CSEP505: Programming Languages

Lecture 8: Types Wrap-Up; Object-Oriented Programming

Dan Grossman

Spring 2006

Today's plan

Three last things about types

1. Type inference (ML-style)
2. Bounded polymorphism (subtyping & forall)
3. Polymorphic references (must avoid them)

Then: Object-oriented programming

- What's different
- How do we define it (will stay informal)
- What else could we do
- What are types for (and how aren't they classes)

The ML type system

Have already defined (most of) the ML type system

- System F with 4 restrictions
- (Plus bells, whistles, and a module system)
- (No subtyping or overloading)

Semi-revisionist history; this type system is what a simple, elegant *inference algorithm* supports

- Called “Algorithm W” or “Hindley-Milner inference”
- In theory, inference “fills out explicit types”
- In practice, often merge inference and checking

An algorithm best understood by example…

Example #1

```
let f x =
  let (y,z) = x in
  (abs y) + z
```

Example #2

```
let rec sum lst =
 match lst with
    [] ->  0
  |hd::tl -> hd + (sum tl)
```

Example #3

```
let rec length lst =
 match lst with
    [] ->  0
  |hd::tl -> 1 + (length tl)
```

Example #4

```
let compose f g x = f (g x)
```

More generally

- Infer each let-binding or toplevel binding in order
  - Except for mutual recursion (do all at once)
- Give each variable and subexpression a fresh "constraint variable"
- Add constraints for each subexpression
  - Very similar to typing rules
- Circular constraints fail (so `x x` never typechecks)
- After inferring let-body, *generalize* (unconstrained constraint variables become type variables)

In practice

- Described algorithm as
  - "generate a ton of constraints"
  - "solve them" (stop on failure, generalize at let)
- In practice, faster & equivalent "unification" algorithm:
  - Each constraint variable is a "pointer" (a reference)
  - Equality constraints become indirection
  - Can "shorten paths" eagerly
- Value restriction done separately

Today's plan

Three last things about types

1. Type inference (ML-style)
2. Bounded polymorphism (subtyping & forall)
3. Polymorphic references (must avoid them)

Then: Object-oriented programming

- What's different
- How do we define it (will stay informal)
- What else could we do
- What are types for (and how aren't they classes)

Why bounded polymorphism

Could 1 language have $\tau_1 \quad \tau_2$ and $\forall\alpha.\tau$?

- Sure! They're both useful and complementary
- But how do they *interact*?

1. When is $\forall\alpha.\tau_1 \quad \forall\beta.\tau_2$?
2. What about bounds?

```
let dblL1 x = x.l1 <- x.l1*2; x
```

- Subtyping: `dblL1 : {l1=int}->{l1=int}`
  - Can pass subtype, but result type loses a lot
- Polymorphism: `dblL1 : ∀α.α->α`
  - Lose nothing, but body doesn't type-check

What bounded polymorphism

The type we want: `dblL1 : ∀α {l1=int}.α->α`

Java and C# generics have this (different syntax)

Formally:

- Syntax of types (`∀α τ.τ`) , generics (`Λα τ.e`) and contexts (Γ, `α τ`) changes
- Typing rule for `Λα τ.e` "introduces bound"
- Typing rule for `e [τ]` "requires bound is satisfied"

Subtyping revisited

When is ∀α τ1.τ2 ∀α τ3.τ4 ?

- Note: already "alpha-converted" to same type variable

Sound answer:

- Contravariant bounds (τ3 τ1)
- Covariant bodies (τ2 τ4)

Problem: Makes subtyping undecidable (surprised many)

Common workarounds:

- Require invaraint bounds (τ3 τ1 and τ1 τ3)
- Some ad hoc approximation

Today's plan

Three last things about types

1. Type inference (ML-style)
2. Bounded polymorphism (subtyping & forall)
3. Polymorphic references (must avoid them)

Then: Object-oriented programming

- What's different
- How do we define it (will stay informal)
- What else could we do
- What are types for (and how aren't they classes)

