



# Subtyping and type reconstruction in Timber

Johan Nordlander

Oct '09



# Subtyping

- Java style:

```
interface Vehicle { .. }
```

```
interface Car extends Vehicle { .. }
```

```
interface Bike extends Vehicle { .. }
```

```
class Volvo implements Car { .. }
```

```
class DSB implements Bike { .. }
```

...

```
Vehicle list[] = new Vehicle[3];
```

```
list[0] = new Volvo();
```

```
list[1] = new DSB();
```

# Type reconstruction

- In Haskell:

```
f x = x ++ ".com"
```

```
f :: String -> String      -- obtained from the type of ++
```

```
twice g x = g (g x)
```

```
twice :: (a -> a) -> a -> a  -- a polymorphic type!
```

- Inferred types in Haskell are principal types; i.e., they are as general as possible. C.f. another legal typing:

```
twice :: (Int -> Int) -> Int -> Int
```

This type can be obtained by substituting `Int` for `a` in the principal type for `twice`

- Java has polymorphism, but not type reconstruction
- Experience suggests reconstruction is vital to usefulness

# Overloading

- Static overloading:

```
typeclass Eq a where (==) :: a -> a -> Bool
```

```
instance Eq Int where ...
```

```
instance Eq String where
```

```
f x y = x == (y ++ ".com")
```

```
f :: String -> String -> Bool    -- inferred type
```

- Dynamic overloading:

```
f x [] = False
```

```
f x (y:ys) = if x == y then True else f x ys
```

```
f :: a -> [a] -> Bool \ \ Eq a
```

# Inferred constraints

- Constraint sets like `Eq a`, `Num b`, `Show [c]` etc are collected during type reconstruction
- Can in theory be returned as they are in inferred types
- However, programmers tend to prefer some amount of constraint solving/reduction/simplification:
  - Removal of duplicates (`Eq a, Eq a --> Eq a`)
  - Remove tautologies by instance axioms (`Eq Int -->` )
  - Reduce using instance rules  
(`Eq [a] --> Eq a`, if we have `instance Eq [a] \\Eq a)`
  - Utilize "sub-superclass" relationships  
(`Eq a, Ord a --> Ord a`, if `typeclass Ord a < Eq a`)

# Incorporating subtyping #1

- By encoding in the type class system:

**typeclass** Sub a b **where**

`coerce :: a -> b`

- Explicit coercions (`f (coerce e)` instead of `f e`)
- Explicit subtype instances

**instance** Sub Car Vehicle **where**

`coerce car = ...`

- Extremely tedious to manually write coercions, goes against the very idea of subtyping

# Incorporating subtyping #2

- Encoding + syntactic sugar
  - Implicit coercions: silently apply `coerce` to
    - + every function argument
    - + every struct selection
    - + every case scrutinee
  - Explicit subtype instances (as in #1, possibly with `a < b` as syntactic sugar for `Sub a b`)
- Has been suggested and tried "in principle" (Jones, Kaes, Smith & Volpano)
- But subtyping isn't just an arbitrary relation on types...

# The subtyping relation

- Reflexive ( $T < T$  for every  $T$ )
- Transitive ( $A < B$  and  $B < C$  implies  $A < C$ )
- Asymmetric ( $A < B$  and  $B < A$  implies  $A = B$ )
- Co/contravariant ( $A \rightarrow B < A' \rightarrow B'$  iff  $A' < A$  and  $B < B'$ )
- Coherent (coercing  $A < C$  via  $A < B_1 < C$  or  $A < B_2 < C$  is dynamically irrelevant)
- These properties constitute requirements on instances
  - Prohibits an arbitrary instance structure
- They also offer extra assumptions on constraints
  - Enables more kinds of simplifications

# Incorporating subtyping #3

- Timber

- Implicit coercions (at every function application, struct selection, case selection, ...)
- Implicit subtype instances derived from type declarations

`struct B < A where ...`

`data D > C = ...`

- Challenge: incorporate new forms of constraint simplification!

# Constraint simplification

## ○ General:

- Remove duplicates (and superclass implications)
- Reduce by applying instance axioms and rules

## ○ Subtype specific:

- Remove reflexive constraints ( $a < a$ ,  $T < T$ , ...)
- Utilize transitive relationships  
(replace  $a < b$ ,  $b < c$  with  $a < c$ )
- Utilize asymmetry of subtyping (C-simplification):  
unify  $a$ ,  $b$  and  $c$  whenever  $a < b$ ,  $b < c$ ,  $c < a$
- Utilize co/contravariance (S-simplification)

# S-simplification

- Example: due to co/contravariance of the function type constructor, the constrained type  $(a \rightarrow b \ \backslash \ a < A, B < b)$  can be rewritten as  $A \rightarrow B$
- Illustrates human preference for syntactically smaller types!
- Justified by theorem that shows

$$P \vdash (a \rightarrow b \ \backslash \ a < A, B < b) < T$$

iff

$$P \vdash A \rightarrow B < T$$

# Simplification algorithms

- Has been extensively studied for subtyping in isolation, but not incorporated in a full language
- Combination with type classes proposed by Shields & Peyton Jones (extremely complex, never implemented)
- However, there is another, and more fundamental, problem lurking: the question of

principality vs. syntactic minimality!

