Structure and Abstraction in HOT Languages

A Comparison of Polymorphism, Modules, and Objects

David MacQueen

Bell Labs

How OO and FP Support Adaptable Programming

Parameterization, Subtyping, Iinheritance

David MacQueen

Bell Labs

Hot Languages

- higher-order
  - binding code and data
- typed
  - static type checking (for error detection)
  - specifying structure of data and program interfaces
  - formalism for analyzing language constructs
- functional and object-oriented (OO) paradigms

Functional Paradigm

- value-oriented programming -- computing by constructing
- functions as data (closures)
- parameterization at value, type, and module level
- "algebraic" data types for unions, records for products
- pure vs impure functional languages
  - Haskell pure, lazy
  - ML impure (imperative), strict

OO Paradigm

- state-oriented, imperative programming
- objects as data (instance state + method code)
- subtyping (based on subclassing)
- subclasses for unions, objects for products
- pure vs impure object oriented languages
  - Smalltalk and Java pure (everything an object)
  - Eiffel, Modula3, C++, Object Pascal, etc. impure

Theme: Static typing good

Static typing, based on a sound type system ("well-typed programs do not go wrong") is a basic requirement for robust system programming.

Why Types?

- safety: well typed programs do not go wrong
- a language and disciple for design of data structures and program interfaces
- support for separate development via precise interfaces
- properties and invariants verified by the compiler ("a priori guarantee of correctness")
- support for orderly evolution of software
  - consequences of changes can be traced

Types and FP

- In FP, types determine behavior to a much greater extent than in conventional procedural or OO languages.
- OO languages use “modeling languages” like UML. For FP, the type system serves as the modeling language.
- In FP, the extent of program behavior covered by type checking (in Greg Nelson’s terminology) is greater than for imperative programming.

A type-based approach

Evaluating language designs on the based of their type systems

- type systems provide a common framework for comparison (synthesis?) of designs
- type systems are a major factor determining the flexibility and expressiveness of language designs
- type systems have been thoroughly studied and provide a connection between language theorists and language designers and users

Theme: Program adaptation

To support code reuse, we must be able to build *general purpose* program units and then adapt them to particular uses.

How do we build *generic* units, and how do we specialize them, or derive specialized versions?

Overview

- Introduction: parameterization, subtyping, inheritance
- Review of Type Systems
- Functional Programming (ML)
  - functions, polymorphism, modules
- Object-Oriented programming
  - reconstruction from first principles
- Comparison of OO and FP
- Synthesis of OO and FP

Adaptation Mechanisms

- parameterization
  - values: procedures and functions
  - types: parametric polymorphism
- subtype polymorphism
- dynamic dispatch
- implementation inheritance: extension and update of functionality
- information hiding, abstraction, modularity

Procedural abstraction

- Most basic form of reuse
- Name a block of code, call it repeatedly
- Preserves env. of definition (closure)

```
procedure P() =
begin
  x := x+1;
  y := y*x;
end

var x: int; ... P(); ... P(); ...
```

data parameters: 1st order functions

- abstract over names of values
- specialize by application to different arguments

data parameters: sorting lists

```
fun sort(l: int list) =
    (if null l ... x < y ...)
                          -- code to sort l

sort : int list -> int list

sort [2, 17, -3, 5]  =>  [-3, 2, 5, 17]
```

this can sort different lists of ints, but only with respect to fixed, hardwired ordering <

function parameters: higher-order fns

- abstract over names of functions
- specialize by passing functions

```
fun sort (< : int*int->bool)(l: int list) =
    (if null l … x < y …)

sort : (int*int->bool) -> int list
         -> int list

sort (>) [2, 17, -3, 5]
  =>  [-3, 2, 5, 17]
```

Type parameters: polymorphic functions

- abstract over names of types
- specialize by passing a type argument

Polymorphic sort

```
fun sort [t](<: t*t->bool)(l: t list) =
          (if null l ... x < y ...)
```

```
sort : ∀t.(t*t->bool) -> t list -> t list
```

sort[int](>$_{int}$)[2, 17, -3, 5]
  => [17, 5, 2, -3]

sort[string](<$_{string}$)[“bob”, “alice”, “tom”]
  => [“alice”, “bob”, “tom”]

Type parameters: polymorphic sort

Note that if the element type is a parameter, the comparison function < must also be a parameter.

```
fun sort [t](<: t*t->bool)(l: t list) =
     (if null l ... x < y ...)

sort : ∀t.(t*t->bool) -> t list -> t list
```

Interface parameters

- Interface specifies a set of components (types and values), defining the type of a module.
- Abstract a module over an interface.

```
signature Order =
sig type t
    val < : t * t -> bool
end

functor Sort(X: Order) =
struct
   fun sort(l: X.t list) = ...
end
```

Subtype polymorphism

- The subtype relation

  `A <: B` "every *A* is a B"

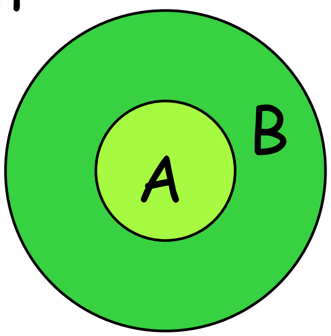
  `Nat <: Int, Ascii <: Unicode`

- Subsumption Rule

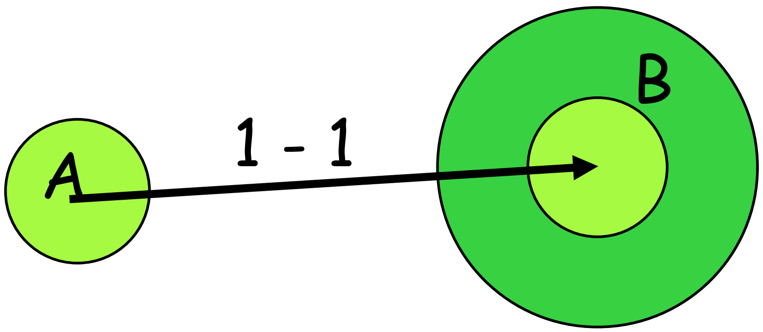
$$\frac{\mathtt{A <: B} \qquad \mathtt{x: A}}{\mathtt{x: B}}$$

Picturing subtyping: A <: B

subset



embedding



Sources of subtyping

Mathematics: **`Nat <: Int`**

Machines: **`UnsignedInt32 <: Int32 ?`**

Mathematics: **`Int <: Real`**

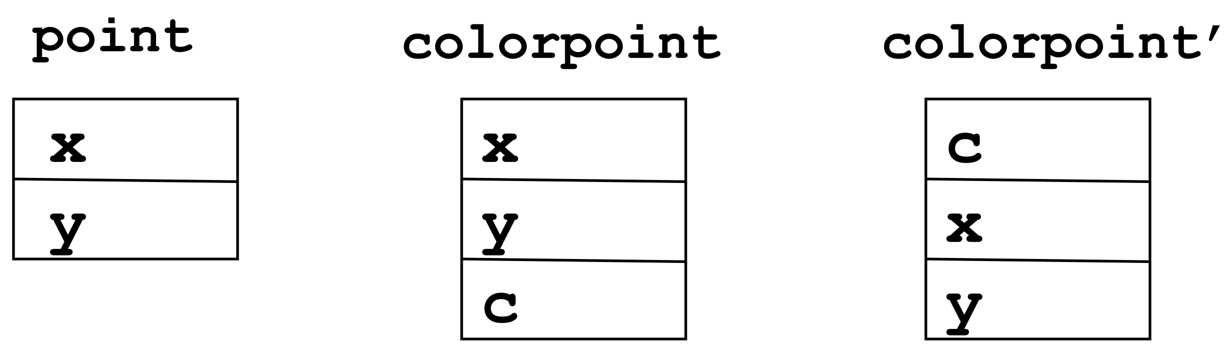
Machines: **`Int32 <: Float64`**
(representation shift)

Sources of subtyping: records

point = {x: int, y: int}

colorpoint = {x: int, y: int, c: color}

colorpoint <: point



Using Subtyping

```
move : Point -> Point

ColorPoint <: Point

cp : ColorPoint

cp : Point

move cp : Point
```

Implementation Inheritance

- A principle adaptation mechanism of OO languages
  - class-based: subclasses
  - object-based: cloning + extension + method update

