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, Inheritance

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 basis 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)

```plaintext
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.

\[
\text{fun sort } [t](\:< t*t->\text{bool}) (l: t \text{ list}) = \\
(\text{if null } l \ldots x < y \ldots) \\
\text{sort : } \forall t. (t*t->\text{bool}) \rightarrow t \text{ list } \rightarrow t \text{ list}
\]
Interface parameters

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

```plaintext
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

\[ A <: B \quad \text{“every } A \text{ is a } B\text{”} \]

Nat <: Int, Ascii <: Unicode

• Subsumption Rule

\[
\begin{align*}
A & <: B \\
x & : A \\
\hline
x & : B
\end{align*}
\]
Picturing subtyping: $A <: B$

subset

embedding

$A\rightarrow B$

$1 - 1$
Sources of subtyping

Mathematics: \(\text{Nat} <: \text{Int}\)

Machines: \(\text{UnsignedInt32} <: \text{Int32}\) ?

Mathematics: \(\text{Int} <: \text{Real}\)

Machines: \(\text{Int32} <: \text{Float64}\)

(representation shift)
Sources of subtyping: records

point = \{x: \text{int}, y: \text{int}\}

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

colorpoint <: point

\begin{array}{ccc}
\text{point} & \text{colorpoint} & \text{colorpoint'} \\
\begin{array}{c}
\text{x} \\
\text{y}
\end{array} & \begin{array}{c}
\text{x} \\
\text{y} \\
\text{c}
\end{array} & \begin{array}{c}
\text{c} \\
\text{x} \\
\text{y}
\end{array}
\end{array}
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

\[
\text{fun } f(x) = \ldots g(-) \ldots g(-) \ldots \\
\text{fun } g(x) = \ldots g(-) \ldots f(-) \ldots \\
\text{fun } F(f,g)(-) = \ldots f(-) \ldots g(-) \ldots \\
\text{fun } G(f,g)(-) = \ldots g(-) \ldots f(-) \ldots \\
\text{fun } G'(f,g)(-) = \ldots g(-) \ldots \\
\]

Old family                              New family
\[
f = F(f,g)                               f' = F(f',g') \\
g = G(f,g)                               g' = G'(f',g') \\
\]

Requires recursive types.
Open recursion with function refs

\[
\begin{align*}
fr &= \text{ref}(\text{dummy}); & gr &= \text{ref}(\text{dummy}) \\
\text{fun} \ f(x) &= \ldots \ !gr(-) \ldots \ !gr(-) \ldots \\
\text{fun} \ g(x) &= \ldots \ !gr(-) \ldots \ !fr(-) \ldots \\
\text{fun} \ g'(x) &= \ldots \ !gr(-) \ldots \\
\text{Old family} \\
fr &:= f; \ gr := g; \\
\ldots \ !fr(-) \ldots \\
\text{New family} \\
gr &:= g' \\
\ldots \ !fr(-) \ldots
\end{align*}
\]
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
    \[ \text{int} = \{... \text{-2, -1, 0, 1, 2, ...}\} \]
  - 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:

\[ A ::= \text{int} \mid A \times B \mid A \rightarrow B \mid \ldots \]

E.g. \( \text{int}, \text{int} \times \text{int}, \text{int} \rightarrow \text{int}, \text{int} \rightarrow (\text{int} \rightarrow \text{int}) \)

• add constructs to increase power and expressiveness
Types and Terms

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

\[ e ::= x \mid \text{fun}(x)e \mid e\ e' \quad \text{(implicit)} \]
\[ e ::= x \mid \text{fun}(x:A)e \mid e\ e' \quad \text{(explicit)} \]

through typing judgements:

\[ e : A \quad \text{“e has type A”} \]
Typing environments

• How do we know the type of a free variable?
  \[ x : ? \]

• Answer: typing environments
  \[ C ::= \emptyset \mid C; x: A \]
  (finite mapping from variables to types)

• Contexts are added to typing judgements to deal with free vars
  \[ C \vdash e : A \]

E.g. \[ x: \text{int}; y: \text{bool} \vdash x: \text{int} \]
Typing Rules

- deduction rules for typing judgements

\[
\begin{align*}
C \vdash e : A \rightarrow B & \quad C \vdash e' : A \\
\hline
C \vdash e \, e' : B
\end{align*}
\]

\[\text{\textit{\rightarrow Elim}}\]

\[
\begin{align*}
C; x : A \vdash e : B \\
\hline
C \vdash \text{fun}(x:A)e : A \rightarrow B
\end{align*}
\]

\[\text{\textit{\rightarrow Intro}}\]
Typing rules: atomic expressions

Integer constants

\( C \vdash n : \text{Int} \)  
(n an integer)