# Principality vs. minimality

- Polymorphic systems:

For all  $e :: T$ , if  $T$  is principal (most useful in all contexts) the  $T$  is also minimal (no bigger than any ad hoc type for  $e$ )

- Polymorphic systems with overloading:

Principal type  $T$  might be bigger than an ad hoc type for  $e$ , but difference is bounded by the number of overloaded operators in  $e$

- Polymorphic systems with subtyping:

Principal type  $T$  is in general as big as  $e$ !

"Inferred subtype constraints considered harmful"

# Principality vs. minimality

- Consider

$\text{min } x \ y = \text{if less } x \ y \text{ then } x \ \text{else } y$

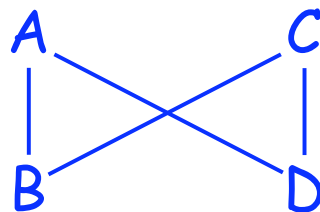
$\text{less} :: A \rightarrow A \rightarrow \text{Bool}$

- In Haskell,  $A \rightarrow A \rightarrow A$  is a principal type for  $\text{min}$

- But if subtyping is added the principal type becomes

$\text{min} :: a \rightarrow b \rightarrow c \ \backslash \ a < A, b < A, a < c, b < c$

- An subtype hierarchy where this generality is needed:



# Constraint approximation

- Ad hoc algorithm of O'Haskell (precursor to Timber)
  - Performed extremely well in practice
  - Approximation intertwined with simplification, using only local knowledge
  - Only partial completeness results
  - Totally ignored type-class constraints
- Goal set up for Timber: define new algorithm that
  - Simplifies subtype & class constraints in tandem
  - Simplifies first, then approximates if needed
  - Is simple & performs extremely well in practice!

# Three insights

- Subtype simplification is a form of **improvement**
- Subtype instances form an **overlapping** hierarchy
- Subtype approximation is a form of **defaulting**
  - which is a form of "aggressive" improvement

# Improvement

- Note type inferred for `single` below:  
`typeclass Collection e ce where`  
    `insert :: e -> ce -> ce`  
    `member :: e -> ce -> Bool`  
`instance Collection Char String where ...`  
    `single x = insert x ""`  
    `single :: a -> String \ Collection a String`
- Would have expected `single :: Char -> String`, since we "know" there will be no other `instance Collection X String`
- Expressible as a functional dependency (Haskell extension)  
`typeclass Collection e ce | ce -> e where ...`  
Read as "`ce` uniquely determines `e`"

# Improvement

- With functional dependencies we can expect constraint `Collection a String` to be improved by substitution `Char/a` (and then removed because it becomes a tautology)
- An improving substitution replaces variables in a constraint without making it less general
- Now recall the constraint set `a < A, B < b`. If `a` and `b` are contra- and co-variant in the inferred type, `A/a` and `B/b` are indeed improving substitutions for `a < A, B < b`

# Overlapping instances

- Showing lists as text:

```
instance Show [a] \ Show a where
```

```
  show xs = "[" ++ intersperse ", " (map show x) ++ "]"
```

- An established exception: showing lists of characters

```
instance Show [Char] where
```

```
  show xs = "\"" ++ xs ++ "\""
```

- Note how `Show [Char]` overlaps with `Show [a]`, with the former instance having preference over the latter
- Not accepted by Haskell, but common as an extension

# Overlapping instances

- Consider the type  $a \rightarrow \text{Int} \ \backslash \ a < A, \text{Eq } a$  in the context  
struct  $B < A$  where ...  
struct  $C < B$  where ...  
instance  $\text{Eq } B$  where ...
- We have many implicit instances that can solve  $a < A$ :  
 $A < A, B < A, C < A, \dots$
- But since  $a$  is in a contravariant position, we'd like  $A < A$  to have preference over  $B < A$ , and  $B < A$  over  $C < A$ , etc...
- No **instance Eq A** means solving  $a < A$  using  $A < A$  fails
- The second choice  $B < A$  works, though:  $B/a$  improves the type into  $B \rightarrow \text{Int} \ \backslash \ B < A, \text{Eq } B$  (reduces to  $B \rightarrow \text{Int}$ )

# Defaulting

- Consider

$f\ x = \text{if show (1+6) == "7" then } x \text{ else } x++x$

$f :: \text{String} \rightarrow \text{String} \ \backslash \ \text{Num } b, \text{ Show } b$

- How choose this  $b$ ? The choice obviously determines whether  $\text{show (1+6) == "7"}$  may succeed

- Lesson from Haskell: the type of  $f$  is ambiguous and cannot be accepted. But one may declare a default instance for  $\text{Num}$ , say  $\text{Num Int}$ , to use in such cases

