$$
\begin{array}{c}
CE, \{u:\alpha\}, h \overset{context}{\vdash} cx : \theta \\
IE'_{sup} = vs \tilde{:} \theta \\
IE_{sup} = \forall\alpha.\Gamma\alpha \tilde{\Rightarrow} IE'_{sup} \\
\langle CE, TE \cup \{u:\alpha\}, DE\rangle \overset{sigs}{\vdash} sigs : VE_{sigs} \\
i \in [1,n] : GE, IE \oplus \{v_d : \Gamma\alpha\}, VE \overset{method}{\vdash} bind_i \rightsquigarrow \mathtt{fbind}_i : VE_i \\
VE_1 \oplus \ldots \oplus VE_n \subseteq VE_{sigs} \\
\alpha = u^{\kappa} \\
\Gamma = B^{\kappa} \\
CE, IE \oplus IE_{sup}, CE, VE, cs \overset{dinstDecls}{\vdash} didecls \rightsquigarrow \mathtt{dibinds} : IE_{di} \\
CE' = \{B : \langle \Gamma, h, v_{def}, \alpha, IE'_{sup}\rangle\} \\
VE' = \forall\alpha.\Gamma\alpha \tilde{\Rightarrow}_c VE_{sigs} \\
GE = \langle CE, TE, DE\rangle \\
J_{\mathtt{dict}}, \mathtt{vs}, \mathtt{v_{def}} \text{ fresh}
\end{array}
$$

$$
\begin{array}{l}
GE, IE, VE \overset{ctDecl}{\vdash} \quad \mathtt{class}\, cx \Rightarrow B\, u\, \mathtt{where} \\
\qquad sigs; \\
\qquad bind_1; \ldots; bind_n; \\
\qquad didecls; \\
\qquad \rightsquigarrow \left\{ \begin{array}{l} \mathtt{data}\, \hat{\Gamma} = \mathtt{J_{dict}}\{\widehat{IE'_{sup}}, \widehat{VE_{sigs}}, \}; \\ v_{def} : (\forall\alpha.\hat{\Gamma}\alpha \to \hat{\Gamma}\alpha) \\ \quad = \Lambda\alpha.\lambda \mathtt{v_d} : (\hat{\Gamma}\alpha).\mathtt{J_{dict}}\alpha\{\mathtt{fbind}_1, \ldots \mathtt{fbind}_n\}; \\ \mathtt{dibinds} \end{array} \right\} \\
\qquad : \langle CE', \{\}, \{\}, IE_{sup} \oplus IE_{di}, VE'\rangle
\end{array}
$$

Figure 4.2: New semantics of class declarations

Rule 3 Select an instance $I_i\, a$ such that for any other instance $I_n\, a$ holds

$$I_n\, a \prec I_i\, a$$

The motivation for the Rule 1 is to preserve backward compatible behavior, i. e. to select existing instances, and to enable user to provide his own implementation of the instance. We propose to issue a warning when a proper instance is selected over default instance.

The motivation for the Rule 2 is to decide between two instances that are not superclass of each other. In this case we omit them both and try to select their common ancestor. We expect the ancestor to be general–or abstract–enough to provide sufficient default instance.

In the Rule 3 we have possibly several instances that are all either a superclass or a subclass of each other. The ratio behind the rule is to select the instance which is the least abstract, i. e. the most specific. We expect this instance to provide possibly better implementation regarding the performance as it has the most specific problem related information.

4.3.2 Semantics of the Extension

In this section we provide static semantics of our extension. Default instances require a change on the `ctDecl` inference rule described in the Figure 4.2. This

$$\frac{\begin{array}{c} i \in [1,n] : CE \overset{dinstCtx}{\vdash} dinstDecl_i : \{\Gamma_i\ \tau_i\} \\ cx' = cx^+ \setminus \{\Gamma_i\ \tau_i \ldots \Gamma_n\ \tau_n\}^+ \\ i \in [1,n] : GE, IE, VE, cx' \overset{dinstDecl}{\vdash} dinstDecl_i \rightsquigarrow \texttt{binds}_i : IE_i \end{array}}{GE, IE, VE \overset{dinstDecls}{\vdash} \left\{ \begin{array}{l} dinstDecl_1; \\ \ldots; \\ dinstDecl_n; \end{array} \right\} \rightsquigarrow \left\{ \begin{array}{l} \texttt{binds}_1; \\ \ldots; \\ \texttt{binds}_n; \end{array} \right\} : IE_1 \oplus \ldots \oplus IE_n}$$