- Based on “open recursion”

  replacing members of a family of mutually recursive functions

Inheritance

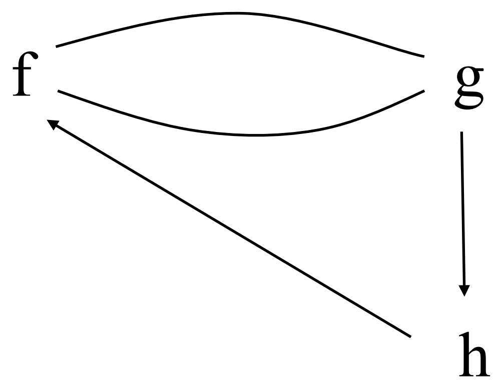
A class and a derived subclass

```
class Counter(n: Int)
  var x: Int = n
  method get() = x
  method add(y: Int) = x := x + y

class BCounter(max: int) inherits Counter
  method add(y: Int) =  (* override *)
    if get()+y < max then x := get() + y
  method inc() =        (* extend *)
    if get() < max-1 then x := get() + 1
```

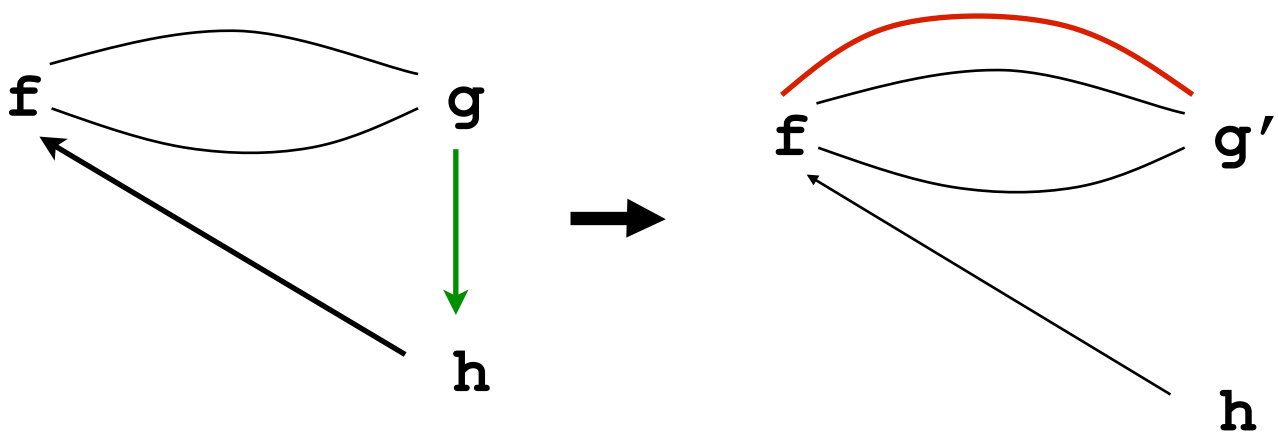
Open recursion

- conventional closed recursion



Open recursion

open recursion: replacing members of a recursive family of functions



Open recursion with function passing

```
fun f(x) = ... g(-) ... g(-) ...
fun g(x) = ... g(-) ... f(-) ...

fun F(f,g)(-) = ...f(-)...g(-)...
fun G(f,g)(-) = ...g(-)...f(-)...
fun G′(f,g)(-) = ...g(-)...
```

Old family

```
f = F(f,g)
g = G(f,g)
```

New family

```
f′ = F(f′,g′)
g′ = G′(f′,g′)
```

Requires recursive types.

Open recursion with function refs

```
fr = ref(dummy);    gr = ref(dummy)
fun f(x) = ... !gr(-) ... !gr(-) ...
fun g(x) = ... !gr(-) ... !fr(-) ...
fun g'(x) = ... !gr(-) ...
```

Old family

```
  fr := f; gr := g;
  ... !fr(-) ...
```

New family

```
  gr := g';
  ... !fr(-) ...
```

General support for adaptation

- abstraction and information hiding support adaptation by reducing dependencies between program components
  - e.g. the implementation of an abstract type can be modified without disturbing clients
- explicit, enforced interfaces support change by allowing the consequences of a change to be easily tracked through a large system

II. Type systems

- What is a type?
  - a set of values

    int = {... -2, -1, 0, 1, 2, ...}

  - a specification of the form or structure of values
  - a device for controlling how we can act on values of the type

Language of types

- Types are expressed by terms in a type abstract syntax:

```
A ::= int | A * B | A -> B | ...

E.g.  int, int * int, int -> int,
      int -> (int -> int)
```

- add constructs to increase power and expressiveness

Types and Terms

- Types are related to an underlying language for expressing and manipulating values

```
e ::= x | fun(x)e | e e'          (implicit)
e ::= x | fun(x:A)e | e e'        (explicit)
```

through typing judgements:

```
e : A
```

"e has type A"

Typing environments

- How do we know the type of a free variable?

  x: ?

- Answer: typing environments

  C ::= ∅ | C; x: A

  (finite mapping from variables to types)

- Contexts are added to typing judgements to deal with free vars

  C ⊢ e : A

  E.g. **x: int; y: bool ⊢ x: int**

Typing Rules

- deduction rules for typing judgements

$$\frac{\texttt{C} \vdash \texttt{e: A -> B} \qquad \texttt{C} \vdash \texttt{e' : A}}{\texttt{C} \vdash \texttt{e e' : B}} \quad \text{(-> Elim)}$$

$$\frac{\texttt{C; x: A} \vdash \texttt{e : B}}{\texttt{C} \vdash \texttt{fun(x:A)e : A -> B}} \quad \text{(-> Intro)}$$

Typing rules: atomic expressions

Integer constants

$$\frac{}{\texttt{C} \vdash n : \texttt{Int}}$$ (n an integer)

Variables

$$\frac{}{\texttt{C; x: A} \vdash \texttt{x: A}}$$ (Var)

Typing derivations

- proof of a valid typing judgement
- laid out as a tree of rule instances

Derivation of : `C |- +(x,3): int`, where

`C = + : int * int -> int; x: int`

$$\dfrac{\dfrac{}{C \vdash +:\ int*int \to int} \qquad \dfrac{\dfrac{}{C \vdash x:int} \quad \dfrac{}{C \vdash 3:int}}{C \vdash (x,3) : int*int}}{C \vdash +(x,3) : int}$$

Well-typing

e is well-typed wrt context **C** if there is a type **A** such that

$$C \vdash e : A$$

is a valid judgement.

1+3 - well typed wrt context **C**

1+true - ill typed (**true: bool**)

there is no type **A** with valid judgement

$$C \vdash 1{+}true : A$$

Type Inference

- Type inference (for a term **`e`**) is the process of discovering a type **`A`** and a derivation of

$$\mathbf{C} \vdash \mathbf{e} : \mathbf{A}$$

Multiple typings

- There may be more than one type **A** such that **C ⊢ e : A**
- E.g. typing (untyped) identity function

$$\frac{\texttt{x: Int} \vdash \texttt{x : Int}}{\emptyset \vdash \texttt{fun(x)x : Int -> Int}}$$

$$\frac{\texttt{x: Bool} \vdash \texttt{x : Bool}}{\emptyset \vdash \texttt{fun(x)x : Bool -> Bool}}$$

$$\frac{\texttt{x: A} \vdash \texttt{x : A}}{\emptyset \vdash \texttt{fun(x)x : A -> A}}$$

References for Type Systems

- Cardelli: "Type Systems" tutorial

    in library, Cardelli's web page

- B. Pierce: "Type Systems"

    forthcoming comprehensive book with software

    toolkit

Typing and program behavior

- Denotational semantics
  - [·] : expressions → values
  - [·] : types → sets of values
- Soundness:

$$\varnothing \vdash \mathtt{e} : \mathtt{A} \;\Rightarrow\; [\mathtt{e}] \in [\mathtt{A}]$$

- Type errors

[**e**] = **wrong** if **e** contains a type error

e.g. [**1+true**] = **wrong**

wrong ∉ [**A**] for any type **A**

Typing and program behavior

- Operational semantics

  $e \rightarrow e'$ **single step reduction**

  $e \rightarrow^* e'$ **multiple step reduction**

- Subject reduction

  $e \rightarrow^* e'$ and $\emptyset \vdash e : A \Rightarrow \emptyset \vdash e' : A$

- Corollary: well-typed expressions don't get stuck.

