Constraints for Type Class Extensions

Master's Thesis Defense

Gerrit van den Geest

Supervisor: prof. dr. S. D. Swierstra
Daily supervisors: dr. B. J. Heeren and dr. A. Dijkstra

Center for Software Technology
Universiteit Utrecht

February 22, 2007

Type classes

$$
\begin{array}{l}
\textbf{class } \mathit{Eq}\ a \textbf{ where} \\
\quad \mathit{eq} :: a \rightarrow a \rightarrow \mathit{Bool}
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance } \mathit{Eq}\ \mathit{Char} \textbf{ where} \\
\quad \mathit{eq} = \mathit{primEqChar}
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance } \mathit{Eq}\ a \Rightarrow \mathit{Eq}\ [a] \textbf{ where} \\
\quad \begin{array}{llll}
\mathit{eq} & [\,] & [\,] & = \mathit{True} \\
\mathit{eq} & (x : \mathit{xs}) & (y : \mathit{ys}) & = \mathit{eq}\ x\ y \wedge \mathit{eq}\ \mathit{xs}\ \mathit{ys} \\
\mathit{eq} & \_ & \_ & = \mathit{False}
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\mathit{elem} :: \mathit{Eq}\ a \Rightarrow a \rightarrow [a] \rightarrow \mathit{Bool} \\
\mathit{elem}\ x\ [\,] \qquad\quad = \mathit{False} \\
\mathit{elem}\ x\ (y : \mathit{ys}) = \mathit{eq}\ x\ y \vee \mathit{elem}\ x\ \mathit{ys}
\end{array}
$$

$$
\mathit{main} = \mathit{elem}\ \texttt{"Hello"}\ [\texttt{"Hello"}, \texttt{"World"}]
$$

Type classes

$$eq :: Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool$$

$$\begin{array}{l}
\textbf{class}\ Eq\ a\ \textbf{where} \\
\quad eq :: a \rightarrow a \rightarrow Bool
\end{array}$$

$$\begin{array}{l}
\textbf{instance}\ Eq\ Char\ \textbf{where} \\
\quad eq = primEqChar
\end{array}$$

$$\begin{array}{llll}
\multicolumn{4}{l}{\textbf{instance}\ Eq\ a \Rightarrow Eq\ [a]\ \textbf{where}} \\
\quad eq\ [\,] & [\,] & = & True \\
\quad eq\ (x : xs) & (y : ys) & = & eq\ x\ y \wedge eq\ xs\ ys \\
\quad eq\ _ & _ & = & False
\end{array}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{elem :: Eq\ a \Rightarrow a \rightarrow [a] \rightarrow Bool} \\
elem\ x\ [\,] & = & False \\
elem\ x\ (y : ys) & = & eq\ x\ y \vee elem\ x\ ys
\end{array}$$

$$main = elem\ \texttt{"Hello"}\ [\texttt{"Hello"}, \texttt{"World"}]$$

Type classes

$$
\begin{array}{l}
\textbf{class } Eq\ a \textbf{ where} \\
\quad eq :: a \rightarrow a \rightarrow Bool
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance } Eq\ Char \textbf{ where} \\
\quad eq = primEqChar
\end{array}
$$

$$
\begin{array}{llll}
\multicolumn{4}{l}{\textbf{instance } Eq\ a \Rightarrow Eq\ [a] \textbf{ where}} \\
\quad eq\ [\,] & [\,] & = & True \\
\quad eq\ (x : xs) & (y : ys) & = & eq\ x\ y \wedge eq\ xs\ ys \\
\quad eq\ \_ & \_ & = & False
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{elem :: Eq\ a \Rightarrow a \rightarrow [a] \rightarrow Bool} \\
elem\ x\ [\,] & = & False \\
elem\ x\ (y : ys) & = & eq\ x\ y \vee elem\ x\ ys
\end{array}
$$

$$main = elem\ \texttt{"Hello"}\ [\texttt{"Hello"}, \texttt{"World"}]$$

$$eq :: Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool$$

overloading $\not\equiv$ polymorphism

$$length :: [a] \rightarrow Int$$

Type classes

$$\mathbf{class}\ Eq\ a\ \mathbf{where}$$
$$\quad eq :: a \to a \to Bool$$

$$\mathbf{instance}\ Eq\ Char\ \mathbf{where}$$
$$\quad eq = primEqChar$$

$$\mathbf{instance}\ Eq\ a \Rightarrow Eq\ [a]\ \mathbf{where}$$
$$\begin{aligned}
&eq\ []\ &&[] &&= True\\
&eq\ (x : xs)\ &&(y : ys) &&= eq\ x\ y \land eq\ xs\ ys\\
&eq\ _\ &&_ &&= False
\end{aligned}$$

$$\begin{aligned}
&elem :: Eq\ a \Rightarrow a \to [a] \to Bool\\
&elem\ x\ []\ = False\\
&elem\ x\ (y : ys) = eq\ x\ y \lor elem\ x\ ys
\end{aligned}$$

