamma \vdash e : O \rhd U \Rightarrow \tau} \qquad (\lambda)^{\mathrm{TI}}\quad \frac{\theta; \Delta; \Gamma, x : \tau_1 \vdash e : O_2 \rhd U_2 \Rightarrow \tau_2}{\theta; \Delta; \Gamma \vdash (\lambda x.e) : O_2 \rhd U_2 \Rightarrow \tau_1 \to \tau_2}$$

$$(\mathsf{App})^{\mathrm{TI}}\quad \frac{\begin{array}{ll}\theta_1; \Delta; \Gamma \vdash e_1 : O_1 \rhd U_1 \Rightarrow \tau_1 \to \tau & \theta = \theta_1 \sqcup \theta_2 \sqcup \mathsf{mgu}(\tau_1, \tau_2) \\ \theta_2; \Delta; \Gamma \vdash e_2 : O_2 \rhd U_2 \Rightarrow \tau_2 & U = \mathsf{join}(O_1 \rhd U_1, O_2 \rhd U_2, \Gamma)\end{array}}{\theta; \Delta; \Gamma \vdash (e_1\ e_2) : \theta([] \rhd U \Rightarrow \tau)}$$

$$(\lambda_{\#,1})^{\mathrm{TI}}\quad \frac{\theta; \Delta; \Gamma, i : \mu \vdash e : O \rhd U \Rightarrow \tau \qquad \mu \not\equiv \forall \bar{\alpha}.\mu_0}{\theta; \Delta; \Gamma \vdash (\lambda_{\#} i.e) : ([\mu] \# O) \rhd U \Rightarrow \tau}$$

$$(\lambda_{\#,2})^{\mathrm{TI}}\quad \frac{\theta; \Delta; \Gamma, i : \mu \vdash e : O \rhd U \Rightarrow \tau}{\theta; \Delta; \Gamma \vdash (\lambda_{\#}(i::\mu).e) : ([\mu] \# O) \rhd U \Rightarrow \tau}$$

$$(\#)^{\mathrm{TI}}\quad \frac{\begin{array}{ll}\theta_1; \Delta; \Gamma \vdash e : O_1 \rhd U_1 \Rightarrow \tau & (O \rhd U, \theta_3) = \mathsf{fstmgu}(O_1 \rhd U_1, \mu) \\ \theta_2; \Delta; \Gamma \vdash i : \mu & \theta = \theta_1 \sqcup \theta_2 \sqcup \theta_3\end{array}}{\theta; \Delta; \Gamma \vdash (e \,\#\, i) : \theta(O \rhd U \Rightarrow \tau)}$$

$$(\mathsf{Let})^{\mathrm{TI}}\quad \frac{\begin{array}{ll}\theta_1; \Delta; \Gamma \vdash u : O_1 \rhd U_1 \Rightarrow \tau_1 & \eta = \mathsf{gen}(\theta_1\Gamma, O_1 \rhd U_1 \Rightarrow \tau_1) \\ \theta_2; \Delta; \Gamma, p : \eta \vdash t : O_2 \rhd U_2 \Rightarrow \tau_2 & \theta = \theta_1 \sqcup \theta_2\end{array}}{\theta; \Delta; \Gamma \vdash (\texttt{let}\ p = u\ \texttt{in}\ t) : \theta(O_2 \rhd U_2 \Rightarrow \tau_2)}$$

$$(\mathsf{auto})^{\mathrm{TI}}\quad \frac{\begin{array}{ll}\theta; \Delta; \Gamma \vdash e : \forall \bar{\alpha}\ .\ O \rhd U \Rightarrow \tau & U_2 = \{\kappa\langle\tau_1\rangle, \ldots, \kappa\langle\tau_n\rangle\} \\ & \exists \theta_1, \theta_2\ .\ \theta_1(\kappa\langle\chi\ \tau_1 \ldots \tau_n\rangle) \in \theta_2 U_2 \\ U = U_1 \oplus U_2 \oplus \{\kappa\langle\chi\ \tau_1 \ldots \tau_n\rangle\} & U_2 \Vdash_\Gamma \kappa\langle\chi\ \tau_1 \ldots \tau_n\rangle\end{array}}{\theta; \Delta; \Gamma \vdash e : O \rhd U_1 \oplus U_2 \Rightarrow \tau}$$

$$(\mathsf{force})^{\mathrm{TI}}\quad \frac{\theta; \Delta; \Gamma \vdash e : O \rhd (U_1 \oplus U_2 \oplus U_3) \Rightarrow \tau \qquad U_1 \cup U_3 \Vdash_\Gamma U_2 \qquad U_3 = \{\mu_1, \ldots, \mu_k\}}{\theta; \Delta; \Gamma \vdash (e::(O \# [\mu_1, \ldots, \mu_k]) \rhd U_1 \Rightarrow \tau) : (O \# [\mu_1, \ldots, \mu_k]) \rhd U_1 \Rightarrow \tau}$$

