Type Class Directives

Bastiaan Heeren   Jurriaan Hage

Institute of Information and Computing Sciences
Utrecht University

{bastiaan,jur}@cs.uu.nl

January 10, 2005

Overview

- Introduction
- Type class directives
  - The `never` directive
  - The `close` directive
  - The `disjoint` directive
  - The `default` directive
- Implementation
- Generalization
- Specialized type rules
- Conclusion

Introduction

- Student exercise: decrement the elements of a list

```
f xs = map -1 xs
```

Introduction

- Student exercise: decrement the elements of a list

```
f xs = map -1 xs
```

- "If a program type checks, it works ... usually"
- The following type is inferred by ghci (without extensions):

```
f :: forall b a t.
        ( Num (t -> (a -> b) -> [a] -> [b])
        , Num ((a -> b) -> [a] -> [b])
        ) => t -> (a -> b) -> [a] -> [b]
```

- This type is likely to cause problems at the site where `f` is used

Introduction

- Student exercise: decrement the elements of a list

```
f xs = map -1 xs
```

- "If a program type checks, it works ... usually"
- The following type is inferred by ghci (without extensions):

```
f :: forall b a t.
        ( Num (t -> (a -> b) -> [a] -> [b])
        , Num ((a -> b) -> [a] -> [b])
        ) => t -> (a -> b) -> [a] -> [b]
```

- This type is likely to cause problems at the site where `f` is used
- More examples:

```
f1 xs = map (-1) xs
f2 n  = [n] : [4,5,6]
f3 r  = r * sin .2
```

Type class directives

- Haskell declarations:
  - class declarations: list superclasses ($Ord \subseteq Eq$)
  - instance declarations: add type to a type class ($Int \in Eq$)

- We propose four directives to enrich the specification of H98 type classes

- Builds on the approach of an earlier paper:
  "Scripting the type inference process" (ICFP'03)
  - The directives for `X.hs` can be found in `X.type`

Type class directives

- Haskell declarations:
  - class declarations: list superclasses ($Ord \subseteq Eq$)
  - instance declarations: add type to a type class ($Int \in Eq$)

- We propose four directives to enrich the specification of H98 type classes

- Builds on the approach of an earlier paper:
  "Scripting the type inference process" (ICFP'03)
  - The directives for `X.hs` can be found in `X.type`

- The directive approach proceeds in three steps:
  1. Collect the class and instance declarations in scope
  2. Check consistency between the directives and the declarations
  3. Perform type inference using the directives

The `never` directive (1)

- Is used to exclude a type from a type class
- Advantage: the directive is accompanied by a tailor-made error message

in .type file

```
never Eq (a -> b): functions cannot be tested for equality
never Num Bool: arithmetic on booleans is not supported
```

The `never` directive (1)

- Is used to exclude a type from a type class
- Advantage: the directive is accompanied by a tailor-made error message

in .type file

```
never Eq (a -> b): functions cannot be tested for equality
never Num Bool: arithmetic on booleans is not supported
```

- If an instance for `Num Bool` is required to resolve overloading, the special purpose error message is reported

```
f x = if x then x + 1 else x
```

```
(1, 19): arithmetic on booleans is not supported
```

The `never` directive (2)

- Before type inference, we check the consistency of the directives with the class and instance declarations

- An example:

```
instance Num Bool where ...
```

in .type file

```
never Num Bool: arithmetic on booleans is not supported
```

- For this inconsistency, we report the following:

```
The instance declaration for
   Num Bool at (3,1) in A.hs
is in contradiction with the directive
   never Num Bool defined at (1,1) in A.type
```

The `close` directive (1)

- Close a type class: no new instances can be defined for that class
- Similar to the case-by-case `never` directive
- We give `never` precedence over `close`
- Advantage: compiler knows all instances of a closed type class

in .type file

```
close Integral: the only instances of Integral
                are Int and Integer
```

- We can further exploit having the fixed set of instances...