$$\frac{\begin{array}{c} T : \chi \in TE \\ i \in [1,n] : \alpha_i = u_i^{\kappa_i} \\ C : \langle \Gamma, h, x_{def}, \alpha, IE_{sup} \rangle \in CE \\ CE, \{u_1 : \alpha_1\} \oplus \ldots \oplus \{u_n : \alpha_n\}, \_ \overset{context}{\vdash} cx : \theta \\ i \in [1,m] : GE, IE \oplus vs\tilde{:}\theta, VE \overset{method}{\vdash} bind_i \rightsquigarrow \texttt{fbind}_i : VE_i \\ VE_{ops}[\chi\ \alpha_1 \ldots \alpha_n/\alpha] = VE_1 \oplus \ldots \oplus VE_m \\ (\forall \alpha . \Gamma\ \alpha \Rightarrow_c VE_{ops}) \subseteq VE(x_1, \ldots, x_n)\tilde{:}\theta sup = IE_{sup} \\ IE \oplus \texttt{vs}\tilde{:}\theta \overset{dict}{\vdash} (e_1, \ldots, e_n) : \theta_{sup}[\chi\ \alpha_1 \ldots \alpha_n/\alpha] \\ GE = \langle CE, TE, DE \rangle \\ IE_{inst} = \{v_{dict} : \forall \alpha_1 \ldots \alpha_n . \theta \Rightarrow \Gamma(\chi\ \alpha_1 \ldots \alpha_n/\alpha) \\ \texttt{vs}, \texttt{v}_{dict} \text{ fresh} \end{array}}{\begin{array}{l} GE, IE, VE, cx \overset{dinstDecl}{\vdash} \texttt{default instance}\ C\ (T\ u_1 \ldots u_k)\texttt{where} \\ \quad bind_1; \ldots; bind_m \\ \rightsquigarrow \left\{ \begin{array}{l} \texttt{v}_{dict}; \forall \alpha_1, \ldots, \alpha_k . \hat{\theta} \rightarrow \hat{\Gamma}\ (\chi\ \alpha_1 \ldots \alpha_k) \\ \quad = \Lambda \alpha_1, \ldots, \alpha_k . \lambda \texttt{vs}\hat{:}\theta . \\ \quad \texttt{let rec}\ v_d : \hat{\Gamma}\ (\chi\ \alpha_1 \ldots \alpha_k) \\ \quad\quad = (x_d ef\ (\chi\ \alpha_1 \ldots \alpha_k) v_d)\{ \\ \quad\quad\quad x_1 = e_1; \ldots; x_n = e_n; \\ \quad\quad\quad \texttt{fbind}_1; \ldots \texttt{fbind}_n \\ \quad\quad \}\ \texttt{in}\ v_d \end{array} \right\} \\ \quad : IE_{inst} \end{array}}$$

Figure 4.3: Semantics of default instance declarations

change requires new inference rule `dinstDecls`.

The new inference rules `dinstDecls` and `dinstDecl` transform default instances into the same representation as original rules transforms ordinary instances. These rule are also modeled in a similar way to `instDecls` and `instDecl` respectively.

The new inference rule `dinstCtx` gathers annotated class names from a default instnace declaration.

4.4 Class Aliases

We propose to add a new syntax construct into the class definition as described in the Section 1.4.3. Programmer may provide a *class alias* for several different classes, e. g.:

```
classalias  (Read a, Show a) ⇒ Textual a
```

$$\frac{C : \langle \Gamma, \_, \_, \alpha, \_ \rangle \in CE}{CE \overset{dinstCtx}{\vdash} \texttt{default instance}\ C\ (T\ u_1 \ldots u_k)\ \texttt{where}\ bind_1; \ldots; bind_m : \Gamma\ \tau}$$

Figure 4.4: Semantics of default instance contexts

topdecl	→	`classalias` [*scontext* `=>`] *tycls tyvar*
scontext	→	*simpleclass*
	|	`(` $simpleclass_1$ `,` ... `,` $simpleclass_n$ `)` $(n \geq 0)$
simpleclass	→	*qtycls tyvar*

Figure 4.5: Class alias declarations

The class alias may be instantiated by usual instance declaration. Compiler generates separate instances for `Read` and `Show` classes and distributes the class method accordingly.

4.4.1 Syntax

Class aliases are new top level declaration. We present changes in formal syntax in the Figure 4.5.

A class alias declaration has a general form:

```
classalias cx ⇒ D u
```

This introduces new *class alias* of the classes in the context *cx*. The class in context are required to form an acyclic directed graph and must be all different. The aliased classes may not contain methods with the same name.

The declaration of *class alias* may contain binding only for the class methods of aliased classes. If no binding is given for some method default method in the class declaration is used. It there is no such method the method of the instance is bound to *undefined*.

4.4.2 Relation to The Superclass Default Instances

Note that the class aliases are not necessary for use of superclass default instances. However, class aliases make some changes in class hierarchy easier and more direct. Assume following classes in a library code

```
class C a where
        ...

class D a where
        method :: a
```

and instance in a client code:

```
instance D MyData where
        method = ...
```