Fig. 3. Type inference rules

$$\theta\Delta; \theta\Gamma \vdash t : \sigma \implies \exists \theta_1, \theta_2, \sigma_1\ .\ \theta_1; \Delta; \Gamma \vdash t : \sigma_1 \wedge \theta\Delta = \theta_2\theta_1\Delta \wedge \theta\Gamma = \theta_2\theta_1\Gamma \wedge \sigma = \theta_2\sigma_1$$

*If furthermore the unordered constraint of $\sigma$ is irreducible wrt.* $(\mathsf{auto})^{\mathrm{WT}}$*, then the unordered constraint of $\sigma_1$ is irreducible wrt.* $(\mathsf{auto})^{\mathrm{TI}}$*, too.*

The proofs of both theorems proceed by induction on the structure of derivations. Note that Theorem 2 implies $\mu_1 \preceq \mu$ and $\sigma_1 \preceq \sigma$. Apart from that, possible types $\sigma$ can only differ in unordered constraints of $t$. Since this holds for all possible module types $\mu$ of $i$ and all possible types $\tau$ of $t$, the algorithm yields the principal types for $i$ and $t$.

$\beta$-reduction is defined as usual, and it interacts sensibly with type inference. Since ordered constraints have to be joined in expression application,

we obtain a weaker variant of subject reduction:

**Definition 9.3** A qualified type $\sigma_1 = O_1 \rhd U_1 \Rightarrow \tau_1$ is called *weaker than* a qualified type $\sigma_2 = O_2 \rhd U_2 \Rightarrow \tau_2$ under context $\Gamma$, iff there are a substitution $\theta$ and three subsequences $O_\mathrm{o}$, $O_\mathrm{u}$, and $O_\mathrm{e}$ of $O_1$, such that $O_1$ is an interleaving of $O_\mathrm{o}$, $O_\mathrm{u}$, and $O_\mathrm{e}$, and

$$\theta\tau_1 = \tau_2 \ , \qquad \theta O_\mathrm{o} = O_2 \ , \qquad \theta(|O_\mathrm{u}|) \cup \theta U_1 = U_2 \ , \qquad U_2 \Vdash_\Gamma \theta O_\mathrm{e} \ .$$

**Theorem 9.4 (Subject reduction)** *If $\theta_1; \Delta; \Gamma \vdash t_1 : \sigma_1$ holds by type inference and the term $t_1$ $\beta$-reduces to another term $t_2$, then there are $\theta_2$ and $\sigma_2$ with $\theta_2 \preceq \theta_1$ such that $\sigma_2$ is weaker than $\sigma_1$ and the type inference judgement $\theta_2; \Delta; \Gamma \vdash t_2 : \sigma_2$ holds.*

As an aside, the above definition of the *weaker than* relation can also be used to help remedy the fact that with lazier context reduction for named instances — in the same way as with implicit parameters — user-supplied type signatures frequently hinder adaptability of Haskell code. We propose *lax type signatures*, which are to be understood as asserting a lower bound (with respect to the above *weaker than* relation) on the type of the defined entity. This implies that

```
f ::< forall a . (Eq a) => a -> b
```

would allow any of the following:

```
f :: (Eq a        ) => a -> b
f :: (Eq a        ) => a -> String
f :: (Ord a       ) => a -> b   -- modulo entailment equivalence
f :: (Eq a, Read b) => a -> b
```

Forcing the user to make the `forall` explicit follows the guideline that these lax type signatures enable users to *enforce* structure for inferred types by explicitly providing it. Writing

```
f ::< exists b . (Eq a) => a -> b
```

instead would rather have a taste of explicitly specifying degrees of freedom — but it does not specify degrees of freedom in the context. For the latter, an appropriate syntax seems to be much harder to find. Therefore, we find the `forall` variant more natural.

## 10 Type Checking Prototype

In order to be able to experiment with our design of module type constraints, we developed a prototype type checker. It extends the Haskell 98 type checker "Typing Haskell in Haskell" of Mark Jones [7] with module type constraints and performs type inference in the manner described in the preceding sections. We mainly extended the handling of constraints and introduced user defined

type classes and (named) instances as well as type checking inside them. Furthermore, some experimental features, as for example subtyping (see Sect. 7), have been implemented as well as extensions that are not covered in detail by this paper (see below).

Nevertheless, the basic features of "Typing Haskell in Haskell" have not been compromised, and our prototype continues to accept the original examples provided by Mark Jones, and to infer their principal types. This serves to increase our confidence that our extension is essentially conservative. The prototype is available on the WWW[7]. Further developments will also be published at this address.

## 11 Conclusion and Outlook

The desire to enable later binding of Haskell type class members together with the known analogies between the Haskell type class system and ML module systems lead us to design an extension of Haskell comprising named instances, explicit instance supply, instance functors and module types. This extension incorporates a subset of Jones' parameterised signatures, but in a way that preserves compatibility with the Haskell 98 type class system and refrains form giving structures first-class citizenship, so in this respect stays closer to the separate module language of OCaml.

In addition, the desire to remain compatible to conventional Haskell leads also to a new feature: type class constraints now have to be considered as module type constraints and thus lead to a new class of qualified types which, as we have seen, can be considered as closely related to the implicit parameters of Lewis *et al.* [13].