Variables

\( C; x : A \vdash x : A \)  
(Var)
Typing derivations

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

Derivation of: $C \vdash +(x,3) : \text{int}$, where
$C = + : \text{int} \times \text{int} \rightarrow \text{int}; x: \text{int}$

\[
\begin{align*}
C \vdash x: \text{int} & \quad C \vdash 3: \text{int} \\
\hline
C \vdash +: \text{int} \times \text{int} \rightarrow \text{int} \quad C \vdash (x,3) : \text{int} \times \text{int} \\
\hline
C \vdash +(x,3) : \text{int}
\end{align*}
\]
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

\[ C \vdash e : A \]
Multiple typings

• There may be more than one type $A$ such that $C \vdash e : A$

• E.g. typing (untyped) identity function

\[
\begin{align*}
x : \text{Int} & \vdash x : \text{Int} \\
\emptyset & \vdash \text{fun}(x)x : \text{Int} \rightarrow \text{Int} \\
x : \text{Bool} & \vdash x : \text{Bool} \\
\emptyset & \vdash \text{fun}(x)x : \text{Bool} \rightarrow \text{Bool} \\
x : A & \vdash x : A \\
\emptyset & \vdash \text{fun}(x)x : A \rightarrow A
\end{align*}
\]
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
  - $[\cdot]$: expressions $\rightarrow$ values
  - $[\cdot]$: types $\rightarrow$ sets of values

• Soundness:
  \[
  \emptyset \vdash e: A \Rightarrow [e] \in [A]
  \]

• Type errors
  \[
  [e] = \text{wrong} \text{ if } e \text{ contains a type error}
  \]
  e.g. $[1+\text{true}] = \text{wrong}$

  \[
  \text{wrong} \notin [A] \text{ for any type } A
  \]
Typing and program behavior

• Operational semantics
  \[ e \rightarrow e' \quad \text{single step reduction} \]
  \[ e \rightarrow^* e' \quad \text{multiple step reduction} \]

• Subject reduction
  \[ e \rightarrow^* e' \quad \text{and} \quad \emptyset \vdash e : A \Rightarrow \emptyset \vdash e' : A \]

• Corollary: well-typed expressions don’t get stuck.
  \[ 1 + \text{true} \text{ is a stuck expression} \]
Some basic type constructs

- products: \( A \times B \)
- records: \( \{ m_1 : A_1, \ldots, m_n : A_n \} \)
- sums: \( A + B \)
- functions: \( A \rightarrow B \)
- recursion: \( \mu t.A \)
- refs: \( \text{Ref } A \)
- subtypes: \( A <: B \)
Products

\[ A \times B = \{ (a, b) \mid a \in A, b \in B \} \]

\[
\begin{align*}
C \vdash e_1 : A & \quad C \vdash e_2 : B \\
\hline
C \vdash (e_1, e_2) : A \times B
\end{align*}
\]

(* Intro)

\[
\begin{align*}
C \vdash e : A \times B & \\
\hline
C \vdash \text{fst } e : A
\end{align*}
\]

(* Elim left)

\[
\begin{align*}
C \vdash e : A \times B & \\
\hline
C \vdash \text{snd } e : B
\end{align*}
\]

(* Elim right)
Products

\[ p = (1, \text{true}) : \text{Int} \times \text{Bool} \]

\[
\text{fst } p : \text{Int} \Rightarrow 1
\]

\[
\text{snd } p : \text{Bool} \Rightarrow \text{true}
\]

\[
\emptyset \vdash 1 : \text{Int} \quad \emptyset \vdash \text{true} : \text{Bool}
\]

\[
\emptyset \vdash (1, \text{true}) : \text{Int} \times \text{Bool}
\]
Products

Projections as primitive operations for each pair of types $A$ and $B$:

$$\text{fst}(A,B) : A \times B \rightarrow A$$

$$\text{snd}(A,B) : A \times B \rightarrow B$$

pairing as a primitive?

$$(\cdot, \cdot)(A,B) : A \rightarrow B \rightarrow A \times B \quad (?)$$
Records

record = labeled product

= finite map from labels to values

\[ r = \{ \text{age} = 13, \text{name} = "Bob" \} \]

record type = finite map from labels to types

\[ r : \{ \text{age} : \text{Int}, \text{name} : \text{String} \} \]

field = labeled component of a record

\[ r.\text{age} : \text{Int} \implies 13 \quad \text{(field selection)} \]
Typing records