- Substituting  $\text{Int}$  for  $b$  in the type above can be viewed as an aggressive form of improvement, that takes the "artificial" overlap  $\text{Num Int} \leq \text{Num } \_$  into account

# Defaulting

- Defaulting deliberately approximates types since we want to avoid an ambiguous semantics
- With subtyping we simply add another criterion for defaulting: we want to avoid an excessive type syntax!
- Consider

$\text{twice } f \ x = f (f \ x)$

Before defaulting  $f :: (a \rightarrow b) \rightarrow a \rightarrow b \ \backslash \ b < a$

After defaulting  $f :: (a \rightarrow a) \rightarrow a \rightarrow a,$

using the built-in forced preference for the reflexive instance over anything else

# The algorithm: preliminaries

○ We have:

- a set  $P$  of instance axioms and rules:

$Eq\ Int, Eq\ Float, Eq\ [a] \setminus \setminus Eq\ a, Num\ Int, A < B, \dots$

Note:  $P$  has no free variables

- a set  $Q$  of constraints that must be implied by  $P$  for some substitution of its free variables:

$Num\ x, Eq\ [y], z < A, \dots$

○ Formally:  $P \vdash \theta Q$  for some substitution  $\theta$

○ That is: a logic programming problem!

# Preparing P

- Automatically close  $P$  under reflexivity & transitivity:
  - add the reflexive subtyping instance ( $a < a \ \backslash \ a$ )
  - add  $A < C$  whenever  $\{A < B, B < C\} \subseteq P$
- This way reflexivity and transitivity will not require special treatment during constraint simplification
- Good idea to close  $P$  under sub-superclass relation as well (e.g., add  $\text{Eq } A$  whenever  $\{\text{Ord } A\} \subseteq P$ )
- Also encode function co/contravariance by adding rule  $a \rightarrow b < a' \rightarrow b' \ \backslash \ b' < b, a < a'$

# Which solution?

- Logic programming engines usually stop as soon as one solution is found
- In general there might be many different solutions
- We want to calculate an improving substitution; i.e., the common factor present in all solutions
- Thus we let the algorithm back-track and compute all solutions to the given problem, then we return their intersection computed by **anti-unification**

Formally: find  $\sqcap \theta_i$  over all  $i$  such that  $P \vdash \theta_i Q$

# Anti-unification

- Anti-unification of substitutions:

$$\theta \sqcap \theta = \theta \quad \theta \sqcap \theta' = \theta'$$

$$\theta \sqcap \theta' = \theta(a) \sqcap \theta'(a) / a \text{ for all } a \text{ in } \text{dom}(\theta) \cap \text{dom}(\theta')$$

- Anti-unification of types:

$$\tau \sqcap \tau = \tau \quad a \sqcap a = a \quad (t \ s) \sqcap (t' \ s') = (t \sqcap t') (s \sqcap s')$$

$ts = f(t,s)$  if  $t \neq s$ , where  $f$  is a memo-function mapping every distinct type pair to a fresh type variable

# Algorithm complexity

- Theoretical: a whopping complexity of  $O(m^n)$ , where
  - $m$ : number of instance axioms and rules
  - $n$ : size of goal
- However, real life instances tend to be rather diverse
- Thus, a constraint is likely to be either
  - **specific enough** to make most solution attempts fail
  - or **general enough** to make intersection of solutions quickly reach the empty substitution
- Practical experience suggests typical complexity of  $O(mn)$

# Overlapping instances

- A partial order on the instance database
- Definition:  
 $(C T \setminus a) \preceq (C T' \setminus a)$  iff there is an  $S$  such that  
 $T = [S/a]T'$
- Pruning  $P$ : let  $P \setminus c = P \setminus \{ c' \text{ in } P \mid c \preceq c' \}$
- Always sort  $P$  according to  $\preceq$ . Then, whenever a fact/rule  $c$  succeeds in finding a solution, use only  $P \setminus c$  when searching for further solutions

# Forcing default solutions

- The improvement algorithm "in overdrive"
- Simply use an extended  $\leq$  that also includes the defaults:
  - declared default relation between typeclasses
  - reflexive instance  $\leq$  all other instances
  - lower bounds  $\leq$  upper bounds

# After improvement

- After applying the improving substitution, the resulting Q can be reduced using ordinary Haskell methods (although cyclic subtype constraints might still remain)
- As a separate step, one may also **check Q as if it were a top-level instance set** and see what errors are found
- This will, without any further
  - identify any overlaps in Q (can be removed)
  - identify any sub-superclass implication in Q (same)
  - identify any subtype cycle in Q (gives another improving substitution)

# Conclusion

- Subtyping can be conveniently added to a polymorphic type system with type reconstruction and overloading
- The subtype relation must be treated as a built-in type-class, accompanied by special properties
- Constraint simplification is generically defined as an logic programming algorithm that just slightly generalizes three concepts already found in Haskell (+ extensions):
  - improvement
  - overlapping instances
  - defaulting
- Try it out using the Timber compiler!