Polymorphic references

A sound type system cannot accept this program:

```
let x = ref [] in
x := 1::[];
match !x with _ -> () | hd::_ -> hd ^ "gotcha"
```

But it would assuming this interface:

```
type 'a ref
val ref : 'a -> 'a ref
val !   : 'a ref -> 'a
val :=  : 'a ref -> 'a -> unit
```

Solutions

Must restrict the type system, but many ways exist

1. "Value restriction": `ref []` cannot have a polymorphic type (syntactic look for `ref` not enough)
2. Let `ref []` have type `(`$\forall$`α.α list)ref` (not useful and not an ML type)
3. Tell the type system "mutation is special," not "just another polymorphic library interface"

Today's plan

Three last things about types

1. Type inference (ML-style)
2. Bounded polymorphism (subtyping & forall)
3. Polymorphic references (must avoid them)

Then: Object-oriented programming

- What's different
- How do we define it (will stay informal)
- What else could we do
- What are types for (and how aren't they classes)

OOP the sales pitch

OOP lets you:

1. Build extensible software concisely
2. Exploit an intuitive analogy between interaction of physical entities and interaction of software pieces

It also:

- Raises tricky semantic and style issues that require careful investigation
- Is more complicated than functions
  - Does *not mean* it's worse

So what is it?

OOP "looks like this", but what is the essence

```
class Pt1 extends Object {
 int x;
 int  get_x() { x }
 unit set_x(int y) { self.x = y }
 int  distance(Pt1 p) { p.get_x() - self.get_x() }
 constructor() { x = 0 }
}
class Pt2 extends Pt1 {
 int y;
 int  get_y() { y }
 unit get_x() { 34 + super.get_x() }
 constructor() { super(); y = 0 }
}
```

OOP can mean many things

- An ADT (private fields)
- Subtyping
- Inheritance: method/field extension, method override
- Implicit self/this
- Dynamic dispatch
- All the above in one (class) definition

Design question: Better to have small orthogonal features or one "do it all" feature?

Anyway, let's consider how "unique to OO" each is…

OO for ADTs

Object/class *members* (fields, methods, constructors) often have *visibilities*

What code can invoke a member/access a field?

- Methods of the same object?
- Methods defined in same class?
- Methods defined in a subclass?
- Methods within another "boundary" (package, assembly, friend-class, …)
- Methods defined anywhere?

What can we do with just class definitions?

Subtyping for hiding

- As seen before, can use upcasts to "hide" members
  - Modulo downcasts
  - Modulo binary-method problems
- With just classes, upcasting is limited
- With interfaces, can be more selective

```
interface I { int distance(Pt1 p); }
class Pt1 extends Object {
 …
 I f() { self }
 …
}
```

Records of functions

If OOP = functions + private state, we already have it

- But it's more (e.g., inheritance)

```
type pt1 = {get_x    : unit -> int;
            set_x    : int  -> unit;
            distance : pt1  -> int}
let pt1_constructor () =
  let x = ref 0 in
  let rec self = {
   get_x    = (fun() -> !x);
   set_x    = (fun y -> x := y);
   distance = (fun p -> p.get_x() + self.get_x())
  } in
  self
```

Subtyping

Many class-based OO languages purposely "confuse" classes and types

- If C is a class, then C is a type
- If C extends D (via declaration) then C D
- Subtyping is reflexive and transitive

Novel subtyping?

- New members in C just width subtyping
- "Nominal" (by name) instead of structural
- What about override…

Subtyping, continued

- If C extends D, overriding m, what do we need:
  - Arguments contravariant (assume less)
  - Result covariant (provide more)
- Many "real" languages are more restrictive
  - Often in favor of static overloading
- Some languages try to be more flexible
  - At expense of run-time checks/casts

Good we studied this in a simpler setting

Inheritance & override

Subclasses:

- inherit superclass's members
- can override methods
- can use **`super`** calls

Can we code this up in Caml?