Obvious generalization of product rules:

\[
\begin{align*}
C & \vdash e_i : A_i \quad (i=1,\ldots,n) \\
C & \vdash \{m_1=e_1,\ldots,m_n=e_n\} : \{m_1:A_1,\ldots,m_n:A_n\} \\
C & \vdash e : \{m_1:A_1,\ldots,m_n:A_n\} \\
C & \vdash e.m_i : A_i
\end{align*}
\]

(Record Intro)

(Record Elim)
**Sums**

\[ A + B \] - tagged union of \( A \) and \( B \)

\[
\begin{align*}
\text{inl: } & A \rightarrow A + B \\
\text{outl: } & A + B \rightarrow A \\
\text{isl: } & A + B \rightarrow \text{Bool} \\
\text{inr: } & B \rightarrow A + B \\
\text{outr: } & A + B \rightarrow B \\
\text{isr: } & A + B \rightarrow \text{Bool}
\end{align*}
\]

Actually, should be: \( \text{inl}(A, B) \), etc.
Sums

\[ i = \text{inl } 3 : \text{Int} \times \text{Bool} \quad (\text{inl}(\text{Int},\text{Bool})) \]

\[ b = \text{inr } \text{true} : \text{Int} \times \text{Bool} \]

\[ \text{isl } b : \text{Bool} \implies \text{false} \]

\[ \text{outl } a : \text{Int} \implies 1 \]
Could replace isl, isr, outl, outr with \textbf{case}

case (A, B, C) : (A+B) \times (A \rightarrow C) \times (B \rightarrow C) \rightarrow C

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

case (a,
  \text{fun}(n) (n+1),
  \text{fun}(b) (\text{if } b \text{ then } 3 \text{ else } 4)) \implies 2
Recursive types

\[ \mu t.A \] [or Rec(t)A, Fix(Fun(t)A)]

Equirecursive types:

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

\[ C |- e: \mu t.A \]
\[ C |- e: [\mu t.A/t]A \]

\[ C |- e: \mu t.A \]
\[ C |- e: [\mu t.A/t]A \]
**Isorecursive types**

\[ \mu t. A \leftrightarrow [\mu t. A/t]A \]

\[ C \vdash e : \mu t. A \]

\[ C \vdash \text{unfold } e : [\mu t. A/t]A \]  \hspace{1cm} \text{(Rec Elim)}

\[ C \vdash e : [\mu t. A/t]A \]

\[ C \vdash \text{fold } e : \mu t. A \]  \hspace{1cm} \text{(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 -> Int
Ref

Ref A -- mutable cells containing A values

\[ \text{ref}_A : A \rightarrow \text{Ref} A \]

\[ !_A : \text{Ref} A \rightarrow A \quad (\text{dereference}) \]

\[ :=_A : \text{Ref} A \times A \rightarrow \text{Unit} \]

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:

\[
\begin{align*}
  C \vdash e : A & \quad C \vdash A <: B \\
  \hline
  C \vdash e : B & \quad \text{(Subsumption)}
\end{align*}
\]

How is C relevant to A <: B?
Propagation of subtyping

• Products (monotonic)

\[ A_1 <: A_2 \quad B_1 <: B_2 \]

\[ A_1 \ast B_1 <: A_2 \ast B_2 \]

• Sums (monotonic)

\[ A_1 <: A_2 \quad B_1 <: B_2 \]

\[ A_1 + B_1 <: A_2 + B_2 \]
**Propagation of subtyping**

- function types

When is \( A_1 \rightarrow B_1 \triangleleft\rightarrow A_2 \rightarrow B_2 \) ?
Function subtyping

\[ f \in A \rightarrow B \text{ if } x \in A \Rightarrow f \, x \in B \]

Assume \( A_2 \prec\prec A_1 \) and \( B_1 \prec\prec B_2 \).
Assume \( f \in A_1 \rightarrow B_1 \) and \( x \in A_2 \).
Then \( x \in A_1 \Rightarrow f \, x \in B_1 \Rightarrow f \, x \in B_2 \).
Hence \( f \in A_2 \rightarrow B_2 \).
Subtyping function types

function types

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

\[
\begin{align*}
A_2 &\ <:\ A_1 &\ B_1 &\ <:\ B_2 \\
\hline
A_1 &\ ->\ B_1 &\ <:\ A_2 &\ ->\ B_2
\end{align*}
\]
Subtyping records

Depth subtyping (like products)

\[ A_i <: B_i \quad (i=1,\ldots,n) \]

\[ \{m_1:A_1,\ldots,m_n:A_n\} <: \{m_1:B_1,\ldots,m_n:B_n\} \]

\[ \text{Nat} <: \text{Int} \]

\[ \{\text{age: Nat, b: Bool}\} <: \{\text{age: Int, b: Bool}\} \]
Subtyping records 2

Width subtyping -- adding fields makes a subtype.