  `1 + true` **is a stuck expression**

Some basic type constructs

- products: **A * B**
- records: $\{m_1 : A_1 , \ldots , m_n : A_n\}$
- sums: **A + B**
- functions: **A -> B**
- recursion: μ**t.A**
- refs: **Ref A**
- subtypes: **A <: B**

Products

$A * B = \{(a,b) \mid a \in A,\ b \in B\}$

$$\frac{C \mid\!- \ e_1 : A \qquad C \mid\!- \ e_2 : B}{C \mid\!- \ (e_1,\ e_2) : A * B}$$ (* Intro)

$$\frac{C \mid\!- \ e : A * B}{C \mid\!- \ \mathtt{fst}\ e : A}$$ (* Elim left)

$$\frac{C \mid\!- \ e : A * B}{C \mid\!- \ \mathtt{snd}\ e : B}$$ (* Elim right)

Products

```
p = (1,true) : Int * Bool

fst p : Int ==> 1

snd p : Bool ==> true
```

$$\frac{\varnothing \vdash \texttt{1 : Int} \qquad \varnothing \vdash \texttt{true : Bool}}{\varnothing \vdash \texttt{(1, true): Int * Bool}}$$

Products

Projections as primitive operations

for each pair of types **A and B**:

```
fst(A,B) : A * B -> A

snd(A,B) : A * B -> B
```

pairing as a primitive?

```
(.,.)(A,B) : A -> B -> A * B   (?)
```

Records

record = labeled product

= finite map from labels to values

```
r = {age = 13, name = "Bob"}
```

record type = finite map from labels to types

```
r : {age : Int, name : String}
```

field = labeled component of a record

```
r.age : Int ==> 13
```

(field selection)

Typing records

Obvious generalization of product rules:

$$\frac{C \vdash e_i : A_i \quad (i=1,\ldots,n)}{C \vdash \{m_1=e_1,\ldots,m_n=e_n\} : \{m_1:A_1,\ldots,m_n:A_n\}}$$

(Record Intro)

$$\frac{C \vdash e : \{m_1:A_1,\ldots,m_n:A_n\}}{C \vdash e.m_i : A_i}$$

(Record Elim)

Sums

A + B - tagged union of **A** and **B**

`inl: A -> A + B`   `inr: B -> A + B`

`outl: A + B -> A`   `outr: A + B -> B`

`isl: A + B -> Bool`   `isr: A + B -> Bool`

Actually, should be: $\mathbf{inl}_{(\mathbf{A},\mathbf{B})}$, etc.

Sums

```
i = inl 3 : Int * Bool    (inl(Int,Bool))

b = inr true : Int * Bool

isl b : Bool ==> false
outl a : Int ==> 1
```

Sums

Could replace isl, isr, outl, outr with case

```
case(A,B,C) : (A+B) * (A->C) * (B->C) -> C

case(A,B,C)(x,f,g) =
  if isl x then f(outl x)
  else g(outr x)

case (a,
      fun(n)(n+1),
      fun(b)(if b then 3 else 4)) ==> 2
```

Recursive types

μ`t.A`   `[or Rec(t)A, Fix(Fun(t)A)]`

Equirecursive types:

$$\mu t.A = [\mu t.A/t]A$$

$$\frac{C \vdash e : \mu t.A}{C \vdash e : [\mu t.A/t]A} \qquad \frac{C \vdash e : [\mu t.A/t]A}{C \vdash e : \mu t.A}$$

Isorecursive types

μt.A ↔ [μt.A/t]A

$$\frac{\texttt{C |- e: }\mu\texttt{t.A}}{\texttt{C |- unfold e: [}\mu\texttt{t.A/t]A}}$$

(Rec Elim)

$$\frac{\texttt{C |- e: [}\mu\texttt{t.A/t]A}}{\texttt{C |- fold e: }\mu\texttt{t.A}}$$

(Rec Intro)

Recursive type example: integer lists

```
List = μt.(Unit + Int * t)

nil = fold(inl())
    : List

cons = fun(i:int,x:list)fold(inr(i,x))
     : Int * List -> List

hd = fun(x:list)
        if isr(unfold x)
          then fst(outr(unfold x))
          else error_List
   : List -> Int
```

Ref

`Ref A` -- mutable cells containing **`A`** values

$\mathtt{ref}_{\mathtt{A}}$ `: A -> Ref A`

$!_{\mathtt{A}}$ `: Ref A -> A    (deref)`

$:=_{\mathtt{A}}$ `: Ref A * A -> Unit`

Example of a storage type; also arrays and mutable records

Ref

```
r = ref 3 : Ref Int

!r : Int ==> 3

r := 4 : Unit ==> ()    ( (): Unit )

!r : Int ==> 4
```

The subtype relation

- **A <: B**
  - **A** is a subset of **B**: [**A**] $\subseteq$ [**B**], or
  - There is a canonical injection of [**A**] into [**B**] (coercion semantics)

- **<: is reflexive, transitive and antisymmetric***

Subsumption

the typing rule reflecting the interpretation of subtypes as subsets:

$$\frac{\texttt{C |- e: A} \qquad \texttt{C |- A <: B}}{\texttt{C |- e: B}} \quad \text{(Subsumption)}$$

How is **C** relevant to **A <: B**?

Propagation of subtyping

- Products (monotonic)

$$\frac{A_1 <: A_2 \qquad B_1 <: B_2}{A_1 * B_1 <: A_2 * B_2}$$

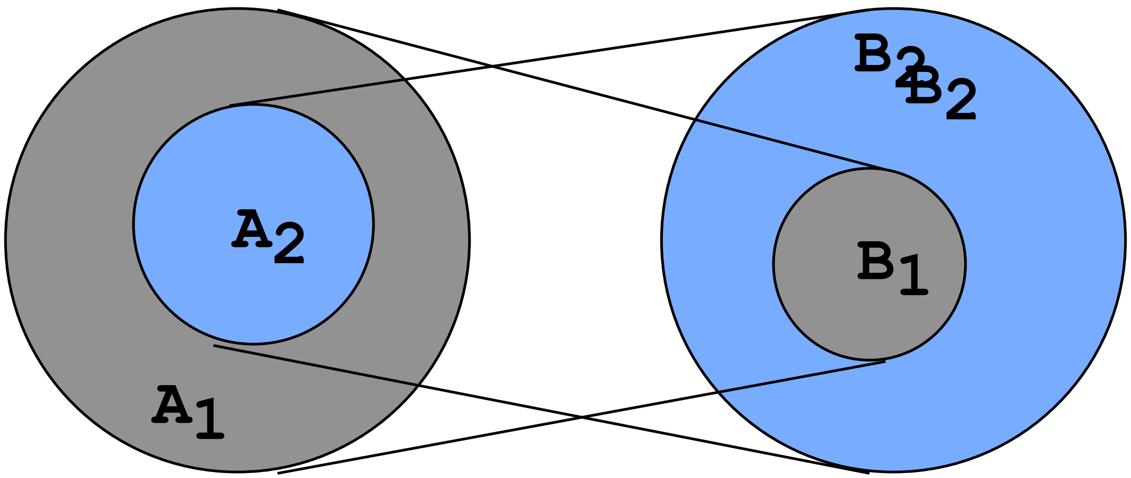
- Sums (monotonic)

$$\frac{A_1 <: A_2 \qquad B_1 <: B_2}{A_1 + B_1 <: A_2 + B_2}$$

Propogation of subtyping

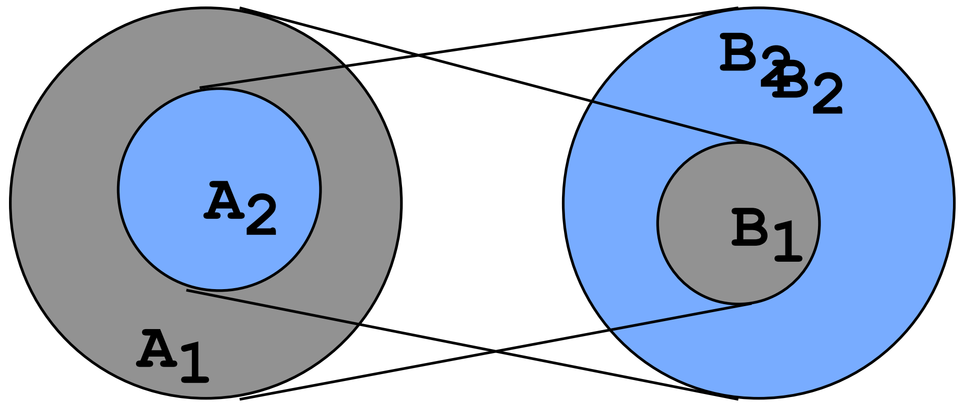
- function types

When is A_1 `->` B_1 `<:` A_2 `->` B_2 ?



Function subtyping

$f \in A \to B$ if
$x \in A \Rightarrow f\ x \in B$



$A_2 <: A_1$ $B_1 <: B_2$

Assume $A_2 <: A_1$ and $B_1 <: B_2$.

Assume $f \in A_1 \to B_1$ and $x \in A_2$.

Then $x \in A_1 \Rightarrow f\ x \in B_1 \Rightarrow f\ x \in B_2$.

Hence $f \in A_2 \to B_2$.