- No because of field-name reuse and subtyping, but ignoring that we can get *close…*

Almost OOP?

```
let pt1_constructor () =
  let x = ref 0 in
  let rec self = {
    get_x    = (fun() -> !x);
    set_x    = (fun y -> x := y);
    distance = (fun p -> p.get_x() + self.get_x())
  } in self
(* note: field reuse precludes type-checking *)
let pt2_constructor () = (* extends Pt1 *)
  let r = pt1_constructor () in
  let y = ref 0 in
  let rec self = {
    get_x    = (fun() -> 34 + r.get_x());
    set_x    = r.set_x;
    distance = r.distance;
    get_y    = (fun() -> !y);
  } in self
```

Problems

Small problems:

- Have to change pt2_constructor whenever pt1_constructor changes
- But OOPs have tons of "fragile base class" issues too
  - Motivates C#'s version support
- No direct access to "private fields" of super-class

Big problem:

- Distance method in a pt2 doesn't behave how it does in OOP
- We do not have late-binding of self (i.e., dynamic dispatch)

The essence

Claim: Class-based objects are:

- So-so ADTs
- Same-old record and function subtyping
- Some syntactic sugar for extension and override

- A fundamentally different rule for what self maps to in the environment

More on late-binding

Late-binding, dynamic-dispatch, and open-recursion all related issues (nearly synonyms)

Simplest example I know:

```
let c1 () =
 let rec r = {
  even = (fun i -> if i=0 then true else r.odd (i-1));
  odd = (fun i -> if i=0 then false else r.even (i-1))
 } in r

let c2 () =
 let r1 = c1() in
 let rec r = {
  even = r1.even; (* still O(n) *)
  odd = (fun i -> i % 2 == 1)
 } in r
```

More on late-binding

Late-binding, dynamic-dispatch, and open-recursion all related issues (nearly synonyms)

Simples example I know:

```
class C1 {
 int even(int i) {if i=0 then true else odd (i-1))}
 int odd(int i) {if i=0 then false else even (i-1))}
}

class C2 {
 /* even is now O(1) */
 int even(int i) { super.even(i) }
 int odd(int i) {i % 2 == 1}
}
```

The big debate

Open recursion:

- Code reuse: improve even by just changing odd
- Superclass has to do less extensibility planning

Closed recursion:

- Code abuse: break even by just breaking odd
- Superclass has to do more abstraction planning

Reality: Both have proved very useful; should probably just argue over “the right default”

Our plan

- Dynamic dispatch is the essence of OOP
- How can we define/implement dynamic dispatch?
  - Basics, not super-optimized versions (see P501)
- How do we use/misuse overriding?
- Why are subtyping and subclassing separate concepts worth keeping separate?

Defining dispatch

Methods "compile down" to functions taking self as an extra argument

- Just need self bound to "the right thing"

Approach #1:

- Each object has 1 code pointer per method
- For new C() where C extends D:
  - Start with code pointers for D (inductive definition!)
  - If C adds m, add code pointer for m
  - If C overrides m, change code pointer for m
- Self bound to the object

Defining dispatch

Methods "compile down" to functions taking self as an extra argument

- Just need self bound to "the right thing"

Approach #2:

- Each object has 1 run-time tag
- For new C() where C extends D:
  - Tag is C
- Self bound to the object
- Method call to m reads tag, looks up (tag,m) in a global table

Which approach?

- The two approaches are very similar
  - Just trade space for time via indirection
- vtable pointers are a fast encoding of approach #2
- This “definition” is low-level, but with overriding simpler models are probably wrong

Our plan

- Dynamic dispatch is the essence of OOP
- How can we define/implement dynamic dispatch?
  - Basics, not super-optimized versions (see P501)
- How do we use/misuse overriding?
  - Functional vs. OO extensibility
- Why are subtyping and subclassing separate concepts worth keeping separate?