Apart from this, our work is of course closely related to work on first class module systems, most notably that by Russo [17,18]. We believe that extending our proposal towards first-class module system capabilities is a natural step and will most probably be necessary for many uses. Nevertheless we have intentionally left out such considerations. We consider that already our relatively small extensions have quite far-reaching consequences both conceptually and from the point of view of expressive power and (re)usability, so that it makes sense to study them in relative isolation.

The relation with work on multi-parameter classes [16] turns out to be relatively weak — module type constraints mainly offer a way to obtain many of the benefits of multi-parameter classes, short-circuiting a lot of the problems research has been centring around. We think that our extension may in fact allow more exploration of library design centred around multi-parameter classes — it would certainly have been useful in the design of the relation algebra toolkit RATH [10].

The design we presented should not be considered as an attempt at a

[7] URL: http://ist.unibw-muenchen.de/Haskell/NamedInstances/

conclusive definition of named instance features. Instead, we tried to present an apparently reasonable subset of non-trivial features amidst some discussion of the decisions involved. Due to lack of space we have concentrated on the basic idea and left out further extensions that are already implemented in our prototype and seem to prove their usefulness. A detailed discussion will be contained in the second author's diploma thesis [19].

Among these extensions are notations to directly access the anonymous *default* instances (via the special instance name `Default`) and anonymous *derived* instances (via the special instance name `Derived`). The possibility to access these instances directly makes it possible to use them as explicit functor arguments. We consider furthermore the possibility to import instances "`as Default`".

A straightforward step towards first-class instances is to make their parameterisation via module variables possible as well as their *local* definition via `let` or `where` expressions — the example in Sect. 8 shows a possible notation.

We have not discussed implementation at all — this is mainly because it does not seem to be a serious problem. Since current Haskell implementations of type classes rely on dictionary translation, implementing our features mostly amounts to extending the parser and connecting it to existing features of the implementation, which just had not been directly accessible via the user-level language before. We intend to address this in the near future.

In summary, following through the consequences of explaining the Haskell type system in terms of the ML module system is not quite as trivial as it might seem at first sight. However, it enforces useful conceptual clarifications and then gives rise to a natural extension that, in our opinion, will also be of great value to users of Haskell.

## References

[1] Ralf Hinze. A generic programming extension for Haskell. In E. Meijer, editor, *Proceedings of the 3rd Haskell Workshop, Paris, France*, September 1999. Tech. Report UU-CS-1999-28.

[2] Ralf Hinze. A new approach to generic functional programming. Technical Report IAI-TR-99-9, Institut für Informatik III, Universität Bonn, 1999.

[3] Ralf Hinze and Simon Peyton-Jones. Derivable type classes. In *Haskell Workshop 2000*, September 2000.

[4] M. P. Jones and J.C. Peterson. *Hugs 98 — A functional programming system based on Haskell 98 — User Manual*, 1999.

[5] Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*, pages 97–136. Springer, 1995.

[6] Mark P. Jones. Using parameterized signatures to express modular strucure. In *23rd POPL*, pages 68–78. acm press, 1996.

[7] Mark P. Jones. Typing Haskell in Haskell. In *Third International Haskell Workshop 1999*, 1999.

[8] Mark P. Jones. Type classes with functional dependencies. In Smolka [20], pages 230–244.

[9] S. Peyton Jones and P. Wadler. A static semantics for Haskell. Technical Report G12 8 QQ, University of Glasgow, 1992.

[10] Wolfram Kahl and Gunther Schmidt. Exploring (finite) Relation Algebras using Tools written in Haskell. Technical Report 2000-02, Fakultät für Informatik, Universität der Bundeswehr München, October 2000.

[11] Xavier Leroy. Manifest types, modules, and separate compilation. In *21th POPL*, pages 109–122. acm press, 1994.

[12] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *22nd POPL*. acm press, 1995.

[13] J. Lewis, M. Shields, E. Meijer, and J. Launchbury. Implicit parameters: Dynamic scoping with static types. In *27th POPL*. acm press, 2000.

[14] David B. MacQueen and Mads Tofte. A semantics for higher-order functors. In Donald Sanella, editor, *ESOP '94*, volume 788 of *LNCS*, pages 409–424. Springer, 1994.

[15] Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, 1991.

[16] Simon L. Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space, 1997.

[17] Claudio V. Russo. *Types For Modules*. LFCS thesis ECS-LFCS-98-389, University of Edinburgh, 1998.

[18] Claudio V. Russo. First-class structures for Standard ML. In Smolka [20], pages 336–350.

[19] Jan Scheffczyk. *Named Instances with Class in Haskell*. Diploma thesis UniBwM-ID 04/01, Fakultät für Informatik, Universität der Bundeswehr München, 2001. See also:
http://ist.unibw-muenchen.de/Haskell/NamedInstances/.

[20] G. Smolka, editor. *ESOP 2000, 10th European Symposium on Programming*, volume 1782 of *LNCS*. Springer, March 2000.