CSE-505: Programming Languages

Lecture 24 — Bounded Polymorphism

Zach Tatlock
2015

Revenge of Type Variables

Sorted lists in ML (partial):

```
type 'a slist
make : ('a -> 'a -> int) -> 'a slist
cons : 'a slist -> 'a -> 'a slist
find : 'a slist -> 'a -> 'a option
```

Getting by with OOP subtyping:

```
interface Cmp  { Int f(Object,Object); }
class SList {
  ... some field definitions ...
  constructor (Cmp x)   {...}
  Slist  cons(Object x) {...}
  Object find(Object x) {...}
}
```

Wanting Type Variables

Will downcast (potential run-time exception) the arguments to `f` and the result of `find`

We are not enforcing list-element type-equality

OOP-style subtyping is no replacement for parametric polymorphism; we can have both:

```
interface Cmp<'a>  { Int f('a,'a); } // Cmp not a type

class SList<'a> { // SList not a type (SList<Int> e.g. is)
  ... some field definitions (can use type 'a) ...

  constructor (Cmp<'a> x) {...}
  Slist<'a> cons('a x)    {...}
  'a        find('a x)    {...}
}
```

Same Old Story

- Interface and class declarations are *parameterized*; they produce types
- The constructor is polymorphic
  - For all `T`, given a `Cmp<T>`, it makes a `SList<T>`
- If `o` has type `SList<T>`, its `cons` method:
  - Takes a `T`
  - Returns a `SList<T>`

No more downcasts; the best of both worlds

Complications

“Interesting” interaction with overloading and multimethods

```
class B {
  unit f(C<Int>    x) {...}
  unit f(C<String> x) {...}
}
class C<'a> { unit g(B x) { x.f(self); } }
```

For `C<T>` where `T` is neither `Int` nor `String`, can have no match

- Cannot resolve static overloading at compile-time without code duplication and no abstraction (C++)
- To resolve overloading or multimethods at run-time, need run-time type information *including the instantiation* `T` (C#)
- Could disallow such overloading (Java)
- Or could just reject this sort of call as unresolvable (?)

Wanting bounds

There are compelling reasons to *bound* the instantiation of type variables

Simple example: Use at supertype without losing that it's a subtype

```
interface I { unit print(); }
class Logger< 'a <: I >  { // must apply to subtype of I
   'a item;
   'a get_it() { syslog(item.print()); item }
}
```

Without polymorphism or downcasting, client could only use `get_it` result for printing

Without bound or downcasting, `Logger` could not print

Issue isn't special to OOP

Fancy Example from "A Theory of Objects" Abadi/Cardelli

With forethought, bounds can avoid some subtyping limitations

```
interface Omnivore  { unit eat(Food); }
interface Herbivore { unit eat(Veg);  } // Veg <= Food
```

Allowing `Herbivore`$\leq$`Omnivore` could make a vegetarian eat meat (unsound)! But this works:

```
interface Omnivore< 'a <: Food >  { unit eat('a); }
interface Herbivore< 'a <: Veg > { unit eat('a); }
```

If `Herbivore<T>` is legal, then `Omnivore<T>` is legal *and* `Herbivore<T>`$\leq$`Omnivore<T>` !

Useful for `unit feed('a food, Omnivore<'a> animal) {...}`

Bounded Polymorphism

This "bounded polymorphism" is useful in any language with universal types and subtyping. Instead of $\forall\alpha.\tau$ and $\Lambda\alpha.e$, we have $\forall\alpha < \tau'.\tau$ and $\Lambda\alpha < \tau'.e$:

- Change Δ to be a list of bounds ($\alpha < \tau$) instead of a set of type variables
- In e you can subsume from α to τ'
- $e_1[\tau_1]$ typechecks when τ_1 "satisfies the bound" in type of e_1

One limitation: When is $(\forall\alpha_1{<}\tau_1.\tau_2) \leq (\forall\alpha_2{<}\tau_3.\tau_4)$?

- Contravariant bounds and covariant bodies assuming bound are sound, but makes subtyping undecidable
- Requiring invariant bounds and covariant bodies regains decidability, but obviously allows less subtyping