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

\{age: \text{Int}, \text{name: String}, \text{Id: Int}\} <: \{age: \text{Int}, \text{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 combined.

\{age: \texttt{Nat}, name: String, \texttt{Id}: \texttt{Int}\} <: \\
\{age: \texttt{Int}, name: String\}
Subtyping Refs

Ref types are invariant!

Thm: \( \text{Ref } A <: \text{Ref } B \Rightarrow A = B \)

\([B <: A]\) let \( r : \text{Ref } A \) and \( b : B \)
\( r : \text{Ref } B \) (assumption)
\( r := b \)
\( b = !r : A \)

\([A <: B]\) let \( r : \text{Ref } A \) and \( a : A \)
\( r : \text{Ref } B \) (assumption)
\( r := a \)
\( a = !r : B \)
Subtyping and recursion 1

- The “Amber” rule:

\[
\begin{align*}
C, s <: t & \vdash A <: B \\
\hline
C & \vdash \mu s.A <: \mu t.B
\end{align*}
\]
Subtyping and recursion 2

Example:

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

1. s <: t \Rightarrow 
2. Unit -> s <: Unit -> t \Rightarrow 
3. \{a: Nat, b: Unit -> s, c: Bool\} <: 
\{a: Int, b: Unit -> t\}
Subtyping and recursion 3

Exercise:

NatList = \( \mu t. (\text{Unit} + \text{Nat} \times t) \)

IntList = \( \mu t. (\text{Unit} + \text{Int} \times t) \)


List = Fun(s) \( \mu t. (\text{Unit} + s \times t) \)

2. Show that List is monotonic:

List s <: List t if s <: t.
Subtyping and recursion 4

fold and unfold rules (equirecursion)

\[ \mu s. F(s) = F(\mu s. F(s)) \] implies

\[ \mu s. F(s) <: F(\mu s. F(s)) \] (fold)

\[ F(\mu s. F(s)) <: \mu s. F(s) \] (unfold)
Parametric Polymorphism

Parameterize expressions over types

- Add type variables and quantified types to the type language

\[ A ::= \ldots | t | \text{All}(t)A \quad (\text{Cardelli style}) \]

\[ A ::= \ldots | t | \forall t.A \quad (\text{Classical}) \]

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

\[ e ::= \ldots | \text{Fun}(t)e | e[t] \]

\[ e ::= \ldots | \Lambda t.e | e[t] \]
Parametric Polymorphism: Examples

Examples

Polymorphic identity function

\[
\text{Id} = \text{Fun}(t) \text{fun}(x:t)x : \text{All}(t)t\rightarrow t
\]

\[
\text{Id} = \Lambda t.\lambda x:t.x : \forall t.t\rightarrow t
\]

\[
\text{Id[Int]} : \text{Int} \rightarrow \text{Int}; \text{Id[Int]3} : \text{Int}
\]

Self-application combinator (1st class polymorphism)

\[
\text{fun}(x:\text{All}(t)t\rightarrow t)(x[\text{All}(t)t\rightarrow t])x
\]

\[
: (\text{All}(t)t\rightarrow t) \rightarrow (\text{All}(t)t\rightarrow t)
\]
Parametric Polymorphism: Rules

Typing rules

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

\[ C \vdash e : \text{All}(t)A \]
\[ \frac{}{C \vdash e[B] : [B/t]A} \quad (\text{All Elim}) \]
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 \( \text{Fun}(t) \) works the same regardless of how \( t \) is instantiated (i.e. computations don’t depend on the identity of the type \( t \)).
  - Cor: \( \text{fun}(x) \ x \) is the unique (computable) member of the type \( \text{All} \ t \) \( (t \rightarrow t) \).
Polymorphic primitive operations