The `close` directive (2)

Two optimizations (both are optional):

- Create an error message for predicates concerning an empty type class
  - Advantage: report mistakes early on
  - Same reasoning applies to sets of closed type classes
    e.g., `(X a, Y a)` and intersection of `X` and `Y` is empty

The `close` directive (2)

Two optimizations (both are optional):

- Create an error message for predicates concerning an empty type class
  - Advantage: report mistakes early on
  - Same reasoning applies to sets of closed type classes
    e.g., `(X a, Y a)` and intersection of `X` and `Y` is empty

- Improvement substitution for singleton type class

```
f :: (Bounded a, Num a) => a -> a
```

can be improved to

```
f :: Int -> Int
```

  - Advantage: simpler types
  - Disadvantage: not something to rely on in large programming projects

The `disjoint` directive

- The classes `Integral` and `Fractional` are intentionally disjoint
- Advantage: report mistakes early on

in .type file

```
disjoint Integral Fractional:
   something which is fractional can never be integral
```

```
wrong x = (div x 2, x/2)
```

```
wrong :: (Fractional a, Integral a) => a -> (a,a)
```

The `disjoint` directive

- The classes `Integral` and `Fractional` are intentionally disjoint
- Advantage: report mistakes early on

in .type file

```
disjoint Integral Fractional:
   something which is fractional can never be integral
```

```
wrong x = (div x 2, x/2)
```

```
wrong :: (Fractional a, Integral a) => a -> (a,a)
```

- Because `Floating` $\subseteq$ `Fractional`, we implicitly have

```
disjoint Integral Floating
```

The `default` directive

- Some seemingly innocent programs are in fact ambiguous

```
main = show []
```

- Haskell 98: defaulting for numeric type classes (conservative)
- ghci extends defaulting rules to standard classes (ad-hoc)

The `default` directive

- Some seemingly innocent programs are in fact ambiguous

```
main = show []
```

- Haskell 98: defaulting for numeric type classes (conservative)
- ghci extends defaulting rules to standard classes (ad-hoc)

in .type file

```
default Num (Int, Integer, Float, Double)
default Show ((), String, Bool, Int)
```

- Defaulting can be considered a type class directive
- Unquestionably useful at times
- Always notify the programmer with a warning

Implementation: context reduction

- To implement the four type class directives, we change context reduction
- Performed at all generalization points
- Divided into three phases:
  1. **Simplify** using instance declarations
     - `Eq Int` → can be removed
     - `Eq (a, b)` → split into (`Eq a`) and (`Eq b`)
     - `Num Bool` → create an error message

Implementation: context reduction

- To implement the four type class directives, we change context reduction
- Performed at all generalization points
- Divided into three phases:

1. **Simplify** using instance declarations
   - `Eq Int` $\longrightarrow$ can be removed
   - `Eq (a, b)` $\longrightarrow$ split into (`Eq a`) and (`Eq b`)
   - `Num Bool` $\longrightarrow$ create an error message

2. Remove **duplicates** and **superclasses**
   - {`Eq a, Eq a`} $\longrightarrow$ remove duplicate
   - {`Eq a, Ord a`} $\longrightarrow$ remove (`Eq a`)

Implementation: context reduction

- To implement the four type class directives, we change context reduction
- Performed at all generalization points
- Divided into three phases:

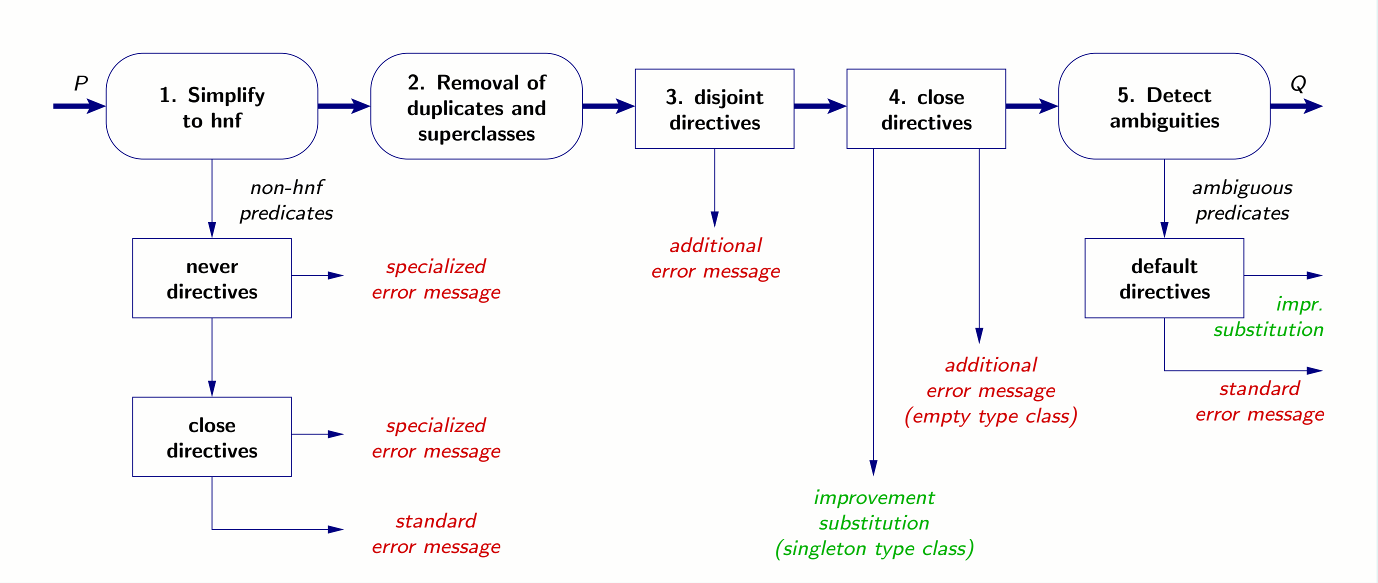
1. **Simplify** using instance declarations
   - `Eq Int` → can be removed
   - `Eq (a, b)` → split into (`Eq a`) and (`Eq b`)
   - `Num Bool` → create an error message

2. Remove **duplicates** and **superclasses**
   - `{Eq a, Eq a}` → remove duplicate
   - `{Eq a, Ord a}` → remove (`Eq a`)

3. Report **ambiguities**
   - `Eq a` → create an error message if a is a polymorphic type variable that does not occur in the type

Implementation: type class directives

- Context reduction with type class directives for Haskell 98



Generalization of directives

- Essentially, type class directives describe *invariants* over type classes
- Generalization: a small language to specify constraints on sets

$$
\begin{array}{lcl}
\mathit{Constraint} & ::= & \mathit{Type}\ \mathit{EltOp}\ \mathit{Set} \mid \mathit{Set}\ \mathit{SetOp}\ \mathit{Set} \\
\mathit{Set} & ::= & \mathit{BinOp}\ \mathit{Set}\ \mathit{Set} \mid \mathit{SetLiteral} \mid \mathit{Class} \\
\mathit{SetLiteral} & ::= & \{\} \mid \{\ \mathit{Type}\ (,\ \mathit{Type})^*\ \} \\
\mathit{EltOp} & ::= & \texttt{isin} \mid \texttt{isnotin} \\
\mathit{SetOp} & ::= & <= \mid == \mid >= \\
\mathit{BinOp} & ::= & \texttt{union} \mid \texttt{intersect} \mid \texttt{difference}
\end{array}
$$