Subtyping function types

function types

- contravariant (antimonotonic) in the domain
- covariant (monotonic) in the range.

$$\frac{A_2 <: A_1 \qquad B_1 <: B_2}{A_1 \to B_1 <: A_2 \to B_2}$$

Subtyping records

Depth subtyping (like products)

$$\frac{\mathtt{A_i} \ \mathtt{<:} \ \mathtt{B_i} \quad (\mathtt{i=1,...,n})}{\{\mathtt{m_1:A_1,...,m_n:A_n}\} \ \mathtt{<:} \ \{\mathtt{m_1:B_1,...,m_n:B_n}\}}$$

$$\frac{\mathtt{Nat} \ \mathtt{<:} \ \mathtt{Int}}{\{\mathtt{age:\ Nat,\ b:\ Bool}\} \ \mathtt{<:} \ \{\mathtt{age:\ Int,\ b:\ Bool}\}}$$

Subtyping records 2

Width subtyping -- adding fields makes a subtype.

$\{m_1:A_1,...,m_{n+k}:A_{n+k}\}$ <: $\{m_1:A_1,...,m_n:A_n\}$

```
{age: Int, name: String, Id: Int} <:
{age: Int, name: String}
```

Intuition: If all you can do to a record is select fields, extra fields don't get in the way.

Subtyping Records

Depth and width subtyping can be combinded.

```
{age: Nat, name: String, Id: Int} <:
{age: Int, name: String}
```

Subtyping Refs

Ref types are invariant!

Thm: `Ref A <: Ref B` $\Rightarrow$ `A = B`

```
[B<:A] let r : Ref A and b : B
       r : Ref B  (assumption)
       r := b
       b = !r : A

[A<:B] let r : Ref A and a : A
       r : Ref B  (assumption)
       r := a
       a = !r : B
```

Subtyping and recursion 1

- The “Amber” rule:

$$\frac{\texttt{C, s <: t |- A <: B}}{\texttt{C |- }\mu\texttt{s.A <: }\mu\texttt{t.B}}$$

Subtyping and recursion 2

Example:

```
Nat <: Int |-
   μt.{a: Nat, b: Unit -> t, c: Bool} <:
   μt.{a: Int, b: Unit -> t}


1. s <: t ⇒
2. Unit -> s <: Unit -> t ⇒
3. {a: Nat, b: Unit -> s, c: Bool} <:
   {a: Int, b: Unit -> t}
```

Subtyping and recursion 3

Exercise:

```
NatList = μt.(Unit + Nat * t)

IntList = μt.(Unit + Int * t)
```

1. Show that `NatList <: IntList`, assuming `Nat <: Int`.

```
List = Fun(s)μt.(Unit + s * t)
```

2. Show that `List` is monotonic:

```
List s <: List t if s <: t.
```

Subtyping and recursion 4

fold and unfold rules (equirecursion)

μ**s.F(s) = F(**μ**s.F(s))** implies

μ**s.F(s) <: F(**μ**s.F(s))** (fold)

F(μ**s.F(s)) <:** μ**s.F(s)** (unfold)

Parametric Polymorphism

Parameterize expressions over types

- Add type variables and quantified types to the type language

  `A ::= ... | t | All(t)A` **(Cardelli style)**

  `A ::= ... | t | ∀t.A` **(Classical)**

- Add lambda-abstraction over types and type application to the expression language

  `e ::= ... | Fun(t)e | e[t]`

  `e ::= ... | Λt.e | e[t]`

Parametric Polymorphism: Examples

Examples

Polymorphic identity function

```
Id = Fun(t)fun(x:t)x : All(t)t->t
```

$\texttt{Id = } \Lambda\texttt{t.}\lambda\texttt{x:t.x : }\forall\texttt{t.t->t}$

```
Id[Int] : Int -> Int; Id[Int]3 : Int
```

Self-application combinator (1st class polymorphism)

```
fun(x:All(t)t->t)(x[All(t)t->t])x
  : (All(t)t->t) -> (All(t)t->t)
```

Parametric Polymorphism: Rules

Typing rules

$$\frac{\texttt{C, t : Type} \vdash \texttt{e : A}}{\texttt{C} \vdash \texttt{Fun(t)e : All(t)A}} \quad \text{(All Intro)}$$

$$\frac{\texttt{C} \vdash \texttt{e : All(t)A}}{\texttt{C} \vdash \texttt{e[B] : [B/t]A}} \quad \text{(All Elim)}$$

Parametric Polymorphism: Semantics

- The meaning of a polymorphic type can be:
  - the intersection of all its instances, or
  - a parametric family of types
- **parametricity**: a polymorphic function **`Fun(t)e`** works the same regardless of how **`t`** is instantiated (i.e. computations don't depend on the identity of the type **`t`**).
  - Cor: **`fun(x)x`** is the unique (computable) member of the type **`(All t)(t -> t)`**.

Polymorphic primitive operations

Primitive operations associated with type constructions can be polymorphic

```
fst: All(s)All(t) s * t -> s

inl: All(s)All(t) s -> s + t

if_then_else: All(s) Bool * s * s -> s

ref: All(s)(s -> Ref s)

:= : All(s)(Ref s * s -> Unit)
```

but not `fold, unfold, r.m` !

Type functions

Type functions are defined by abstracting over type terms

```
A ::= ... | Fun(t)A | A(B)

Pair = Fun(t)(t*t)

List = Fun(s)μt.(Unit + s * t)

  null = Fun(s)fun(x:List s)
              isl[Unit][s*List(s)]
                (unfold x)
           : All(s)(List s -> Bool)
```

Typing type functions: kinds

Types now need to be well-typed!

Types of type terms are called kinds.

```
K ::= Type | Type -> Type

   Int : Type
   List : Type -> Type
   List(Int) : Type
   * : Type -> Type -> Type
```

In `C |- e: A`, `A` must be of kind `Type`.

Type system problems

- What can go wrong when designing a type system for a language?
  - type system is unsound (i.e. inconsistent with the evaluation semantics)
  - type system is undecidable (i.e. can't find a terminating algorithm for discovering typings)
  - type system is incomplete (can't find typings for "sensible" expressions)
  - typings are not canonical (i.e. terms have multiple typings, none of which is "best")
  - type system is too complex or difficult to use to be practical (e.g. not enough inference)

III. Functional Programming

Essence of Functional Programming

- functions as first-class values
  - higher-order functions: functions that operate on, or create, other functions
  - functions as components of data structures
- value-oriented programming
  - lego vs the abacus
  - compute by building and traversing values
  - values are shareable
- parameterization is a core concept
- algebraic types and pattern matching

Functional programming history

- lambda calculus (Church, 1932)
- simply typed lambda calculus (Church, 1940)
- lambda calculus as prog. lang. (McCarthy(?), 1960, Landin 1965)
- polymorphic types (Girard, Reynolds, early 70s)
- algebraic types (Burstall & Landin, 1969)
- type inference (Hindley, 1969, Milner, mid 70s)
- lazy evaluation (Wadsworth, early 70s)

Varieties of Functional Programming

- typed (ML, Haskell) vs untyped (scheme, Erlang)
- Pure vs Impure
  - impure have state and imperative features
  - pure have no side effects, “referential transparency”
- Strict vs Lazy evaluation
- Hindley-Milner vs System F typing
  - H-M: implicit typing with type inference
  - System F: explicit type abstraction and appl.

A brief introduction to (Core) ML

- strict evaluation
- impure
  - refs, arrays, imperative I/O, exceptions
- Hindley-Milner type inference
- algebraic types with pattern matching
  - sums and recursive types via datatypes

ML 2

expressions

```
1: int                        1.0: real
true: bool                    “Bob”: string

(1,2): int * bool
{a = true, b = “x”}: {a: int, b: string}

if x<3 then 0 else x+1 : int

(fn x => x+1) : int -> int
square 3 : int

(print “hello”; 4): int
```

ML 3

- declarations

```
val x = 3     (* x: int *)

val inc = (fn x => x+1) (* inc: int->int *)
fun inc x = x+1

type point = {x: int, y: int}  (* type *)
type 'a pair = 'a * 'a  (* type function *)

let val p : int pair = (2,3)  (* a block *)
    fun inc x = x+1
 in inc(#1 p)
end : int
```

Algebraic datatypes

```
datatype tree = Node of int * tree * tree
              | Leaf of int

val t = Node(3,Leaf 2,Leaf 1)
val t : tree

fun depth(Leaf n) = 1
  | depth(Node(n,left,right)) =
     1+max(depth(left),depth(right))
val depth : tree -> int

depth t : int  ==> 2
```