Primitive operations associated with type constructions can be polymorphic

\[
\begin{align*}
\text{fst: } & \text{All}(s)\text{All}(t)\ s \times t \rightarrow s \\
\text{inl: } & \text{All}(s)\text{All}(t)\ s \rightarrow s + t \\
\text{if\_then\_else: } & \text{All}(s)\ \text{Bool} \times s \times s \rightarrow s \\
\text{ref: } & \text{All}(s)(s \rightarrow \text{Ref}\ s) \\
\text{:=: } & \text{All}(s)(\text{Ref}\ s \times s \rightarrow \text{Unit})
\end{align*}
\]

but not \text{fold}, \text{unfold}, \text{r.m} !
Type functions

Type functions are defined by abstracting over type terms

\[ A ::= \ldots \mid \text{Fun}(t)A \mid A(B) \]

\[ \text{Pair} = \text{Fun}(t)(t\times t) \]

\[ \text{List} = \text{Fun}(s)\mu t. (\text{Unit} + s \times t) \]

\[ \text{null} = \text{Fun}(s)\text{fun}(x: \text{List } s) \]
\[ \quad \text{isl}[\text{Unit}][s\times \text{List}(s)] \]
\[ \quad (\text{unfold } x) \]
\[ \quad : \text{All}(s)(\text{List } s \rightarrow \text{Bool}) \]
Typing type functions: kinds

Types now need to be well-typed!

Types of type terms are called kinds.

\[ K ::= \text{Type} \mid \text{Type} \rightarrow \text{Type} \]

\begin{align*}
\text{Int} & : \text{Type} \\
\text{List} & : \text{Type} \rightarrow \text{Type} \\
\text{List(Int)} & : \text{Type} \\
* & : \text{Type} \rightarrow \text{Type} \rightarrow \text{Type} \\
\end{align*}

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
expressions

1: int
true: bool

1.0: real
“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
• 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)

\[
\text{val id = (fn x => x)}
\]

\[
\begin{align*}
\gamma &= \alpha \rightarrow \beta \\
\alpha &= \beta \\
\gamma &= \alpha \rightarrow \alpha
\end{align*}
\]

\[
(fn \ x \ => \ x) : \alpha \rightarrow \alpha, \text{ for any } \alpha
\]

\[
id : 'a \rightarrow 'a \quad (id : \forall t. t \rightarrow t)
\]
Type inference

Application expressions

\[ \alpha = \beta \rightarrow \gamma \]

\[ \text{Application expression diagram} \]
The let rule

- Polymorphism is introduced by let declarations

\[ C \vdash e : A \quad C, x : \forall X. A \vdash e' : B \]
\[ \frac{}{C \vdash \text{let val } x = e \text{ in } e' \text{ end} : B} \]

where \( x \) is the set of type variables free in \( A \) but not in \( C \): \( X = \text{FV}(A) - \text{FV}(C) \).

**Constraint:** all polymorphic types are prenex.

(so cannot type \((\text{fn } x \Rightarrow x \ x)\)
Polymorphic instantiation

- polymorphic types are implicitly instantiated

\[ C |- x : \forall t. A \]
\[ \quad \rightarrow \quad \]
\[ C |- x : [B/t]A \]

choose B canonically 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:

$$C |- e : A' \Rightarrow$$

$A'$ 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
  
  \( \text{fn } x \Rightarrow x : \ 'a \rightarrow 'a \)

  \( \text{fn } (x,y) \Rightarrow x : \ 'a \ast \ 'b \rightarrow 'a \)

- parametric datatype constructors

  \( \text{nil : 'a list} \)

  \( :: : 'a \ast 'a \text{ list } \rightarrow 'a \text{ list} \)
Polymorphism and refs

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

\( r : \forall t.((t\rightarrow t) \text{ ref}) \)
\( r : (\text{bool}\rightarrow\text{bool}) \text{ ref} \)
\( r : (\text{int}\rightarrow\text{int}) \text{ ref} \)

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^{\omega})

• 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

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)

(∀t.(t->t)) ref

type error!
Polymorphism and refs in System F

\[
\text{let } r = \text{Fun}(t)\text{ref}[t\rightarrow t](\text{fun}(x: t)x) \\
in r[\text{Bool}] := \text{not}; \\
([\text{Int}\rightarrow \text{Int}](r[\text{Int}]))(3) \\
\text{end}
\]

\[
r[\text{Bool}]: \text{Ref(Bool}-\rightarrow \text{Bool}) \quad \text{and}
\]
\[
r[\text{Int}]: \text{Ref(Int}-\rightarrow \text{Int})
\]

are two different ref values!
IV. Modules
Generalizing polymorphism

Example: polymorphic sort

\[
\text{sort} = \text{Fun}(t)\text{fun(less: } t*t\rightarrow\text{bool).} \ldots
\]

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?

\[ \text{sort} = \text{Fun}\langle t, \text{less: } t \times t \rightarrow \text{bool} \rangle \]. ... 