Overriding and hierarchy design

- Subclass writer decides what to override
  - Often-claimed, unchecked style issue: overriding should *specialize behavior*
- But superclass writer typically knows what will be overridden

- Leads to notion of abstract methods (must-override)
  - Classes w/ abstract methods can't be instantiated
  - Does not add expressiveness
  - Adds a static check

Overriding for extensibility

```
class Exp { /* a PL example; constructors omitted */
 abstract Exp interp(Env);
 abstract Typ typecheck(Ctxt);
 abstract Int toInt();
}
class IntExp extends Exp {
 Int i;
 Value interp(Env e) { self }
 Typ typecheck(Ctxt c) { new IntTyp() }
 Int toInt() { i }
}
class AddExp extends Exp {
 Exp e1; Exp e2;
 Value interp(Env e) {
   new IntExp(e1.interp(e).toInt().add(
              e2.interp(e).toInt())) }
 Int toInt() { throw new BadCall() }
 /* typecheck on next page */
}
```

Example cont'd

- We did addition with "pure objects"
  - Int has a binary add method
- To do AddExp::typecheck the same way, assume equals is defined appropriately (structural on Typ):

```
Type typecheck(Ctxt c) {
  e1.typecheck(c).equals(new IntTyp()).ifThenElse(
  e2.typecheck(c).equals(new IntTyp()).ifThenElse(
  (fun () -> new IntTyp()),
  (fun () -> throw new TypeError())),
  (fun () -> throw new TypeError()))
}
```

- Pure "OOP" avoids instanceof IntTyp and if-statements
  - (see Smalltalk, Ruby, etc. for syntactic sugar)

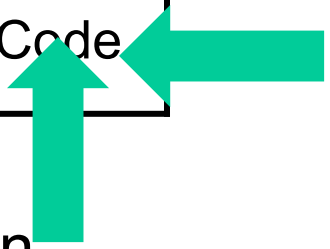
More extension

- Now suppose we want MultExp
  - No change to existing code, unlike ML!
  - In ML, can "prepare" with "Else" constructor
- Now suppose we want a toString method
  - Must change all existing classes, unlike ML!
  - In OOP, can "prepare" with "Visitor" pattern
- A language (paradigm) may be good for some dimensions of extensibility; probably not all!
  - Active research area (including at UW!)

The Grid

- You know it's an important idea if I take the time to draw a picture ☺

	interp	typecheck	toString	…
IntExp	Code	Code	Code	Code
AddExp	Code	Code	Code	Code
MultExp	Code	Code	Code	Code
…	Code	Code	Code	Code



1 new class

1 new function

Back to MultExp

- Even in OOP, MultExp is easy to add, but you'll *copy* the typecheck method of AddExp
- Or maybe AddExp extends MultExp, but it's a *kludge*
- Or maybe *refactor* into BinaryExp with subclasses AddExp and MultExp
  - So much for not changing existing code
  - Awfully heavyweight approach to a helper function

Our plan

- Dynamic dispatch is the essence of OOP
- How can we define/implement dynamic dispatch?
  - Basics, not super-optimized versions (see P501)
- How do we use/misuse overriding?
  - Functional vs. OO extensibility
- Why are subtyping and subclassing separate concepts worth keeping separate?

Subtyping vs. subclassing

- Often convenient confusion: C a subtype of D if and only if C a subclass of D

- But more subtypes are sound
  - If A has every field and method that B has (at appropriate types), then subsume B to A
  - Interfaces help, but require explicit annotation

- And fewer subtypes could allow more code reuse…

Non-subtyping example

`Pt2` ≤ `Pt1` is unsound here:

```
class Pt1 extends Object {
 int x;
 int  get_x() { x }
 Bool compare(Pt1 p) { p.get_x() == self.get_x() }
}
class Pt2 extends Pt1 {
 int y;
 int  get_y() { y }
 Bool compare(Pt2 p) { // override
      p.get_x() == self.get_x()
   && p.get_y() == self.get_y() }
}
```

What happened

- Could inherit code without being a subtype
- Cannot always do this (what if get_x called self.compare with a Pt1)

  Possible solutions:
  - Re-typecheck get_x in subclass
  - Use a really fancy type system
  - Don't override compare

- Moral: Not suggesting "subclassing not subtyping" is useful, but the concepts of inheritance and subtyping are orthogonal

Now what?