Algebraic datatypes: list

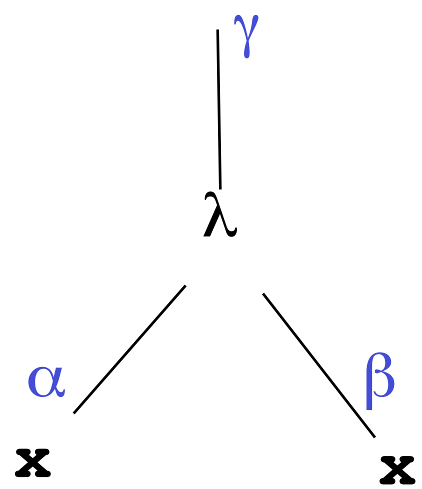
```
datatype 'a list = nil
                 | :: of 'a * 'a list

val a = 2::3::nil  (* or [2,3] *)
val a : int list

fun length nil = 0
  | length (x::xs) = 1 + length xs
val length : 'a list -> int
```

Type inference in ML (Hindley-Milner)

```
val id = (fn x => x)
```



$$\gamma = \alpha \rightarrow \beta$$

$$\alpha = \beta$$

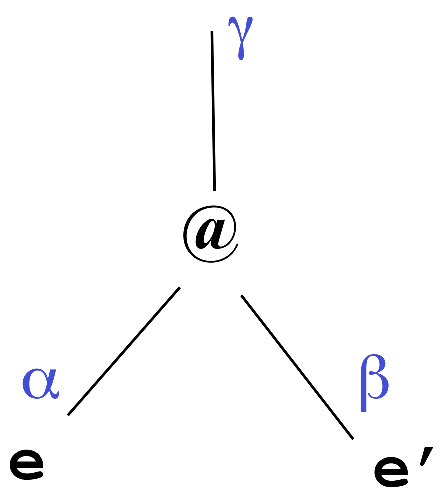
$$\gamma = \alpha \rightarrow \alpha$$

`(fn x => x)` : $\alpha \rightarrow \alpha$, for any α

`id : 'a -> 'a     (id : ∀t.t->t)`

Type inference

Application expressions



$$\alpha = \beta \rightarrow \gamma$$

The let rule

- Polymorphism is introduced by **let** declarations

$$\frac{\mathtt{C} \vdash \mathtt{e} : \mathtt{A} \qquad \mathtt{C},\ \mathtt{x}: \forall \mathtt{X}.\mathtt{A} \vdash \mathtt{e'} : \mathtt{B}}{\mathtt{C} \vdash \mathtt{let\ val\ x = e\ in\ e'\ end} : \mathtt{B}}$$

where **X** is the set of type variables free in **A** but not in **C**: `X = FV(A) - FV(C)`.

Constraint: all polymorphic types are prenex. (so cannot type `(fn x => x x)`)

Polymorphic instantiation

- polymorphic types are implicitly instantiated

$$\frac{C \vdash x : \forall t.A}{C \vdash x : [B/t]A}$$

choose B cannonically by solving equational constraints during type inference

Principal typing

Given a term e and context **C**, if **e** has a typing wrt **C**, then there is a most general typing

C |- e : A

such that any typing of **e** in **C** is an instance:

C |- e : A′ ⇒

A′ is a substitution instance of **A**.

Type inference algorithm

Find the most general typing by:

1. solving for the most general unifier of the equational constraints from typing diagrams.
2. quantifying (∀) any remaining type variables.

Primary sources of polymorphism

- generic functions

```
fn x => x : 'a -> 'a

fn (x,y) => x : 'a * 'b -> 'a
```

- parametric datatype constructors

```
nil : 'a list
::  : 'a * 'a list -> 'a list
```

Polymorphism and refs

```
let val r = ref (fn x => x)
 in r := not;
    !r(3)
end
```

```
r : ∀t.((t->t) ref)
r : (bool->bool) ref
r : (int->int) ref
```

Oops! The type system is unsound!

The value restriction

We fix the type system by restricting the let rule. In

```
let val x = e in ... end
```

we only generalize the type of **x** to a polymorphic type if **e** is a value expression.

A value expression is a constant, variable, or function expression*. Value expressions represent final values, not computations.

System F (F^{ω})

- polymorphism with explicit type abstraction and type application
- polymorphism with variables of higher kinds (e.g. abstraction over type operators)
- nested, nonprenex polymorphic types
- no value restriction needed for refs

Map in ML and System F

```
datatype 'a list = nil | :: of 'a * 'a list

fun map f nil = nil
  | map f (x::xs) = f x :: map f xs
```

val map : ('a -> 'b) -> 'a list -> 'b list

```
List = Fun(s)μt.(Unit + s * t) (+null, hd,...)

map = Fun(s)Fun(t)fun(f:s->t)fun(xs:List s)
       if null[s]xs then nil[t] else
         cons[t](f(hd[s]xs))
          (map[s][t]f(tl[s]xs))
    : All(s)All(t)(s->t) -> List s -> List t
```

Polymorphism and adaptation

- Polymorphic types defined an infinite family of instances, and (with H-M) the appropriate instance for any context is chosen automatically.
- The code for a polymorphic function is shared by all instances (parametricity).

Polymorphism and refs in System F

(∀t.(t->t)) ref

```
let r = ref[All(t)(t->t)](Fun(t)fun(x:t)x)
 in :=[Bool->Bool](r, not);
    (![All(t)(t->t)]r)[Int](3)
end
```

type error!

Polymorphism and refs in System F

`All(t)((t->t)ref)`

```
let r = Fun(t)ref[t->t](fun(x:t)x)
 in r[Bool] := not;
    (![Int->Int](r[Int])(3)
end
```

`r[Bool]: Ref(Bool->Bool)` **and**

`r[Int]: Ref(Int->Int)`

are two different ref values!

IV. Modules

Generalizing polymorphism

Example: polymorphic sort

```
sort = Fun(t)fun(less: t*t->bool). ...
```

In this case, we need to parameterize over a type and an associated comparison function in a coordinated manner.

Generalizing polymorphism

Can we package the type and associated operation together and pass them as one parameter?

```
sort = Fun(<t,less: t*t->bool>). ...
```

What are these hybrid parameter packages?

Modules!

Basic Modules in ML

- A module is a package of (related) declarations (called a structure).

```
struct
  decl_1
  ...
  decl_n
end
```

The component declarations can define types, values (e.g. data, functions, exceptions) and nested modules.

Structure declarations

Structures can be named in structure declarations.

```
structure A = struct ... end
```

The components of the named structure can be accessed using the dot notation:

`A.t` -- a type
`A.f` -- a function
`A.B` -- a substructure (nested module)
`A.B.x` -- a value in **`A.B`**

Example: A stack structure

```
structure Stack =
struct
  type 'a stack = 'a list
  exception Empty
  val empty : 'a stack = nil
  fun push(x,s) = x::s
  fun pop nil = raise Empty
    | pop(_::xs) = xs
  fun top nil = raise Empty
    | top(x::_) = x
end
```

Example: using the Stack structure

```
val s0 : int Stack.stack = Stack.empty

val s1 = Stack.push(1,Stack.push(2,s0))

val s2 = Stack.pop s1

val x = Stack.top s2 handle Empty => 0
```

Signatures (interfaces)

- A **signature** is a type for a structure.
  - It defines the exported interface.
  - Each exported component has a type specification.
  - A signature is implemented by any structure that matches it.

The STACK signature

```
signature STACK =
sig
  type 'a stack
  exception Empty
  val empty : 'a stack
  val push : 'a * 'a stack -> 'a stack
  val pop : 'a stack -> 'a stack
  val top : 'a stack -> 'a
end
```

Signature constraints

We can (and usually do) specify a signature when declaring a structure.

```
structure Stack : STACK =
  struct ... end
```

Then we say that Stack is an instance of signature STACK.

Signature/Structure Independence

- Signatures and structures can be defined independently and later matched.
- Any signature can be matched by multiple structures (i.e. have multiple implementations)
- Any structure can match multiple signatures (i.e. can implement different interfaces, or be viewed through different interfaces)

Signature matching: thining

- Components of a structure can be hidden.
- Component types can be specialized.
- Signature matching is coercive.

```
structure A : sig type t
                  val f : int -> t
              end =
struct
  type t = int
  val a = 3       (* not exported by A *)
  fun f x = a     (* f: 'a -> t *)
end
```

Signatures and record types

There is an analogy between signature matching and width subtyping for records.

Say **`SIG1 <: SIG2`** if for any structure **`S`**, **`S:SIG1`** implies **`S:SIG2`** (i.e. any structure that matches **`SIG1`** will also match **`SIG2)`**.

Width subtyping of signatures

As for records, we can ignore extraneous elements.

```
sig                          sig
  type t                       type t
  type s             <:        val a: t
  val a: t                   end
  val b: s*t
end
```

Depth subtyping of signatures

There is a simple analog of depth subtyping.

```
sig
  val f: 'a -> int
end

     <:

sig
  val f: int -> int
end
```

because `('a -> int) <: (int -> int)`

Transparent signature matching

exported types are not abstract (by default)

```
structure Stack : STACK =
struct
  type 'a stack = 'a list
  ...
end

(* Stack.stack == list *)

null(Stack.empty) : bool ==> true
```

Opaque signature matching

Can specify opaque matching, making exported types opaque.

```
structure Stack :> STACK =
struct
  type 'a stack = 'a list
  ...
end

(* Stack.stack =/= list *)

null(Stack.empty) (* type error *)
```

Sorting again

```
signature ORD = sig
  type elem
  val less : elem * elem -> bool
end

structure IntOrd : ORD = struct
  type elem = int
  val less = (< : int * int -> bool)
end
```

Note that `IntOrd :> ORD` would not be useful!