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.

```plaintext
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: thinning

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

```ocaml
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 $\text{SIG}_1 <: \text{SIG}_2$ if for any structure $S$, $S:\text{SIG}_1$ implies $S:\text{SIG}_2$ (i.e. any structure that matches $\text{SIG}_1$ will also match $\text{SIG}_2$).
Width subtyping of signatures

As for records, we can ignore extraneous elements.

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

<:

sig
  type t
  val a: t
end
```
Depth subtyping of signatures

There is a simple analog of depth subtyping.

```ml
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.

```plaintext
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.).

fun sig 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.

```plaintext
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))
```
functor $F(X:\text{sig type } t \text{ val } f: t\times t\rightarrow t \text{ end})$ : \text{sig type } u \text{ val } g: X.t\times u\rightarrow u \text{ end} =
\begin{verbatim}
struct
  type u = t list
  fun g(x:t,y:u) =
      map (fn z => f(z,x)) y
end
\end{verbatim}
Translation of functors in $F^{(0)}$

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

$F_t : \text{Type} \rightarrow \text{Type} = \text{Fun}(t:\text{Type})(\text{List } t)$

$F_v : \text{All}(t:\text{Type})\{f:t*t\rightarrow t\} \rightarrow$

$$\{g:t*F_t(t)\rightarrow F_t(t)\}$$

$$= \text{Fun}(t:\text{Type}) \text{fun} (\{f=f\} : \{f:t*t\rightarrow t\})$$

$$\{g=(\text{fun}(x:t,y:F_t(t))$$

$$(\text{map}[t][F_t(t)](\text{fun}(z:t)f(z,x))) \ y\}$$
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 $\mathcal{F}$ as a parameter) and give its $\mathcal{F}^{\omega}$ 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:
\[ F^{(\omega)} + <: + \text{Rec} + \text{Ref} + \text{with} + \ldots \]
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)

\[ \text{Rec}(t)I(t) \]

**Existential type** (Pierce, Turner)

\[ \text{Some}(t)I(t) \]

**Recursive-Existential types** (Bruce)

\[ \text{Rec}(t)\text{Some}(u)(u \times (u \rightarrow I(t))) \]

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

\[ \text{Rec}(t)\text{Some}(u<t)(u \times (u \rightarrow I(t))) \]
1. object as record of functions

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

\[
\{x = \text{ref } 0, \\
\quad \text{get} = \text{fun()}!x, \\
\quad \text{set} = \text{fun}(y:\text{Int}) (x:=y)\}
\]

: \{x: \text{Ref Int}, \text{get: Unit}\rightarrow\text{Int}, \\
\quad \text{set: Int}\rightarrow\text{Unit}\}

Bogus: \(x\) is free in bodies of \text{get}, \text{set}.
Cell: private variable \( x \)

We could make the variable \( x \) private:

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

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!)

\[ CI = \{x: \text{Ref Int}, \text{get: Unit} \rightarrow \text{Int}, \text{set: Int} \rightarrow \text{Unit}\} \]

\[ \text{Cell} = \text{fun}(\text{self: CI}) \]
\[ \{x = \text{ref} \ 0, \]
\[ \quad \text{get} = \text{fun}()!\text{self}.x, \]
\[ \quad \text{set} = \text{fun}(y:\text{Int})(\text{self}.x:=y)\} \]
\[ : \text{CI} \rightarrow \text{CI} \]

\[ c = \text{new Cell = Fix(\text{Cell}) = } \mu \text{self.Cell(self)} \]
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

\[ \text{CI2} = \text{CI} + \{\text{inc: Unit} \to \text{Unit}\} \]

\[ \text{Cell2 : Int} \to \text{CI2} \to \text{CI2} = \]
\[ \text{fun}(n: \text{Int})\text{fun}(self: \text{CI2}) \]
\[ \text{Cell}(n)(self) \text{ with} \]