Generalization of directives

- Essentially, type class directives describe *invariants* over type classes
- Generalization: a small language to specify constraints on sets

$$
\begin{array}{lcl}
\mathit{Constraint} & ::= & \mathit{Type}\ \mathit{EltOp}\ \mathit{Set} \;\mid\; \mathit{Set}\ \mathit{SetOp}\ \mathit{Set} \\
\mathit{Set} & ::= & \mathit{BinOp}\ \mathit{Set}\ \mathit{Set} \;\mid\; \mathit{SetLiteral} \;\mid\; \mathit{Class} \\
\mathit{SetLiteral} & ::= & \{\} \;\mid\; \{\ \mathit{Type}\ (,\ \mathit{Type})^{*}\ \} \\
\mathit{EltOp} & ::= & \texttt{isin} \;\mid\; \texttt{isnotin} \\
\mathit{SetOp} & ::= & <= \;\mid\; == \;\mid\; >= \\
\mathit{BinOp} & ::= & \texttt{union} \;\mid\; \texttt{intersect} \;\mid\; \texttt{difference}
\end{array}
$$

in .type file

```
Monad  ==  {Maybe, [], IO}: only Maybe, [], and IO
                            are monads today.
Read   ==  Show
intersect Egglayer Mammal  <=  {Platypus}
```

- The expressiveness must be limited for efficiency and decidability

Specialized type rules (1)

```
map :: (a -> b) -> [a] -> [b]
```

- Reprogram type inference for `map`
  - Change the order of unifications (when is an inconsistency found)
  - Provide special type error messages (domain-specific)

Specialized type rules (1)

```
map :: (a -> b) -> [a] -> [b]
```

- Reprogram type inference for `map`
  - Change the order of unifications (when is an inconsistency found)
  - Provide special type error messages (domain-specific)

in .type file

```
 f :: t1;   xs :: t2;
---------------------
  map f xs :: [b];

t1 == a1 -> b1 : @f.pp@ is not a function
t2 == [a2]     : @xs.pp@ is not a list
a1 == a2       : function @f.pp@ does not work on @a2@
b1 == b        : function @f.pp@ does not return @b@
```

- Guarantee that underlying type system is unchanged
  (a mistake is easily made)

Specialized type rules (2)

```
spread :: (Ord a, Num a) => [a] -> a
spread xs = maximum xs - minimum xs
```

- Specialized type rule for `spread` with class assertions

in .type file

```
    xs :: t1;
--------------------
  spread xs :: t2;

t1 == [t3]: @xs.pp@ must be a list
t3 == t2: @expr.pp@ should return a value of type @t3@
Eq t2: @t2@ is not an instance of Eq, let alone Ord or Num
Ord t2: @t2@ should have a linear ordering imposed on it
Num t2: @t2@ should allow numerical operations
```

- Class assertions are listed and checked after unification constraints
- We can still check the correctness of the specialized type rules

Related work

- Elements of our work can be found in earlier papers:
  - Closed type classes were mentioned by Shields and Peyton Jones *"Object-oriented style overloading for Haskell"*
  - Glynn et al. describe disjoint type classes and complements using CHRs *"Type classes and constraint handling rules"*
  - Improvement substitutions are part of Mark Jones's framework *"Simplifying and improving qualified types"*

- All these efforts focused on the type system: we concentrate on providing high-level support for high quality type error messages

Conclusion and future work

- Advantages of the four type class directives:
  - Tailor-made, domain-specific error messages for special cases
  - Type schemes with a suspicious class context are rejected early on
  - A limited set of instances helps to improve and simplify types
  - An effective defaulting mechanism assists to disambiguate overloading

Conclusion and future work

- Advantages of the four type class directives:
  - Tailor-made, domain-specific error messages for special cases
  - Type schemes with a suspicious class context are rejected early on
  - A limited set of instances helps to improve and simplify types
  - An effective defaulting mechanism assists to disambiguate overloading

- Future work:
  - A small language to specify invariants (including some analysis)
  - Explore directives for various extensions to the type class system
  - A formal description of the directives