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 form the type of ++
twice g x = g (g x)
twice :: (a -> a) -> a -> a    -- a polymorphic type!
```

- Inferred types in Haskell are <u>principal types</u>; 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 <u>vital</u> 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 agains 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 ($A \to B < A' \to B'$ 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 <u>requirements</u> on instances
  - Prohibits an arbitrary instance structure
- They also offer extra <u>assumptions</u> 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 -> b \\ a < A, B < b) can be rewritten as A -> B
- Illustrates human preference for syntactically smaller types!
- Justified by theorem that shows

  $P \vdash$ (a -> b \\ a < A, B < b) < T

  iff

  $P \vdash$ A -> B < T

Simplification algorithms

- Has been extensively studied for subtyping in isolation, but not incorporated in a full language
- Combintion 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

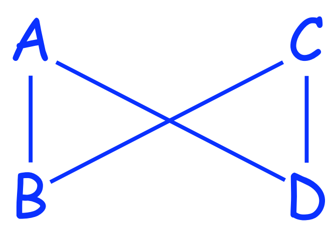
- Consider

  min x y = **if** less x y **then** x **else** y
  less :: A -> A -> Bool

- In Haskell, A -> A -> A is a principal type for min
- But if subtyping is added the principal type becomes

  min :: a -> b -> c \\ 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 <u>in tandem</u>
  - Simplifies <u>first</u>, 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 <u>improved</u> by substitution Char/a (and then removed because it becomes a tautology)
- An improving substitution replaces variables in a constraint <u>without making it less general</u>
- 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 the former instance having preference over the latter
- Not accepted by Haskell, but common as an extension

Overlapping instances

- Consider the type `a -> Int \\ a < A, Eq a` in the context

```
struct B < A where ...
struct C < B where ...
instance Eq B where ...
```

- We have many implicit instances that can solve `a < A`: `A<A`, `B<A`, `C<A`, ...
- But since `a` is in a contravariant position, we'd like `A<A` to <u>have preference</u> 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 -> Int \\ B < A, Eq B` (reduces to `B -> Int`)

Defaulting

- Consider

  f x = **if** show (1+6) == "7" **then** x **else** x++x
  f :: String -> String \\ Num b, Show b

- How choose this b? The choice obviously determines whether show (1+6) == "7" may succeed
- Lesson from Haskell: the type of f is <u>ambiguous</u> and cannot be accepted. But one may declare a <u>default</u> instance for Num, say Num Int, to use in such cases
- Substituting Int for b int the type above can be viewed as an aggressive form of improvement, that takes the "artificial" overlap Num Int ≼ 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

  `twice f x = f (f x)`

  Before defaulting `f :: (a -> b) -> a -> b \\ b < a`

  After defaulting `f :: (a -> a) -> a -> 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] \\ Eq a, Num Int, A < B, ...
    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, ...
- Formally: $P \vdash \theta Q$ for some substitution θ
- That is: a logic programming problem!

Preparing P

- Automatically close P under reflexicity & transitivity:
  - add the reflexive subtyping instance (a < a \\ a)
  - add A < C whenever {A < B,B < C} ⊆ 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 Eq A whenever {Ord A} ⊆ P)
- Also encode function co/contravariance by adding rule a -> b < a' -> b' \\ 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 caclulate an improving substitution; i.e., the common factor present in all solutions
- Thus we let the algoritm 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:

  $\theta \sqcap 0 = \theta \qquad 0 \sqcap \theta' = \theta'$

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

- Anti-unification of types:

  $T \sqcap T = T \qquad a \sqcap a = a \qquad (t\ s) \sqcap (t'\ s') = (t \sqcap t')\ (s \sqcap s')$

  $ts = f(t,s)$ 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 \\ a) ≼ (C T' \\ a) iff there is an S such that
  T = [S/a]T'
- Pruning P: let P\c = P \ { c' in P | c ≼ c' }
- Always sort P according to ≼. Then, whenever a fact/rule c succeeds in finding a solution, use only P\c when searching for further solutions

Forcing default solutions

- The improvement algorithm "in overdrive"
- Simply use an extended $\preccurlyeq$ that also includes the defaults:
  - declared default relation between typeclasses
  - reflexive instance $\preccurlyeq$ all other instances
  - lower bounds $\preccurlyeq$ 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 (+ extendsions):
  - improvement
  - overlapping instances
  - defaulting
- Try it out using the Timber compiler!