$$main = elem\ \texttt{"Hello"}\ [\texttt{"Hello"}, \texttt{"World"}]$$

$$eq :: Eq\ a \Rightarrow a \to a \to Bool$$

$$Eq = \{\, Char, [\,Char\,], [[\,Char\,]], \ldots \}$$

Translation

$$\textbf{data}\ Eq\ a = Eq\{eq :: a \to a \to Bool\}$$

$$eqChar :: Eq\ Char$$
$$eqChar = Eq\{eq = primEqChar\}$$

$$eqList :: Eq\ a \to Eq\ [a]$$
$$eqList\ d = Eq\{eq = f\}$$
$$\quad \textbf{where}\ f\ [\,]\ [\,] = True$$
$$\quad\quad f\ (x : xs)\ (y : ys) = eq\ d\ x\ y \wedge eq\ (eqList\ d)\ xs\ ys$$
$$\quad\quad f\ \_\ \_ = False$$

$$elem :: Eq\ a \Rightarrow a \to [a] \to Bool$$
$$elem\ x\ [\,] = False$$
$$elem\ x\ (y : ys) = eq\ x\ y \vee elem\ x\ ys$$

$$main = elem\ \texttt{"Hello"}\ [\texttt{"Hello"}, \texttt{"World"}]$$

Translation

$$
\begin{array}{l}
\textbf{data}\ Eq\ a = Eq\{eq :: a \to a \to Bool\} \\[6pt]
eqChar :: Eq\ Char \\
eqChar = Eq\{eq = primEqChar\} \\[6pt]
eqList :: Eq\ a \to Eq\ [a] \\
eqList\ d = Eq\{eq = f\} \\
\quad \textbf{where}\ f\ [\,]\ [\,] = True \\
\qquad\qquad f\ (x : xs)\ (y : ys) = eq\ d\ x\ y \wedge eq\ (eqList\ d)\ xs\ ys \\
\qquad\qquad f\ \_\ \_ = False \\[6pt]
elem :: Eq\ a \to a \to [a] \to Bool \\
elem\ d\ x\ [\,] = False \\
elem\ d\ x\ (y : ys) = eq\ d\ x\ y \vee elem\ d\ x\ ys \\[6pt]
main = elem\ (eqList\ eqChar)\ \texttt{"Hello"}\ [\texttt{"Hello"}, \texttt{"World"}]
\end{array}
$$

Superclasses

$$\textbf{class}\ Eq\ a \Rightarrow Ord\ a\ \textbf{where}$$
$$\quad cmp :: a \rightarrow a \rightarrow Ordering$$

$$\textbf{instance}\ Ord\ Char\ \textbf{where}$$
$$\quad cmp = cmpChar$$

Superclasses

class $Eq\ a \Rightarrow Ord\ a$ **where**
  $cmp :: a \rightarrow a \rightarrow Ordering$

instance $Ord\ Char$ **where**
  $cmp = cmpChar$

$Eq\ a \Rightarrow Ord\ a$
means
$Eq \supseteq Ord$

Superclasses

$$\textbf{class}\ Eq\ a \Rightarrow Ord\ a\ \textbf{where}$$
$$\quad cmp :: a \rightarrow a \rightarrow Ordering$$
$$\textbf{instance}\ Ord\ Char\ \textbf{where}$$
$$\quad cmp = cmpChar$$

$Eq\ a \Rightarrow Ord\ a$
means
$Eq \supseteq Ord$

$$\begin{aligned}
&\textbf{data}\ Ord\ a = Ord\{ cmp \quad :: a \rightarrow a \rightarrow Bool \\
&\qquad\qquad\qquad\quad , eqOrd :: Eq\ a \quad \} \\
&ordChar :: \quad Ord\ Char \\
&ordChar = \quad Ord\{ cmp \quad = cmpChar \\
&\qquad\qquad\qquad\quad , eqOrd = eqChar \}
\end{aligned}$$

Problem statement: HUGE design space

Problem statement: HUGE design space

Constructor classes

$$
\begin{array}{l}
\textbf{class}\ \mathit{Monad}\ m\ \textbf{where} \\
\quad \mathit{return} :: a \rightarrow m\ a \\
\quad (\ggg\!=) :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Monad}\ [\,] \\
\textbf{instance}\ \mathit{Monad}\ \mathit{Maybe}
\end{array}
$$

Problem statement: HUGE design space

Constructor classes

Overlapping instances

instance $\mathit{Show}\ a \Rightarrow \mathit{Show}\ [a]$

instance $\mathit{Show}\ [\mathit{Char}]$

instance $\mathit{Show}\ a \Rightarrow \mathit{Show}\ [[a]]$

Problem statement: HUGE design space

Constructor classes

Overlapping instances

Multi-parameter type classes

class *Coll c e* **where**
  $empty :: c$
  $insert :: e \rightarrow c \rightarrow c$

instance *Coll* $[a]$ a

$test\ c = insert$ `'c'` $(insert\ True\ c)$

Problem statement: HUGE design space

Constructor classes

Overlapping instances

Multi-parameter type classes

Functional dependencies

$\textbf{class}\ \mathit{Coll}\ c\ e \mid c \rightarrow e\ \textbf{where}$
$\quad \mathit{empty} :: c$
$\quad \mathit{insert} :: e \rightarrow c \rightarrow c$