- That's basic class-based OOP
  - Not all OOPLs use classes
    (Javascript, Self, Cecil, …)
    (may show you some cool stuff time permitting)
- Now some "fancy" stuff
  - Typechecking
  - Multiple inheritance; multiple interfaces
  - Static overloading
  - Multimethods
  - Revenge of bounded polymorphism

Typechecking

We were sloppy:

talked about types without “what are we preventing”

1. In pure OO, stuck if v.m(v1,…,vn) and v has no m method (taking *n* args)
   - “No such method”
2. Also if ambiguous (multiple methods with same name; no best choice)
   - “No best match”

Multiple inheritance

Why not allow `C extends C1,…,Cn {…}`

(and `C`≤`C1`, …, `C`≤`Cn`)

What everyone agrees on: C++ has it, Java doesn't

We'll just consider some problems it introduces and how (multiple) interfaces avoids some of them

Problem sources:

1. Class hierarchy is a dag, not a tree
2. Type hierarchy is a dag, not a tree

Method-name clash

What if C extends C1, C2 which both define m?

Possibilities:

1. Reject declaration of C
   - Too restrictive with diamonds (next slide)
2. Require C overrides m
   - Possibly with *directed resends*
3. "Left-side" (C1) wins
   - Question: does cast to C2 change what m means?
4. C gets both methods (implies incoherent subtyping)
5. Other? (I'm just brainstorming)

Diamonds

- If C extends C1, C2 and C1, C2 have a common (transitive) superclass D, we have a diamond
  - Always have one with multiple inheritance and a topmost class (Object)
- If D has a field f, does C have one field or two?
  - C++ answer: yes ☺
- If D has a method m, see previous slide.
- If subsumption is coercive, we may be incoherent
  - Multiple paths to D

Implementation issues

- Multiple-inheritance semantics often muddied by wanting "efficient member lookup"
  - If "efficient" is compile-time offset from self pointer, then multiple inheritance means subsumption must "bump the pointer"
  - Roughly why C++ has different sorts of casts

- Preaching: Get the semantics right first

Digression: casts

A “cast” can mean too many different things (cf. C++):

Language-level:

- Upcast (no run-time effect)
- Downcast (failure or no run-time effect)
- Conversion (key question: *round-tripping*)
- “Reinterpret bits” (not well-defined)

Implementation level

- Upcast (see last slide)
- Downcast (check the tag)
- Conversion (run code to make a new value)
- “Reinterpret bits” (no effect)

Least supertypes

[Related to homework 4 extra credit]

For e1 ? e2 : e3

- e2 and e3 need the same type
- But that just means a common supertype
- But which one? (The least one)
  - But multiple inheritance means may not exist!

Solution:

- Reject without explicit casts

Multiple inheritance summary

1. Method clashes (same method m)
2. Diamond issues (fields in top of diamond)
3. Implementation issues (slower method lookup)
4. Least supertypes (may not exist)

Multiple interfaces (type without code or fields) have problems (3) and (4)

(and 3 isn't necessarily a problem)

Now what?

- That's basic class-based OOP
  - Not all OOPLs use classes
    (Javascript, Self, Cecil, …)
    (may show you some cool stuff time permitting)
- Now some "fancy" stuff
  - Typechecking
  - Multiple inheritance; multiple interfaces
  - Static overloading
  - Multimethods
  - Revenge of bounded polymorphism

Static overloading

- So far: Assume every method name unique
  - (same name in subclass meant override; required subtype)
- Many OO languages allow same name, different argument types:

```
A f(B b) {…}
C f(D d, E e) {…}
F f(G g, H h) {…}
```

- Changes method-lookup definition for e.m(e1,…en)
  - Old: lookup m via (run-time) class of e
  - New: lookup m via (run-time) class of e *and* (compile-time) types of e1,…,en

Ambiguity

Because of subtyping, multiple methods can match!

"Best match" rules are complicated. One rough idea:

- Fewer subsumptions is better match
- If tied, subsume to *immediate supertypes* & *recur*

Ambiguities remain

1. `A f(B)` or `C f(B)` (usually disallowed)
2. `A f(B)` or `A f(C)` and `f(e)` where `e` has a subtype of `B` and `C` but `B` and `C` are incomparable
3. `A f(B,C)` or `A f(C,B)` and `f(e1,e2)` where `e1` and `e2` have type `B` and `B` $\leq$ `C`

Now what?