SORT signature

```
signature SORT =
sig
  structure Ord : ORD
  val sort : Ord.elem list -> Ord.elem list
end
```

A Sort structure

```
structure InsertSort : SORT =
struct
  structure Ord : ORD = IntOrd
  fun insert(x,nil) = x::nil
    | insert(x,y::ys) =
       if Ord.less(x,y) then x::y::ys
       else y::insert(x,ys)
  fun sort nil = nil
    | sort (x::xs) = insert(x,sort xs)
end
```

Fine, but it only sorts int lists.

Import by mention

- A structure definition imports other structures by mentioning them (i.e. having their names appear free in the body).

- E.g. **`IntOrd`** is imported by **`InsertSort`**.

A Generic Sort

```
functor InsertSortF(X: ORD) : SORT =
struct
  structure Ord : ORD = X
  fun insert(x,nil) = x::nil
    | insert(x,y::ys) =
       if Ord.less(x,y) then x::y::ys
       else y::insert(x,ys)
  fun sort nil = nil
    | sort (x::xs) = insert(x,sort xs)
end
```

Replace `IntOrd` with a parameter structure variable `X`.

Instantiation of generic sort

A **functor** is a structure parameterized over a structure name. It can be instantiated by applying the functor to a structure.

```
structure IntInsertSort : SORT =
  InsertSortF(IntOrd)

structure StringInsertSort : SORT =
  InsertSortF(StringOrd)

IntInsertSort.sort [4,7,1,3];
StringInsertSort.sort [“bob”, “alice”];
```

Multiple implementations of sort

Other sorting methods can be provided by other functors that can be applied to IntOrd, StringOrd.

```
structure IntBubbleSort : SORT =
  BubbleSortF(IntOrd)

structure StringBubbleSort : SORT =
  BubbleSortF(StringOrd)

IntBubbleSort.sort [4,7,1,3];
StringBubbleSort.sort [“bob”, “alice”];
```

Higher-order Functors

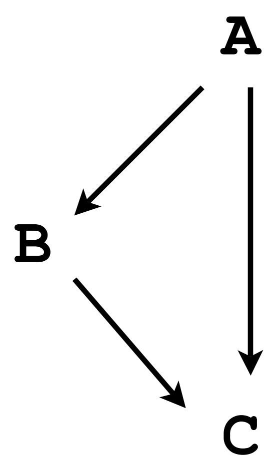
One can imagine abstracting a larger module with respect to the generic sort functor (**InsertSortF**, **BubbleSortF**, etc.).

```
funsig GENSORT(X: ORD) = SORT where Ord = X

functor Prog(SortF: GENSORT) = struct
  structure IntSort = SortF(IntOrd)
  structure StringSort = SortF(StringOrd)
  ...
end
```

Fully Functorized Style

Suppose we have the following dependency graph




```
structure A =
  struct ... end

structure B =
  struct ... A ... end

structure C =
  struct ... A ... B ...
  end
```

Fully Functorized Style

Build the graph through functors, importing via parameters.

```
functor MkA() = struct ... end
functor MkB(A:SIGA) = struct ... A ... end
functor MkC(A:SIGA,B:SIGB) =
  struct ... A ... B ...end

structure A = MkA()
structure B = MkB(B)
structure C = MkC(A,B)
(* or *)
structure C = MkC(A,MkB(A))
```

Translation of functors in F^{ω}

```
functor F(X:sig type t val f: t*t->t end)
  : sig type u val g: X.t*u->u end =
struct
  type u = t list
  fun g(x:t,y:u) =
      map (fn z => f(z,x)) y
end
```

Translation of functors in F^{ω}

F is translated into a pair <F_t, F_v> consisting of a type function and a value function.

```
Ft : Type -> Type = Fun(t:Type)(List t)

Fv : All(t:Type){f:t*t->t} ->
                    {g:t*Ft(t)->Ft(t)}
   = Fun(t:Type)fun({f=f}:{f:t*t->t})
       {g=(fun(x:t,y:Ft(t))
            (map[t][Ft(t)](fun(z:t)f(z,x)) y}
```

Things left out

- transparent and opaque signature matching
- sharing constraints and definitional specs
- higher-order functors

Exercise:

- Construct a similar example of a higher-order functor (e.g. a functor that would take **`F`** as a parameter) and *give* its F^{ω} translation.
- Consider the problem of writing a functor signature that faithfully captures the type function part of your higher-order functor.

V. Object-Oriented Programming

Essence of Object-Oriented Programming

- encapsulation of state
- "dynamic dispatch"
  - Type does not statically determine code run by a method invocation
  - Standard situation in a higher-order language with interface types
- subtyping (or some approximation)
- implementation sharing and modification
  - class-based: inheritance
  - object-based: cloning with extension and update

Degrees of OO purity

- OO as additional features for managing state and allocation on top of a conventional procedural language.
  - E.g.: Simula, C++, Object Pascal, Modula 3, ...
- OO (typically class-based) as the principle means of structuring programs
- pure or pervasive OO
  - everything is an object
  - classes as only program structuring method
  - E.g.: Smalltalk, Java (sort of)

Objects

What is an object?

- state (in the form of instance variables)
- methods (functions acting on the state)

How does this differ from records of function closures?

Object creation

- class-based languages
  - **classes**
- object based
  - **direct creation**
  - **cloning from a prototype object**
    - with extension, method update
    - implemented via embedding or delegation

The magic of objects

capacity for incremental change of functionality

- in the objects themselves (method update)
- as part of object cloning
- in the object generating mechanism (subclassing)

Classes

What is a class?

a template used to generate objects
generated objects are "instances" of the class

- specifies data representation (instance variables)
- specifies implementation of methods
- specifies how objects are initialized
- specifies inheritance from superclass(es)

Deconstructing objects and classes

Goal:

To explain away the magic of OO.

Approach:

Try to code objects and classes in a richly typed functional programming language:

F^{ω} + **<:** + **Rec** + **Ref** + **with** + ...

Existential Types

```
A ::= ... | Some(t)A

e ::= ... | pack t = A with e
          | open e as (t,x) in e′
```

```
           C; t=A |- e : B
--------------------------------------
 C |- pack t = A with e : Some(t)B
```

```
C |- e : Some(t)A    C; t:Type; x:A |- e′: B
---------------------------------------------
       C |- open e as (t,x) in e′ : B
```

Type models for OO languages

Recursive record (Cardelli, Cook, Mitchell)

```
Rec(t)I(t)
```

Existential type (Pierce, Turner)

```
Some(t)I(t)
```

Recursive-Existential types (Bruce)

```
Rec(t)Some(u)(u * (u -> I(t)))
```

Recursive-Bounded-Existential (Abadi, Cardelli, Vis.)

```
Rec(t)Some(u<:t)(u * (u -> I(t)))
```

1. object as record of functions

Cell object is modeled as a record of variables and functions:

```
{x = ref 0,
 get = fun()!x,
 set = fun(y:Int)(x:=y)}
: {x: Ref Int, get: Unit->Int,
    set: Int->Unit}
```

Bogus: `x` is free in bodies of `get`, `set`.

Cell: private variable x

We could make the variable **x** private:

```
let x = ref 0 in
 {get = fun()!x,
  set = fun(y:Int)(x:=y)}
: {get: Unit->Int, set: Int->Unit}
```

But this would make it impossible to extend the record with new functions that have access to **x**.

2. Adding self-reference

give a name for the object to refer to itself (making it a recursive record)

```
self = {x = ref 0,
        get = fun()!self.x,
        set = fun(y:Int)(self.x:=y)}

self = Fix(fun(self)
           {x = ref 0,
            get = fun()!self.x,
            set = fun(y:Int)(self.x:=y)})
```

3. Naming the generator function

Give a name to the fixpoint generating function (essentially a **class**!)

```
CI = {x: Ref Int, get: Unit->Int,
      set: Int->Unit}

Cell = fun(self: CI)
         {x = ref 0,
          get = fun()!self.x,
          set = fun(y:Int)(self.x:=y)}
     : CI -> CI

c = new Cell = Fix(Cell) = μself.Cell(self)
```

4. Variable initialization

add initialization of the **x** variable

```
Cell = fun(n: Int)fun(self: CI)
         {x = ref n,
          get = fun()!self.x,
          set = fun(y:Int)(self.x:=y)}
     : Int -> CI -> CI

c = new (Cell 3)
```

Easy so far -- using value recursion to directly build the object.