$\textbf{instance}\ \mathit{Coll}\ [a]\ a$

$\mathit{test}\ c = \mathit{insert}\ \texttt{'c'}\ (\mathit{insert}\ \mathit{True}\ c)$

Problem statement: HUGE design space

Constructor classes

Overlapping instances

Multi-parameter type classes

Functional dependencies

Local instances

$$
\begin{array}{l}
f\ g = \textbf{let instance}\ \mathit{Eq}\ \mathit{Int}\ \textbf{where} \\
\qquad\qquad x \equiv y = \mathit{primEqInt}\ (x\ \text{`}\mathit{mod}\text{`}\ 360)\ (y\ \text{`}\mathit{mod}\text{`}\ 360) \\
\qquad \textbf{in}\quad \ldots
\end{array}
$$

Problem statement: HUGE design space

Constructor classes

Overlapping instances

Multi-parameter type classes

Functional dependencies

Local instances

Different context-reduction strategies

$$f\ (x : xs)\ (y : xs) = x > y \wedge xs \equiv ys$$

-- Possible types for f

$$f :: Ord\ a \Rightarrow [a] \rightarrow [a] \rightarrow Bool$$
$$f :: (Ord\ a, Eq\ a) \Rightarrow [a] \rightarrow [a] \rightarrow Bool$$
$$f :: (Ord\ a, Eq\ [a]) \Rightarrow [a] \rightarrow [a] \rightarrow Bool$$

Problem statement: HUGE design space

- Constructor classes
- Overlapping instances
- Multi-parameter type classes
- Functional dependencies
- Local instances
- Different context-reduction strategies
- Type class directives

never	$Eq\ (a \rightarrow b)$	: `"functions cannot be tested for equa`
never	$Num\ Bool$	: `"arithmetic on booleans isn't suppor`
close	$Integral$	: `"the only Integral instances are Int`
disjoint	$Integral\ Fractional$	: `"something which is fractional can n`

Problem statement: HUGE design space

- Constructor classes
- Overlapping instances
- Multi-parameter type classes
- Functional dependencies
- Local instances
- Different context-reduction strategies
- Type class directives
- Other extensions encoded using predicates

$$f :: (r \ \mathit{lacks}\ l) \Rightarrow \ldots$$

$$g :: (?\mathit{width}) \Rightarrow \ldots$$

$$h :: (a \leqslant b) \Rightarrow \ldots$$

Problem statement: HUGE design space

- Constructor classes
- Overlapping instances
- Multi-parameter type classes
- Functional dependencies
- Local instances
- Different context-reduction strategies
- Type class directives
- Other extensions encoded using predicates
- Associated type synonyms

class *Collects c* **where**
  type *Elem c*
  empty :: *c*
  insert :: *Elem c* $\longrightarrow$ *c* $\longrightarrow$ *c*

Problem statement and approach

Problem statement

- No uniform approach to formulate type class extensions.
- Not easy to experiment with design decisions and extensions.
- Type error messages are difficult to understand.

Problem statement and approach

Problem statement

- No uniform approach to formulate type class extensions.
- Not easy to experiment with design decisions and extensions.
- Type error messages are difficult to understand.

Approach

- Formulate overloading into constraints.
- Let CHRs generate every (type) correct reduction alternative.
- Represent reduction alternatives in a graph.
- Use heuristics to find a solution in the graph.

Formulating overloading into constraints

Constraint language:

$$\textbf{data}\ \mathit{Constraint}\ \pi = \mathit{Prove}\ \pi \mid \mathit{Assume}\ \pi$$

Formulating overloading into constraints

Constraint language:

$$\textbf{data}\ \mathit{Constraint}\ \pi = \mathit{Prove}\ \pi \mid \mathit{Assume}\ \pi$$

Example 1:

$$\text{-- } \mathit{eq} :: \mathit{Eq}\ a \Rightarrow a \rightarrow a \rightarrow \mathit{Bool}$$

$$\mathit{test} = \mathit{eq}\ \texttt{""}\ \texttt{"Hello"}$$

Formulating overloading into constraints

Constraint language:

$$\textbf{data}\ \mathit{Constraint}\ \pi = \mathit{Prove}\ \pi \mid \mathit{Assume}\ \pi$$

Example 1:

-- $eq :: Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool$

$test = eq$ `""` `"Hello"`

$\mathit{Prove}\ (\mathit{Eq}\ [\mathit{Char}])$

Annotating predicates with scope

Example 2: Local instances

$$\text{-- } insert :: Ord\ a \Rightarrow a \rightarrow [a] \rightarrow [a]$$
$$\text{-- } sort :: Ord\ a \Rightarrow [a] \rightarrow [a]$$

$$
\begin{array}{l}
testInsert :: Ord\ a \Rightarrow a \rightarrow [a] \rightarrow Bool \\
testInsert\ x\ xs = \textbf{let instance}\ Eq\ a \Rightarrow Eq\ [a]\ \textbf{where} \\
\qquad\qquad\qquad\qquad\qquad eq = \ldots \\
\qquad\qquad\qquad\qquad ys = insert\ x\ (sort\ xs) \\
\qquad\qquad\qquad \textbf{in}\ \ eq\ ys\ (sort\ ys)
\end{array}
$$