\[ \{\text{inc} = \text{fun}()(self.x := !self.x+1)\} \]

Note: \text{Cell}(n)\text{self} is well-typed because \text{CI2} \text{ <: CI}. 
6. Record concatenation

We are using record concatenation to add the inc method.

\{{a: \text{Int}, b: \text{Bool}} + \{b: \text{Real}, c: \text{String}\} = \{a: \text{Int}, b: \text{Real}, c: \text{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!!

\[ e_1: \{a: \text{Int}\} \]
\[ e_2: \{a: \text{Bool}, b: \text{Int}\} \]

\[ e_1 \text{ with } e_2: \{a: \text{Bool}, b: \text{Int}\} \]

but \( \{a: \text{Bool}, b: \text{Int}\} <: \{b: \text{Int}\} \) implies \( e_2: \{b: \text{Int}\} \), so we also have

\[ e_1: \{a: \text{Int}\} \]
\[ e_2: \{b: \text{Int}\} \]

\[ e_1 \text{ with } e_2: \{a: \text{Int}, b: \text{Int}\} \]
Record concatenation and subtyping

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

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 subtype 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

$$CI = \mu t. \{ x: \text{Ref Int}, \text{get: Unit} \to \text{Int},$$
$$\text{set: Int} \to \text{Unit},$$
$$\text{me: Unit} \to t \}$$

$$= \text{Fix}(CIF)$$

where

$$CIF = \text{Fun}(t) \{ x: \text{Ref Int},$$
$$\text{get: Unit} \to \text{Int},$$
$$\text{set: Int} \to \text{Unit},$$
$$\text{me: Unit} \to t \}$$
9. Inheritance with recursive interface

Add inc method to cell with me method.

\[
\text{Cell2} : \text{Int} \rightarrow \text{CI2} \rightarrow \text{CI2} = \\
\text{fun}(n: \text{Int}) \text{fun}(self: \text{CI2}) \\
\quad \text{Cell}(n)(self) \text{ with} \\
\quad \{ \text{inc} = \text{fun}()(self.x := !
\quad \text{self.x+1}) \}
\]
Extending recursive interface

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

\[ CI_2 = CI + \{\text{inc: Unit} \rightarrow \text{Unit}\} \]
\[ = \{x: \text{Ref Int}, \text{get: -}, \text{set: -}, \]
\[ \quad \text{me: Unit} \rightarrow CI, \text{inc: Unit} \rightarrow \text{Unit}\} \]

\[ CI_2' = \mu t. (CI_F(t) + \{\text{inc: Unit} \rightarrow \text{Unit}\}) \]
\[ = \{x: \text{Ref Int}, \text{get: -}, \text{set: -}, \]
\[ \quad \text{me: Unit} \rightarrow CI_2', \text{inc: Unit} \rightarrow \text{Unit}\} \]
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.

\[ \text{Cell}(n) : \text{CI} \rightarrow \text{CI} \]

\[ \Rightarrow \text{Cell}(n) \text{self} : \text{CI} \ (\text{self}: \text{CI2} <: \text{CI}) \]

\[ \Rightarrow \text{body of Cell2} : \text{CI} + \{\text{inc}: \text{Unit} \rightarrow \text{Unit}\} \]

\[ \Rightarrow \text{Cell2} : \text{CI2} \rightarrow \text{CI2} \]

\[ \Rightarrow \text{me method}: \text{Unit} \rightarrow \text{CI} \ (\text{not CI2}) \]
Class Functions, Class Interface

Class A :

\[ A = \text{fun}(<\text{init}>))\text{fun}(\text{self}: \text{AI}) \]
\[ \text{<record expression>} \]
\[ : \text{InitTyA} \rightarrow \text{AI} \rightarrow \text{AI} \]

\[ \text{AIF} = \text{Fun}(t)\text{<record type>} \]

\[ \text{AI} = \text{Fix}(\text{AIF}) \]

\[ a = \text{Fix}(A \times_0) \]
Inheritance by deriving class function

Class B inherits from A:

\[ B = \text{fun}(\text{<init>}) \text{fun}(\text{self}: \text{BI}) \]
\[ \quad \text{A}(\text{<init’>})(\text{self}) \]
\[ \quad \text{with} \quad \text{<record expression>} \]
\[ \quad : \quad \text{InitTyB} \rightarrow \text{BI} \rightarrow \text{BI} \]

\[ \text{BIF} = \text{Fun}(t)(\text{AIF}(t) + \text{<record type>}) \]

\[ \text{BI} = \text{Fix}(\text{BIF}) \]

\[ b = \text{Fix}(\text{B} \times_1) \]
11. Method specialization

To get

\[
\text{Cell2: CI2'} \rightarrow \text{CI2'}
\]

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