- That's basic class-based OOP
  - Not all OOPLs use classes
    (Javascript, Self, Cecil, …)
    (may show you some cool stuff time permitting)
- Now some "fancy" stuff
  - Typechecking
  - Multiple inheritance; multiple interfaces
  - Static overloading
  - Multimethods
  - Revenge of bounded polymorphism

Multimethods

Static overloading just saves keystrokes via shorter method names

- Name-mangling on par with syntactic sugar

Multiple (dynamic) dispatch (a.k.a. multimethods) much more interesting: Method lookup based on (run-time) classes of all arguments

A natural generalization: "receiver" no longer special

So may as well write **`m(e1,…,en)`**

instead of **`e1.m(e2,…,en)`**

Multimethods example

```
class A { int f; }
class B extends A { int g; }
Bool compare(A x, A y) { x.f==y.f }
Bool compare(B x, B y) { x.f==y.f && x.g==y.g }
Bool f(A x, A y, A z) { compare(x,y) &&
                        compare(y,z) }
```

- `compare(x,y)` calls first version unless both arguments are B's
  - Could add "one of each" methods if you want different behavior

Pragmatics; UW

Not clear where multimethods should be defined

- No longer “belong to a class” because receiver not special

Multimethods are “more OO” because dynamic-dispatch is the essence of OO

Multimethods are “less OO” because without distinguished receiver the “analogy to physical objects” is less

UW CSE a multimethods leader for years (Chambers)

Revenge of ambiguity

- Like static overloading, multimethods have “no best match” problems
- Unlike static overloading, the problem does not arise until run-time!

Possible solutions:

1. Run-time exception
2. Always define a best-match (e.g., Dylan)
3. A conservative type system

Now what?

- That's basic class-based OOP
  - Not all OOPLs use classes
    (Javascript, Self, Cecil, …)
    (may show you some cool stuff time permitting)
- Now some "fancy" stuff
  - Typechecking
  - Multiple inheritance; multiple interfaces
  - Static overloading
  - Multimethods
  - Revenge of bounded polymorphism

Still want generics

OO subtyping no replacement for parametric polymorphism

So have both

Example:

```
/* 3 type constructors (e.g., Int Set a type) */
interface 'a Comparable { Int f('a,'a); }
interface 'a Predicate { Bool f('a); }
class 'a Set {
 …
 constructor('a Comparable x){…}
 unit add('a x){…}
 'a Set functional_add('a x){…}
 'a find('a Predicate x) {…}
}
```

Worse ambiguity

"Interesting" interaction with overloading or multimethods

```
class B {
 Int f(Int C x){1}
 Int f(String C x){2}
 Int g('a x) { self.f(x) }
}
```

Whether match is found depends on instantiation of `'a`

Cannot resolve static overloading at compile-time without code duplication

At run-time, need run-time type information

- Including instantiation of type constructors
- Or restrict overloading enough to avoid it

Wanting bounds

As expected, with subtyping and generics, want bounded polymorphism

Example:

```
interface I { unit print(); }
class ('a≤I) Logger {
 'a item;
 'a get() { item.print(); item }
}
```

w/o polymorphism, **`get`** would return an **`I`** (not useful)

w/o the bound, **`get`** could not **`send`** print to **`item`**

Fancy example

With forethought, can use bounds to avoid some subtyping limitations

(Example lifted from Abadi/Cardelli text)

```
/* Herbivore1 ≤ Omnivore1 unsound */
interface Omnivore1  { unit eat(Food); }
interface Herbivore1 { unit eat(Veg); }
/* T Herbivore2 ≤ T Omnivore2 sound for any T */
interface ('a≤Food) Omnivore2 { unit eat('a); }
interface ('a≤Veg)  Herbivore2 { unit eat('a); }
/* subtyping lets us pass herbivores to feed
   but only if food is a Veg */
unit feed('a food, 'a Omnivore animal) {
   animal.eat(food);
}
```