Annotating predicates with scope

Example 2: Local instances

$$
\begin{array}{l}
\text{-- } \mathit{insert} :: \mathit{Ord}\ a \Rightarrow a \rightarrow [a] \rightarrow [a] \\
\text{-- } \mathit{sort} :: \mathit{Ord}\ a \Rightarrow [a] \rightarrow [a]
\end{array}
$$

$$
\begin{array}{l}
\mathit{testInsert} :: \mathit{Ord}\ a \Rightarrow a \rightarrow [a] \rightarrow \mathit{Bool} \\
\mathit{testInsert}\ x\ \mathit{xs} = \mathbf{let\ instance}\ \mathit{Eq}\ a \Rightarrow \mathit{Eq}\ [a]\ \mathbf{where} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{eq} = \ldots \\
\qquad\qquad\qquad\qquad\qquad \mathit{ys} = \mathit{insert}\ x\ (\mathit{sort}\ \mathit{xs}) \\
\qquad\qquad\qquad\qquad \mathbf{in}\ \ \mathit{eq}\ \mathit{ys}\ (\mathit{sort}\ \mathit{ys})
\end{array}
$$

Identification of scopes

- A scope is identified by a list of integers ($[\mathit{Int}]$),
- $[\,]$ is the global scope, and
- $[1,1], [1,2]$ are sibling scopes.

Annotating predicates with scope

Example 2: Local instances

$$\text{-- } \mathit{insert} :: \mathit{Ord}\ a \Rightarrow a \rightarrow [a] \rightarrow [a]$$
$$\text{-- } \mathit{sort} :: \mathit{Ord}\ a \Rightarrow [a] \rightarrow [a]$$

$$\begin{array}{l} \mathit{testInsert} :: \mathit{Ord}\ a \Rightarrow a \rightarrow [a] \rightarrow \mathit{Bool} \\ \mathit{testInsert}\ x\ \mathit{xs} = \textbf{let instance}\ \mathit{Eq}\ a \Rightarrow \mathit{Eq}\ [a]\ \textbf{where} \\ \qquad\qquad\qquad\qquad\qquad \mathit{eq} = \ldots \\ \qquad\qquad\qquad\qquad \mathit{ys} = \mathit{insert}\ x\ (\mathit{sort}\ \mathit{xs}) \\ \qquad\qquad\qquad\quad \textbf{in}\ \ \mathit{eq}\ \mathit{ys}\ (\mathit{sort}\ \mathit{ys}) \end{array}$$

$$\begin{array}{ll} \{\mathit{Assume} & (\mathit{Ord}\ c_1, [\,]\ ) \\ ,\ \mathit{Prove} & (\mathit{Ord}\ c_1, [1]) \\ ,\ \mathit{Prove} & (\mathit{Eq}\ [c_1], [1])\} \end{array}$$

Identification of scopes

- A scope is identified by a list of in
- $[\,]$ is the global scope, and
- $[1,1], [1,2]$ are sibling scopes.

Advantages

Constraints allows use to encode:

- Constructor classes
- Multi-parameter type classes
- Predicates annotated with a scope

Advantages

Constraints allows use to encode:

- Constructor classes
- Multi-parameter type classes
- Predicates annotated with a scope

Also other predicate extensions can be encoded:

- *lack* and *has* predicates for extensible records
- ? and % predicates for implicit parameters
- ....

Solving constraints

To solve constraints we

- Let CHRs generate *Reduction* constraints.
- Represent *Reduction* constraints in a graph.
- Use heuristics to find a solution in the graph.

Solving constraints

To solve constraints we

- Let CHRs generate *Reduction* constraints.
- Represent *Reduction* constraints in a graph.
- Use heuristics to find a solution in the graph.

Extend the constraint language

$$\textbf{data}\ \mathit{Constraint}\ \pi\ \mathit{info} = \ldots \\ \qquad\qquad\qquad\qquad \mid\ \mathit{Reduction}\ \pi\ \mathit{info}\ [\pi]$$

Solving constraints

To solve constraints we

- Let CHRs generate *Reduction* constraints.
- Represent *Reduction* constraints in a graph.
- Use heuristics to find a solution in the graph.

$$\{\,Reduction\,(Eq\ Char)\quad \texttt{"eqChar"}\quad [\,]$$
$$,\,Reduction\,(Eq\,(v_1, v_2))\ \texttt{"eqTuple"}\ [Eq\ v_1, Eq\ v_2]\,\}$$

Extend the co

$$\textbf{data}\ Constraint\ \pi\ info = \ldots$$
$$\mid\ Reduction\ \pi\ info\ [\pi]$$

Constraint Handling Rules

CHR syntax

$$C \Longrightarrow G \mid D \ (propagation)$$
$$C \Longleftrightarrow G \mid D \ (simplification)$$

Constraint Handling Rules

CHR syntax

$$C \Longrightarrow G \mid D \; (\mathit{propagation})$$
$$C \Longleftrightarrow G \mid D \; (\mathit{simplification})$$

Constraint Handling Rules

- Language for writing constraint solvers created by Thom Frühwirth.
- Idea of using CHRs for type classes proposed by Sulzmann et al.
- Understanding functional dependencies via CHRs.