5. Inheritance: adding a method

Inheritance: easy case of adding a method

```
CI2 = CI + {inc: Unit -> Unit}

Cell2 : Int -> CI2 -> CI2 =
  fun(n: Int)fun(self: CI2)
    Cell(n)(self) with
     {inc = fun()(self.x := !self.x+1)}
```

Note: `Cell(n)self` **is well-typed because** `CI2 <: CI.`

6. Record concatenation

We are using record concatenation to add the inc method.

```
{a: Int, b: Bool} + {b: Real, c: String}
  = {a: Int, b: Real, c: String}

{a=3, b=true} with {b=1.7, c="cat"}
  = {a=3, b=1.7, c="cat"}
```

Usual rule for map concatenation.

Record concatenation problem

Oops!!

e_1: {a: Int}

e_2: {a: Bool, b: Int}

---

e_1 with e_2: {a: Bool, b: Int}

but {a: Bool, b: Int} <: {b: Int}

implies e_2: {b: Int}, **so we also have**

e_1: {a: Int}

e_2: {b: Int}

---

e_1 with e_2: {a: Int, b: Int}

Record concatenation and subtyping

This example shows that with record concatenation and record subtyping, we can derive two incompatible typings of $\mathbf{e_1}$ **with** $\mathbf{e_2}$.

The reason? **with** is an operation on records where extra fields make a difference.

Exercise

Suggest a fix that will allow record concatenation and width record subtypine to coexist.

8. Recursive interface type

We need to add recursive interface types.

```
Cell = fun(n: Int)fun(self: CI)
          {x = ref 0,
           get = fun()!self.x,
           set = fun(y:Int)(self.x:=y),
           me = fun()self}
      : Int -> CI -> CI
```

where

```
CI = {x: Ref Int, get: Unit->Int,
      set: Int -> Unit,
      me: Unit -> CI}
```

Recursive Interface Type

```
CI = μt.{x: Ref Int, get: Unit -> Int,
         set: Int -> Unit,
         me: Unit -> t}
   = Fix(CIF)
```

where

```
CIF = Fun(t){x: Ref Int,
             get: Unit -> Int,
             set: Int -> Unit,
             me: Unit -> t}
```

9. Inheritance with recursive interface

Add inc method to cell with me method.

```
Cell2 : Int -> CI2 -> CI2 =
  fun(n: Int)fun(self: CI2)
    Cell(n)(self) with
     {inc = fun()(self.x := !
self.x+1)}
```

Extending recursive interface

There are two ways to extend the recursive interface type CI: outside the recursion, or inside.

```
CI2 = CI + {inc: Unit -> Unit}
    = {x: Ref Int, get: -, set: -,
       me: Unit -> CI, inc: Unit -> Unit}

CI2'= μt.(CIF(t) + {inc: Unit -> Unit})
    = {x: Ref Int, get: -, set: -,
       me: Unit -> CI2', inc: Unit -> Unit}
```

Extending recursive interface

`CI2 <: CI`

because of record (width) subtyping

`CI2' <: CI`

because of the recursion rule (**`CI2'`** appears in a positive, covariant position)

10. Method not specialized

Oops! In `CI2`, the type of the me method wasn't specialized.

```
Cell(n) : CI -> CI

=> Cell(n)self : CI (self: CI2 <: CI)

=> body of Cell2 : CI + {inc: Unit -> Unit}

=> Cell2: CI2 -> CI2

=> me method: Unit -> CI  (not CI2)
```

Class Functions, Class Interface

```
Class A :

  A = fun(<init>)fun(self: AI)
        <record expression>
    : InitTyA -> AI -> AI

  AIF = Fun(t)<record type>

  AI = Fix(AIF)

  a = Fix(A x0)
```

Inheritance by deriving class function

```
Class B inherits from A :

  B = fun(<init>)fun(self: BI)
        A(<init'>(self)
         with <record expression>
    : InitTyB -> BI -> BI

  BIF = Fun(t)(AIF(t) + <record type>)

  BI = Fix(BIF)

  b = Fix(B x1)
```

11. Method specialization

To get

```
Cell2: CI2' -> CI2'
```

with the type of `me` specialized to return' `CI2'`, we need to generalize the type of `Cell` (so it doesn't return `CI`).

We make `Cell` polymorphic, but with a new twist: bounded polymorphism.

Method specialization

```
Cell = Fun(t<:CI)fun(n:Int)fun(self:t)
          {x: ref n, get = ...,
           me = fun()self}

Cell: All(t<:CI)Int -> t -> CIF(t)
```

We need `t<:CI` so that `self.x` type checks.

```
c : CI = new (Cell[CI] 3)

   Cell[CI] : Int -> CI -> CIF(CI)
            = Int -> CI -> CI
```

Bounded quantification

`Fun(t<:A)` and **`All(t<:A)`**

are bounded type abstraction and quantification where the type variable t is constrained to range over subtypes of **`A`**.

This guarantees that t has certain properties, e.g. is a record type with certain fields (assuming that a subtype of a record is a record ...).

Method specialization

```
Cell2 = Fun(t<:CI2')fun(n:Int)fun(self:t)
         Cell[t](n)(self) with
         {inc = fun()(self.x := !self.x+1)}

Cell2: All(t<:CI2')Int -> t -> CIF2(t)
```

where

```
CIF2 = Fun(t)(CIF(t)+{inc: Unit -> Unit})
CI2' = Fix(CIF2)
```

Note that `CI2'<:CI`, so `t<:CI2' => t <: CI.`

```
c: CI2' = new (Cell2[CI2'] 3)
```

13. Binary methods

Start over, but with Cell having a binary method `eq`.

```
CIF = Fun(t){x: Ref Int, get: -, set: -,
             eq: t -> Bool}

CI = Fix(CIF)

Cell = fun(n: Int)fun(self: CI)
          {x = ref n, get = -, set = -,
           eq = fun(y:CI)(!self.x = !y.x)}

Cell: Int -> CI -> CI
```

14. Inheritance with Binary Methods

Add a boolean field and override **eq** to check bool field.

```
CI2 = CI + {b: Ref Bool, eq: CI2 -> bool}

Cell2 = fun(n: Int)fun(Self: CI2)
          Cell(n)(self) with
          {b = ref true,
           eq = fun(c:CI2)(!self.x = !c.x
                   and !self.b = !c.b)}
```

Oops! `CI2 </: CI` **because of the contravariant change in** `eq`**'s type. Type error!**

Inheritance with binary methods

How about

`CI2′ = μt.(CIF(t) + {b: Ref Bool})` ?

Again `CI2′ </: CI` because of the contravariant occurrence of `t`. Again `Cell(n)(self)` won't type check.

Bounded quantifier trick won't help here because `CI2′ </: CI`.

F-bounded polymorphism

```
Cell = Fun(t<:CIF(t))fun(n:Int)fun(self:t)
        {x = ref n, get = -, set = -,
         eq = fun(c:t)(!self.x = !c.x)}
Cell: All(t<:CIF(t))(Int -> t -> CIF(t))

CIF2 = Fun(t).(CIF(t)+{b: Ref Bool})

Cell2 = Fun(t<:CIF2(t))fun(n:Int)fun(self:t)
          Cell[t](n)(self) with
          {b = ref true,
           eq = fun(c:t)(!self.x = !c.x
                  and !self.b = !c.b)}
Cell2: All(t<:CIF2(t))Int -> t -> CIF2(t)
```

Inheritance without subtyping

`Cell[t](n)(self)` **type checks because**

```
t <: CIF2(t) = CIF(t) + {b: Ref Bool}
=> t <: CIF(t)
```

```
c2 : CI2 = new (Cell2[CI2] 3)
```

where `CI2 = Rec(CIF2)`

```
CI2 = Fix(CIF2) </: Fix(CIF) = CI
```

so inheritance does not imply subtypes, but

```
CI2 <: CIF(CI2)
```

Object protocols

Constraints like

```
t <: F(t)
```
(1)

are called object protocols. An object type satisfying such a protocol is recursive and provides the interface specified by `F` (wrt to itself).