We make \text{Cell} polymorphic, but with a new twist: \textit{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.

\[
\text{CIF} = \text{Fun}(t)\{x: \text{Ref Int}, \text{get}: -, \text{set}: -, \text{eq}: t \rightarrow \text{Bool}\}
\]

\[
\text{CI} = \text{Fix}(\text{CIF})
\]

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

\[
\text{Cell}: \text{Int} \rightarrow \text{CI} \rightarrow \text{CI}
\]
14. Inheritance with Binary Methods

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

\[
\text{CI2} = \text{CI} + \{b: \text{Ref Bool}, \text{eq: CI2} \rightarrow \text{bool}\}
\]

\[
\text{Cell2} = \text{fun}(n: \text{Int}) \text{fun}(\text{Self: CI2}) \text{Cell}(n)(self) \text{with}
\]

\[
\text{b = ref true,}
\]

\[
\text{eq = fun}(c: \text{CI2})(!self.x = !c.x \text{ and } !self.b = !c.b)
\]

Oops! \text{CI2} \text{<-> CI} because of the contravariant change in \text{eq}'s type. \textbf{Type error!}
Inheritance with binary methods

How about

\[ CI2' = \mu t. (CIF(t) + \{b: \text{Ref Bool}\}) \] ?

Again \( CI2' \not<: CI \) because of the contravariant occurrence of \( t \). Again \( \text{Cell}(n)(\text{self}) \) won't type check.

Bounded quantifier trick won't help here because \( CI2' \not<: CI \).
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)
Cell2[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

\[ t <: CIF2(t) = CIF(t) + \{b: \text{Ref Bool}\} \Rightarrow t <: CIF(t) \]

c2 : CI2 = new (Cell2[CI2] 3)
where CI2 = Rec(CIF2)

CI2 = Fix(CIF2) \(\subseteq\) Fix(CIF) = CI

so inheritance does not imply subtypes, but

CI2 <: CIF(CI2)
Object protocols

Constraints like

\[ t <: F(t) \]

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

\[ t <: ut.F(t) \]
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:

\[
\text{CELL} = \text{Fun}(t <: \text{CIF}(t)) \\
\quad \text{fun}(\text{myclass}: \text{Int} -> t -> t) \\
\quad \quad \text{fun}(n: \text{Int}) \\
\quad \quad \quad \text{fun}(\text{self}: t) \{ x = \ldots, \\
\quad \quad \quad \quad \text{double} = \\
\quad \quad \quad \quad \quad \text{fun}() \text{new}(\text{myclass}(2*!\text{self}.x)) \}
\]

\[
\text{CELL}: \text{All}(t <: \text{CIF}(t))(\text{Int} -> t -> t) -> \\
\quad (\text{Int} -> t -> \text{CIF}(t))
\]

Cell = Fix(CELL[CI])

\(c = \text{new}(\text{Cell} \ 3)\)
Inheritance with Class Recursion

Add a new boolean field to Cell.

\[
CIF2 = \text{Fun}(t)(CIF(t) + \{b: \text{Bool}\})
\]

\[
\text{CELL2} = \text{Fun}(t<:CIF2(t))
\]

\[
\begin{aligned}
&\text{fun}(\text{myclass}: \text{Int} \times \text{Bool} \to t \to t) \\
&\text{fun}(n:\text{Int}, b:\text{Bool}) \\
&\text{fun}(\text{self}:t) \\
&\quad (CELL[t] \\
&\quad (\text{fun}(x:\text{Int})(\text{myclass}(x,\text{self}.b))) \\
&\quad (n)(\text{self})) \\
&\quad + \{b = b\}
\end{aligned}
\]
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
- methodological 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

\[
O_1 = \{ x = \text{fun}()1, \ y = \text{fun}()(2*\text{self}.x()) \}
\]
\[
O_2 = \{ x = \text{fun}()1, \ y = \text{fun}()2 \}
\]
\[
O_1.x() \implies 1 \quad O_1.y() \implies 2
\]
\[
O_2.x() \implies 1 \quad O_2.y() \implies 2
\]

\[
O_1' = O_1 + \{ x = \text{fun}()2 \}
\]
\[
O_2' = O_2 + \{ x = \text{fun}()2 \}
\]
\[
O_1'.x() \implies 2 \quad O_1'.y() \implies 4
\]
\[
O_2'.x() \implies 2 \quad O_2'.y() \implies 2
\]
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:
  \[ \forall \rho. \{a: \text{int}, b: \text{bool}, \rho\} \rightarrow \text{int} \]
- 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
\(F_{<,\text{rec}}\)

- 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