Context reduction

Context reduction using instance declarations

instance *Eq Char* **where** ...

$$
\begin{aligned}
&\mathit{Prove}\ (\mathit{Eq}\ \mathit{Char}, s) \Longrightarrow [\,]\ \text{`}\mathit{visibleIn}\text{`}\ s \\
&\qquad\qquad\qquad\qquad\quad \mid\ \ \mathit{Reduction}\ (\mathit{Eq}\ \mathit{Char}, s)\ \texttt{"eqChar"}\ [\,]
\end{aligned}
$$

Context reduction

Context reduction using instance declarations

instance *Eq Char* **where** ...

$$\begin{aligned} Prove\ (Eq\ Char, s) \Longrightarrow\ & [\,]\ `visibleIn`\ s \\ \mid\ \ & Reduction\ (Eq\ Char, s)\ \texttt{"eqChar"}\ [\,] \end{aligned}$$

instance $Eq\ a \Rightarrow Eq\ [a]$ **where** ...

$$\begin{aligned} Prove\ (Eq\ [a], s) \Longrightarrow\ & [\,]\ `visibleIn`\ s \\ \mid\ \ & Prove\ (Eq\ a, s) \\ ,\ \ & Reduction\ (Eq\ [a], s)\ \texttt{"eqList"}\ [(Eq\ a, s)] \end{aligned}$$

Context reduction

Context reduction using the class hierarchy

class *Eq a* **where** ...
class *Eq a* ⇒ *Ord a* **where** ...

Context reduction

Context reduction using the class hierarchy

class *Eq a* **where** ...
class *Eq a* ⇒ *Ord a* **where** ...

-- reducing using the class hierarchy

$$\mathit{Prove}\ (\mathit{Eq}\ a, s), \mathit{Prove}\ (\mathit{Ord}\ a, s)$$
$$\Longrightarrow \mathit{Reduction}\ (\mathit{Eq}\ a, s)\ \texttt{"eqOrd"}\ [(\mathit{Ord}\ a, s)]$$

Context reduction

Context reduction using the class hierarchy

class *Eq a* **where** ...
class *Eq a* ⇒ *Ord a* **where** ...

$$\{\, Assume\ (Ord\ c_1, [\,])$$
$$,\ Prove\ \ (Eq\ \ c_1, [\,])\,\}$$

-- reducing using the class hierarchy
$Prove\ (Eq\ a, s), Prove\ (Ord\ a, s)$
$\Longrightarrow Reduction\ (Eq\ a, s)$ `"eqOrd"` $[(Ord\ a, s)]$

Context reduction

Context reduction using the class hierarchy

class *Eq a* **where** ...
class *Eq a* ⇒ *Ord a* **where** ...

$$\{\,Assume\ (Ord\ c_1, [\,])$$
$$,\ Prove\ \ (Eq\ \ c_1, [\,])\,\}$$

-- reducing using the class hierarchy

$Prove\ (Eq\ a, s), Prove\ (Ord\ a, s)$
$\Longrightarrow Reduction\ (Eq\ a, s)$ `"eqOrd"` $[(Ord\ a, s)]$

-- propagating the class hierarchy

$Assume\ (Ord\ a, s)$
$\Longrightarrow Assume\ (Eq\ a, s), Reduction\ (Eq\ a, s)$ `"eqOrd"` $[(Ord\ a, s)]$

Simplification of predicates annotated with scope

How can the following predicates be simplified?

$$
\begin{array}{lll}
\{\, \mathit{Assume} & (\mathit{Ord} & c_1, [\,]) \\
,\ \mathit{Prove} & (\mathit{Ord} & c_1, [1]) \\
,\ \mathit{Prove} & (\mathit{Eq} & c_1, [1])\,\}
\end{array}
$$

Simplification of predicates annotated with scope

How can the following predicates be simplified?

$$
\begin{array}{lll}
\{\, \mathit{Assume} & (\mathit{Ord}\ c_1, [\,]) \\
,\ \mathit{Prove} & (\mathit{Ord}\ c_1, [1]) \\
,\ \mathit{Prove} & (\mathit{Eq}\ \ c_1, [1])\,\}
\end{array}
$$

Lifting a predicate to the parent scope

$$
\begin{array}{lll}
\mathit{Prove}\ (p, s) \Longrightarrow & & \mathit{not}\ (\mathit{toplevel}\ s) \\
& | & \mathit{Prove}\ (p, \mathit{parent}\ s) \\
& , & \mathit{Reduction}\ (p, s)\ \texttt{"scope"}\ [(p, \mathit{parent}\ s)]
\end{array}
$$

