Flexible Instances

Brief Explanation

Especially with MultiParamTypeClasses, we would like to write instances like

```
instance MArray (STArray s) e (ST s)
```

However without some restrictions constraint checking is undecidable.

This page lists alternative proposals for liberalizing the form of instances while retaining sufficient restrictions to guarantee termination. Such restrictions are necessarily conservative: there will always be programs that are clearly safe but still rejected. Restrictions on the form of instances also restrict the forms of datatype declarations that can use `deriving` clause (see Context reduction errors in the Haskell 98 Report).

Note that if FunctionalDependencies are present, additional restrictions are required.

See UndecidableInstances for an alternative strategy using dynamic constraints on context reduction.

If one can write instances like

```
instance C [Bool] where ...
instance C [Char] where ...
```

then assertions like `C [a]` can be neither reduced nor rejected, so FlexibleContexts are also needed.

References

- Instance declarations in the Haskell 98 Report
- Type classes: exploring the design space by Simon Peyton Jones, Mark Jones and Erik Meijer, Haskell Workshop 1997.
- Undecidable instances in the GHC 6.4 User's Guide.
- Relaxed rules for instance declarations in the GHC 6.5 User's Guide.

Tickets

#32 add FlexibleInstances

Local termination conditions

The idea here is to impose restrictions on the form of each instance in isolation, such that context reduction will be guaranteed to terminate.

Haskell 98

- an instance head must have the form C (T u_1 ... u_k), where T is a type constructor defined by a `data` or `newtype` declaration (see TypeSynonymInstances) and the u_i are distinct type variables, and
- each assertion in the context must have the form C' v, where v is one of the u_i.

GHC 6.4 and earlier

- at least one of the type arguments of the instance head must be a non-variable type, and
- each assertion in the context must have the form C v_1 ... v_n, where the v_i are distinct type variables.

The distinctness requirement prohibits non-terminating instances like

```
instance C b b => C (Maybe a) b
```

(e.g. `C (Maybe Int) (Maybe Int)` reduces to itself.)

GHC 6.5

Each assertion in the context must satisfy

- no variable has more occurrences in the assertion than in the head, and
- the assertion has fewer constructors and variables (taken together and counting repetitions) than the head.

(These conditions ensure that under any ground substitution, the assertion contains fewer constructors than the head.)

This rule allows instances accepted by the previous rule and more, including

```
instance C a
instance Show (s a) => Show (Sized s a)
instance (C1 a, C2 b) => C a b
instance C1 Int a => C2 Bool [a]
instance C1 Int a => C2 [a] b
instance C a a => C [a] [a]
```

It also allows derived instances like

```
data Sized s a = S Int (s a)
    deriving (Show)
```

because the derived instance (above) has the required form.

Note: instances generated by NewtypeDeriving often do not have the required form, and can lead to non-terminating reductions.

Non-local termination conditions

No local criterion can accept a definition like

```
instance (C1 a, C2 a, C3 a) => C a
```

because termination depends on other instances in the program. However this is clearly safe if none of the instances for `C1`, `C2` or `C3` contain a direct or indirect constraint using `C`.

The proposal is that one of the above local restrictions be used, but only on cycles of instances.

This is more complex than a local criterion: the instance the compiler complains about might not be the one the programmer just added (creating a cycle), or even in the same module.

Last modified on Mar 31, 2006 11:08:11 PM