A `t` satisfying (1) is not necessarily a subtype of `ut.F(t)`, so (1) is a weaker constraint than

```
t <: ut.F(t)
```
(2)

16. Class recursion

Suppose we want a method that returns a new object of the same kind.

```
Cell = Fun(t<:CIF(t))fun(n:Int)fun(self:t)
        {x = -, get = -, set = -,
         double =
          fun()new(Cell[CI](2*!self.x))}

CIF = Fun(t){x: Ref Int,...,double: Unit->t}

CI = Fix(CIF)
```

Class recursion

`Cell` is called recursively, so we create a functional to take the fixpoint of:

```
CELL = Fun(t<:CIF(t))
        fun(myclass: Int -> t ->t)
         fun(n: Int)
          fun(self: t){x = ...,
            double =
             fun()new(myclass(2*!self.x))}

CELL: All(t<:CIF(t))(Int -> t -> t)->
                      (Int -> t -> CIF(t))
Cell = Fix(CELL[CI])
c = new(Cell 3)
```

Inheritance with Class Recursion

Add a new boolean field to Cell.

```
CIF2 = Fun(t)(CIF(t) + {b: Bool})

CELL2 = Fun(t<:CIF2(t))
         fun(myclass: Int*Bool -> t -> t)
          fun(n:Int,b:Bool)
           fun(self:t)
            (CELL[t]
             (fun(x:Int)(myclass(x,self.b)))
             (n)(self))
            + {b = b}
```

Observations

The recursive record coding becomes convoluted.

- Three levels of recursion:
  - **in the construction of objects**
  - **in the construction of object types**
  - **in the construction of classes**
- Bounded polymorphism for covariant method specialization
- F-bounded polymorphism for contravariant method specialization

Observations

- Contravariance causes complications
  - **Inheritance and subtyping are uncoupled: Subclass interface is not a subtype of the superclass interface.**
  - **The F-bounded relation ("matching") replaces subtyping.**

Technical Problems

- Recursive definitions of mixed records of fields and methods
  - more refined models separate fields and methods
- Coexistence of record (width) subtyping and record concatenation

Features Explained

- **objects**
- **object (interface) types**
- **classes**
- **new**
- **inheritance**
- **covariant method specialization**
- **contravariant method specialization (binary methods)**
- **MyType (the type variable in polymorphic class functions)**

Features Not Explained

- super
- private, protected members
- object cloning
- multiple constructors
- class (static) members
- nested (static) classes
- inner classes
- anonymous classes
- multiple inheritance

OO issues

- **implementation inheritance and open recursion**
- **interface types vs implementation (class) types**
- **inheritance vs subtyping**
- **method specialization**
- **contravariance: binary method problem**
- **invariance of mutable variables**
- **access control features and scope dependent types**
- **methodologincal problems of OO**

Method recursion

- methods can call one another, hence are mutually recursive
- recursion of methods is typically indirect, via a **self** variable, to facilitate open recursion

Open Recursion

- Methods are mutually recursive ⇒ incremental change may involve open recursion.
- Some methods change, while others remain the same. But since they are mutually recursive, the behavior of any or all methods may change.
- Extension with new methods does not involve open recursion, because old methods will not call the new ones.

Open Recursion Example

```
O1 = {x = fun()1, y = fun()(2*self.x())}
O2 = {x = fun()1, y = fun()2 }

   O1.x() ==> 1      O1.y() ==> 2
   O2.x() ==> 1      O2.y() ==> 2

O1′ = O1 + {x = fun()2}
O2′ = O2 + {x = fun()2}

   O1′.x() ==> 2     O1′.y() ==> 4
   O2′.x() ==> 2     O2′.y() ==> 2
```

Classes as types

- in most class-based languages, a class is a type, and subclasses are subtypes
- object has a class type if it is an instance of that class or one of its subclasses
- class types determine an interface: a set of publicly accessible attributes and their types
- class types have a family of associated implementations: the class and all its subclasses

Classes as types (cont.)

- An object may have a class type without sharing any of the implementation of that class. Being a member of a class type does not guarantee any consistency of behavior.

- Class types usually support some form of dynamic typecase.
  - **checked casts**
  - **classOf operations to query the class of an object**

Interface types vs Class Types

1. Can't dynamically ask if an object has an interface type

2. Consequently, can't do checked type casts to an interface type

3. Interface types do not determine an implementation

4. In binary methods the two objects may have different implementations.

Classes and abstract types

- OO folklore: class = abstract type
- abstract types determine:
  - **a fixed, hidden data representation**
  - **a fixed interface of operations**
  - **a fixed implementation that guarantees invariants and other desired properties**
- belonging to a class (as type) does not guarantee consistent implementation, because of method override
- only “final” classes are “abstract”

Classes vs modules

- "static" (or "class") variables and methods and (in Java) static embedded classes allow classes to act as a limited form of module
- lack independently defined module interfaces
- the class mechanism is overloaded to perform the role of modules. The function of modules is better performed by simpler, special purpose constructs (see Moby and Loom designs)

Method specialization

- Method specialization with covariant occurrences of the object type is easy
  - C++ allows this, Java requires invariant method types
- Method specialization with contravariant occurrences of the object type (binary methods) is possible,
  - but subtyping is weakened to "matching" an object protocol

Partially abstract object types

Subtype specs like

```
t <: CI
```

or object protocol specs like

```
t <: CIF(t)
```

can be used to partially reveal the interface of objects or limit access to object attributes

VI. Comparing FP and OO

Factoring behavior

- FP
  - fully specify the data (all variants)
  - incrementally add functions over the data
  - adding new variants to data requires rewriting functions
- *OO*
  - incrementally specify the data variants by adding subclasses
  - adding new functions requires adding new methods to all subclasses

FP: modifying behavior

FP relies almost entirely on parameterization

- **have to anticipate what factors will change and abstract over appropriate names**
- **can't change parameterization of existing code without rewriting the code**
- **get maximal flexibility by parameterizing over everything, but this is usually inconvenient**

OO: modifying behavior

inheritance with method override

- **effectively same as modifying the class source**
- **claim that this accommodates “unanticipated” changes**
- **code change requires thorough understanding of old code to determine what behavior changes, what behavior is preserved**

"Binary" functions

- OO
  - **hard to implement symmetric functions (binary methods)**
  - **hard to dispatch over multiple arguments**
- FP
  - **easy to define symmetric functions and dispatch over multiple arguments**

Types in FP

- all types are interface types except abstract types
- abstract types carry a fixed interface implementation

Types in OO

- **interface types**
  - **allow multiple implementations**
    - implementations may match any interface type, or
    - implementations must declare what interface types they match
- **implementation types (class types)**
  - **final classes fix the implementation, can be considered a kind of abstract type**
  - **other classes allow a family of implementations represented by their subclasses (multiple implementations related by subclassing)**

Synthesizing FP and OO

Why?

- clear utility of subtyping (OO)
- clear utility of parametric polymorphism
- state encapsulation ?
- implementation inheritance ?
- popularity of OO

Merging FP and OO

- Pizza (Wadler & Oderski)
  - added encodings of functions, datatypes, and parametric polymorphism to Java

- GJ - Generic Java (Wadler, Oderski, ...)
  - add parametric polymorphism and F-bounded polymorphism to Java

Encoding objects in ML

Tofte and Thorup showed how the existential type model could be coded in Standard ML, using explicit coercions in place of subtyping.

The encoding machinery is heavy, but some simple type system extensions and syntactic sugar could might make this encoding useable.

OCaml (Objective Caml)

- adds objects, object types, classes to Caml dialect of ML
- based on Pierce-Turner existential model
- uses "row" polymorphism in place of record subtyping:

  $\forall\rho$.`{a: int, b: bool,` ρ`} -> int`

- explicit coercion rather than subsumption
- classes not first class, limited interaction with modules

Moby (Fisher and Reppy)

- subtypes, nested parametric polymorphism
- object types and classes, based on Fisher's "protocol" model
- object view and class view
- uses modules for visibility control
- many Java features

$F_{<,rec}$

- provides
  - subtyping
  - parametric polymorphism at all kinds
  - nested polymorphism
  - bounded polymorphism (but not F-bounded)
  - records
  - existential types, bounded existential types
  - can express most OO encodings
- problems
  - cumbersome explicit typing, practical type inference algorithms not proven

Extended ML

Simple extensions of the ML type system can improve ability to emulate OO techniques

- encapsulated existential and universal types
- extensible datatypes (incrementally add data constructors)
- row-polymorphism for record types or limited record subtyping

Final Thoughts

FP vs OO

- FP has simpler, lighter-weight structures
  - **functions vs objects**
  - **records, datatypes vs objects**
  - **polymorphism, modules vs subtyping, inheritance**
- Both OO and FP can encapsulate state
- Subtyping would be a valuable addition to FP, but type inference must be dealt with
- Inheritance is probably not worth adding to FP.

Language Complexity budget

- Adding a full-featured object/class system to a language like ML will roughly double the complexity of the type system.
- The OO features are relevant only to the impure 5-10% of typical ML programs.
- This appears to be a bad bargain.

Language design criticism

Should the study of language designs be like botany or horticulture?

Type Theory

Type theory is an excellent tool for language design and the analysis of language designs.

URLs

Standard ML and Standard ML of New Jersey

`http://www.smlnj.org`