Simplification of predicates annotated with scope

How can the following predicates be simplified?

$$\begin{aligned}&\{\,\mathit{Assume}\ (\mathit{Ord}\ c_1, [\,])\\ &,\ \mathit{Prove}\ \ (\mathit{Ord}\ c_1, [1])\\ &,\ \mathit{Prove}\ \ (\mathit{Eq}\ \ c_1, [1])\,\}\end{aligned}$$

Lifting a

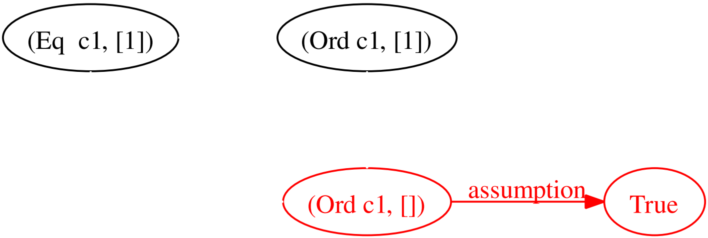
Prove (

$$\begin{aligned}&\{\,\mathit{Reduction}\ (\mathit{Ord}\ c_1, [1])\ \texttt{"scope"}\ [(\mathit{Ord}\ c_1, [\,])]\\ &,\ \mathit{Reduction}\ (\mathit{Eq}\ \ c_1, [1])\ \texttt{"eqOrd"}\ [(\mathit{Ord}\ c_1, [1])]\\ &,\ \mathit{Reduction}\ (\mathit{Eq}\ \ c_1, [1])\ \texttt{"scope"}\ [(\mathit{Eq}\ \ c_1, [\,])]\\ &,\ \mathit{Reduction}\ (\mathit{Eq}\ \ c_1, [\,])\ \ \texttt{"eqOrd"}\ [(\mathit{Ord}\ c_1, [\,])]\\ &\}\end{aligned}$$

Simplification graph for local instances

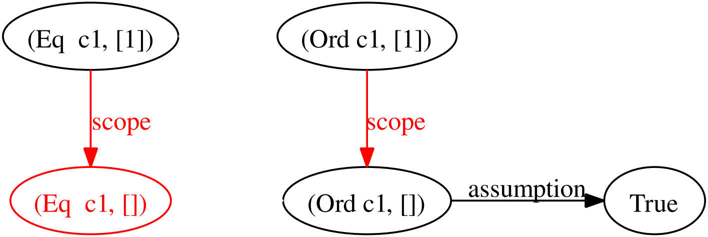
$Prove\ (Eq\ \ c_1, [1])$
$Prove\ (Ord\ c_1, [1])$

Simplification graph for local instances



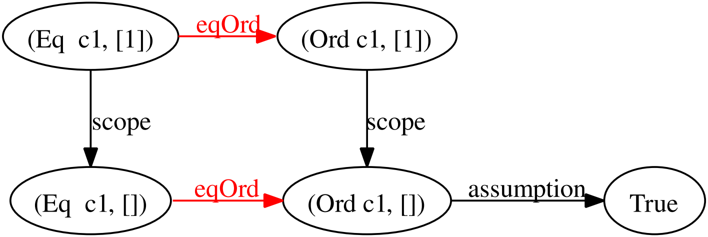
Assume $(Ord\ c_1, [\,])$

Simplification graph for local instances



$$Reduction\ (Eq\ \ c_1, [1])\ \texttt{"scope"}\ [(Eq\ \ c_1, [\,])]$$
$$Reduction\ (Ord\ c_1, [1])\ \texttt{"scope"}\ [(Ord\ c_1, [\,])]$$

Simplification graph for local instances



$$\mathit{Reduction}\ (\mathit{Eq}\ c_1, [1])\ \texttt{"eqOrd"}\ [(\mathit{Ord}\ c_1, [1])]$$
$$\mathit{Reduction}\ (\mathit{Eq}\ c_1, [\,])\ \ \texttt{"eqOrd"}\ [(\mathit{Ord}\ c_1, [\,])]$$

Advantages of simplification graphs

Graphs make experimenting with type classes easy!

- Visualization of the problem!
- Lot of information present for type error messages.
- Every (type) correct reduction alternative is present.
- Specify alternative reduction strategies in heuristics.

Advantages of simplification graphs

Graphs make experimenting with type classes easy!

- Visualization of the problem!
- Lot of information present for type error messages.
- Every (type) correct reduction alternative is present.
- Specify alternative reduction strategies in heuristics.

Examples of heuristics:

- Heuristic emulating Haskell 98 or GHC context reduction.
- Cost-path heuristic.
- Heuristic that utilizes backtracking to find a solution.
- Or a nice combination of the above heuristics.

Example: local and overlapping instance declarations

$$\begin{array}{ll}
\textbf{class}\ \mathit{Show}\ a\ \textbf{where} & \\
\quad \mathit{show} :: a \rightarrow [\mathit{Char}] & \\
\textbf{instance}\ \mathit{Show}\ \mathit{Char} & \text{-- showChar} \\
\textbf{instance}\ \mathit{Show}\ a \Rightarrow \mathit{Show}\ [a] & \text{-- show[]} \\
\textbf{instance}\ \mathit{Show}\ [\mathit{Char}] & \text{-- show[Char]}
\end{array}$$

$$\begin{array}{l}
\mathit{ppTable}\ \mathit{hdr}\ \mathit{tbl} \\
\quad = \textbf{let instance}\ \mathit{Show}\ a \Rightarrow \mathit{Show}\ [[a]] \quad \text{-- show[[]]} \\
\qquad \textbf{in} \quad \ldots\ \mathit{show}\ (\mathit{hdr} : \mathit{tbl})\ \ldots
\end{array}$$

$$\begin{array}{l}
\mathit{main} = \mathit{ppTable}\ [\texttt{"Name"}, \texttt{"DOB"}]\ [[\texttt{"G"}, \texttt{"19830511"}] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad , [\texttt{"A"}, \texttt{"19830208"}]]
\end{array}$$

Example: local and overlapping instance declarations

class *Show a* **where**
  $show :: a \rightarrow [Char]$

instance *Show Char* -- showChar
instance $Show\ a \Rightarrow Show\ [a]$ -- show[]
instance $Show\ [Char]$ -- show[Char]

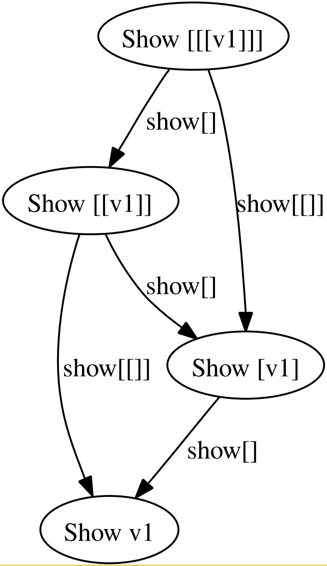
ppTable hdr tbl
  = **let instance** $Show\ a \Rightarrow Show\ [[a]]$ -- show[[]]

m

$Prove\ (Show\ [[[v_1]]])$

$ppTable :: Show\ [[[a]]] \Rightarrow [[a]] \rightarrow [[[a]]] \rightarrow [Char]$
$ppTable :: Show\ [a] \Rightarrow [[a]] \rightarrow [[[a]]] \rightarrow [Char]$
$ppTable :: Show\ a \Rightarrow [[a]] \rightarrow [[[a]]] \rightarrow [Char]$

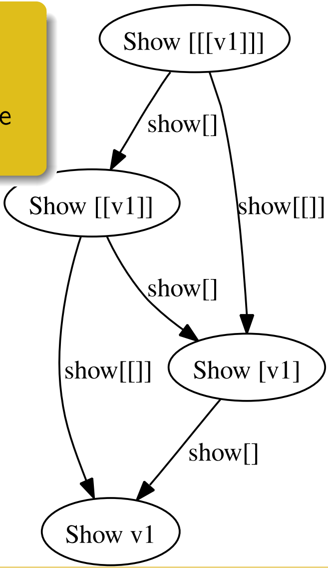
Simplification graph: first attempt



Simplification graph: first attempt

Heuristic:

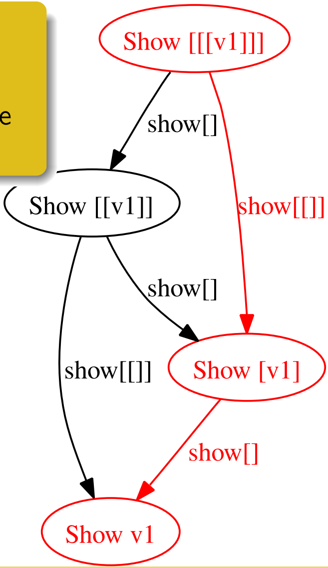
- Eager reduction
- Prefer local above global



Simplification graph: first attempt

Heuristic:

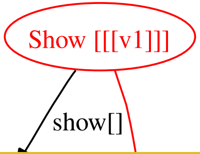
- Eager reduction
- Prefer local above global



Simplification graph: first attempt

Heuristic:

- Eager reduction
- Prefer local above global

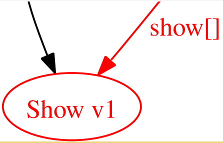


Inferred type for the function *ppTable*

$\textbf{instance}\ \mathit{Show}\ [\mathit{Char}]$ -- show[Char]

$\mathit{ppTable} :: \mathit{Show}\ a \Rightarrow [[a]] \rightarrow [[[a]]] \rightarrow [\mathit{Char}]$

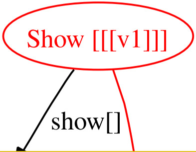
$\mathit{main} = \mathit{ppTable}\ [\texttt{"Name"}, \texttt{"DOB"}]\ [[\texttt{"G"}, \texttt{"19830511"}]$
$\qquad\qquad , [\texttt{"A"}, \texttt{"19830208"}]]$



Simplification graph: first attempt

Heuristic:

- Eager reduction
- Prefer local above global



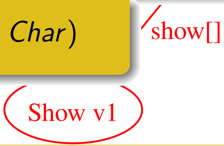
Inferred type for the function *ppTable*

$\textbf{instance}\ Show\ [Char]$ -- show[Char]

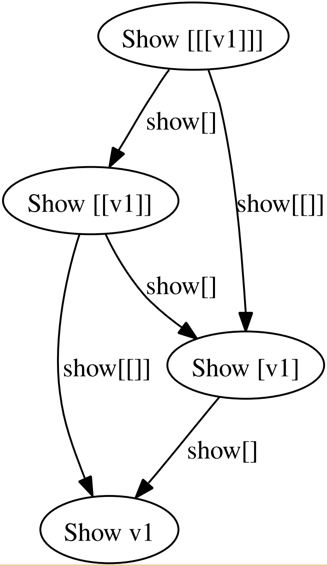
$ppTable :: Show\ a \Rightarrow [[a]] \rightarrow [[[a]]] \rightarrow [Char]$

$main = ppTable$ `["Name", "DOB"] [["G", "19830511"]`
`, ["A", "19830208"]]`

Prove (Show Char)



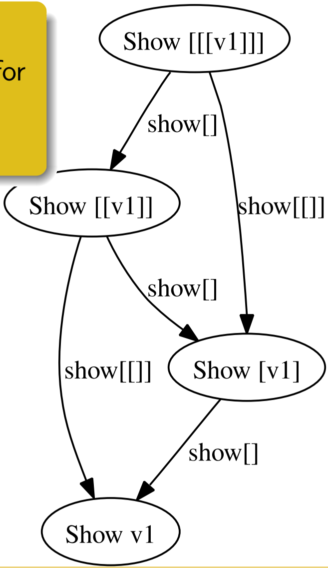
Simplification graph: second attempt



Simplification graph: second attempt

Heuristic II:

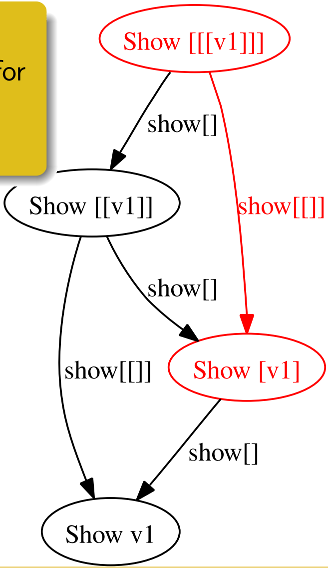
- Eager reduction for local instances.
- Otherwise stop.



Simplification graph: second attempt

Heuristic II:

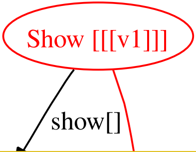
- Eager reduction for local instances.
- Otherwise stop.



Simplification graph: second attempt

Heuristic II:

- Eager reduction for local instances.
- Otherwise stop

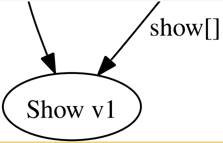


Inferred type for the function *ppTable*

$$\textbf{instance}\ \mathit{Show}\ [\mathit{Char}] \quad \text{-- show[Char]}$$

$$\mathit{ppTable} :: \mathit{Show}\ [a] \Rightarrow [[a]] \rightarrow [[[a]]] \rightarrow [\mathit{Char}]$$

$$\mathit{main} = \mathit{ppTable}\ [\texttt{"Name"}, \texttt{"DOB"}]\ [[\texttt{"G"}, \texttt{"19830511"}]\\ ,[\texttt{"A"}, \texttt{"19830208"}]]$$



Simplification graph: second attempt

Heuristic II:

- Eager reduction for local instances.
- Otherwise stop

Show [[[v1]]]

show[]

Inferred type for the function *ppTable*

$\textbf{instance}\ \mathit{Show}\ [\mathit{Char}]$ -- show[Char]

$\mathit{ppTable} :: \mathit{Show}\ [a] \Rightarrow [[a]] \rightarrow [[[a]]] \rightarrow [\mathit{Char}]$

$\mathit{main} = \mathit{ppTable}\ [\texttt{"Name"}, \texttt{"DOB"}]\ [[\texttt{"G"}, \texttt{"19830511"}]$
$\qquad\qquad\qquad\qquad\qquad\quad , [\texttt{"A"}, \texttt{"19830208"}]]$

Prove $(\mathit{Show}\ [\mathit{Char}])$

show[]

Show v1

Conclusion

Contributions

- First in using graphs and heuristics to solve and experiment with type classes.
- First in using CHRs with explicit *Prove* and *Assume* constraints.
- Implementation of this framework and a basic CHR solver.

Conclusion

Contributions

- First in using graphs and heuristics to solve and experiment with type classes.
- First in using CHRs with explicit *Prove* and *Assume* constraints.
- Implementation of this framework and a basic CHR solver.

Conclusion

- Uniform encoding of type class extensions.
- Graphs and heuristics make it easy to experiment with extensions.
- Every well known type class extension can be encode using this framework.
- Framework can be used for both Helium/Top and EHC.