On Subtyping and Matching

Luca Cardelli

(joint work with Martín Abadi)

Digital Equipment Corporation
Systems Research Center

ATSC'95

INTRODUCTION

- The subtyping relation between object types is the foundation of subclassing and inheritance . . . when it holds.
- Subtyping fails to hold between certain types that arise naturally in object-oriented programming. Typically, recursively defined object types with binary methods.
- F-bounded subtyping was invented to solve this kind of problem.
- A new programming construction, called "matching" has been proposed to solve the same problem, inspired by F-bounded subtyping.
- Matching achieves "covariant subtyping" for Self types. Contravariant subtyping still applies, otherwise.
- We argue that matching is a good idea, but that it should not be based on F-bounded subtyping. We show that a new interpretation of matching, based on higher-order subtyping, has better properties.

O-O PROGRAMMING

- Goals
  - ~ Data abstraction.
  - ~ Polymorphism.
  - ~ Code reuse.
- Mechanisms
  - ~ Objects with *self* (packages of data and code).
  - ~ Subtyping and subsumption.
  - ~ Classes and inheritance.

Object-oriented constructs

Objects and object types

Objects are packages of data (*instance variables*) and code (*methods*).

Object types describe the shape of objects.

```
ObjectType CellType;
    var contents: Integer;
    method get(): Integer;
    method set(n: Integer);
end;

object cell: CellType;
    var contents: Integer := 0;
    method get(): Integer; return self.contents end;
    method set(n: Integer); self.contents := n end;
end;
```

where $a : A$ means that the program a has type A. So, $cell : CellType$.

Classes

Classes are ways of describing and generating collections of objects.

```
class cellClass for CellType;
    var contents: Integer := 0;
    method get(): Integer; return self.contents end;
    method set(n: Integer); self.contents := n end;
end;

var cell : CellType := new cellClass;

procedure double(aCell: CellType);
    aCell.set(2 * aCell.get());
end;
```


Subclasses

Subclasses are ways of describing classes incrementally, reusing code.

```
ObjectType ReCellType;
    var contents: Integer;
    var backup: Integer;
    method get(): Integer;
    method set(n: Integer);
    method restore();
end;

subclass reCellClass of cellClass for ReCellType;          (Inherited:
    var backup: Integer := 0;                                var contents
    override set(n: Integer);                                method get)
        self.backup := self.contents;
        super.set(n);
    end;
    method restore(); self.contents := self.backup end;
end;
```


Subtyping and subsumption

- Subtyping relation, $A <: B$

  An object type is a subtype of any object type with fewer components.

  (e.g.: *ReCellType* <: *CellType*)

- Subsumption rule

  if $a : A$ and $A <: B$ then $a : B$

  (e.g.: *reCell* : *CellType*)

- Subclass rule

  cClass can be a subclass of *dClass* only if *cType* <: *dType*

  (e.g.: *reCellClass* can indeed be declared as a subclass of *cellClass*)

Healthy skepticism

- Object-oriented languages have been plagued, more than any other kind of languages, but confusion and unsoundness.
- How do we keep track of the interactions of the numerous object-oriented features?
- How can we be sure that it all makes sense?

WHEN SUBTYPING WORKS

- A simple and successful treatment of object, classes, and inheritance, for covariant Self types only.

Object Types

- Consider two types Inc and IncDec containing an integer field and some methods:

$$\begin{aligned} \text{Inc} &\triangleq \mu(X)[n:\text{Int}, \text{inc}^+:X] \\ \text{IncDec} &\triangleq \mu(Y)[n:\text{Int}, \text{inc}^+:Y, \text{dec}^+:Y] \end{aligned}$$

- A typical object of type Inc is:

$$\begin{aligned} &p : \text{Inc} \triangleq \\ &\quad [n = 0, \\ &\quad\ \text{inc} = \varsigma(\text{self: Inc})\ \text{self.n} := \text{self.n} + 1] \end{aligned}$$

Subtyping

- Subtyping (<:) is a reflexive and transitive relation on types, with *subsumption*:

$$\text{if}\quad a : A \quad\text{and}\quad A <: B \quad\text{then}\quad a : B$$

- For object types, we have the subtyping rule:

$$[v_i{:}B_i^{\ i\in I}, m_j^+{:}C_j^{\ j\in J}] \quad <: \quad [v_i{:}B_i^{\ i\in I'}, m_j^+{:}C_j'^{\ j\in J'}]$$
$$\text{if } C_j <: C_j' \text{ for all } j\in J', \text{ with } I'\subseteq I \text{ and } J'\subseteq J$$

- For recursive types we have the subtyping rule:

$$\mu(X)A\{X\} <: \mu(Y)B\{Y\}$$
$$\text{if } X <: Y \text{ implies } A\{X\} <: B\{Y\}$$

- Combining them, we obtain a derived rule for recursive object types:

$$\mu(X)[v_i{:}B_i^{\ i\in I}, m_j^+{:}C_j\{X\}^{\ j\in J}] \quad <: \quad \mu(Y)[v_i{:}B_i^{\ i\in I'}, m_j^+{:}C_j'\{Y\}^{\ j\in J'}]$$
$$\text{if } X <: Y \text{ implies } C_j\{X\} <: C_j'\{Y\} \text{ for all } j\in J', \text{ with } I'\subseteq I \text{ and } J'\subseteq J$$

- By applying this derived rule to our example, we obtain:

$$\text{IncDec} <: \text{Inc}$$

Pre-Methods

- The subtyping relation (e.g. IncDec <: Inc) plays an important role in inheritance.
- Inheritance is obtained by reusing polymorphic code fragments.

$$\begin{aligned} &\text{pre-inc} : \forall(X<:\text{Inc})X\to X \triangleq \\ &\quad \lambda(X<:\text{Inc})\ \lambda(\text{self:}X)\ \text{self.n} := \text{self.n}+1 \end{aligned}$$

- We call a code fragment such as pre-inc a *pre-method*.
- We can specialize pre-inc to implement the method inc of type Inc or IncDec:

$$\begin{aligned} &\text{pre-inc(Inc)} : \text{Inc}\to\text{Inc} \\ &\text{pre-inc(IncDec)} : \text{IncDec}\to\text{IncDec} \end{aligned}$$

- Thus, we have reused pre-inc at different types, without retypechecking its code.

Classes

- Pre-method reuse can be systematized by collecting pre-methods into *classes*.
- A class for an object type A can be described as a collection of pre-methods and initial field values, plus a way of generating new objects of type A.
- In a class for an object type A, the pre-methods are parameterized over all subtypes of A, so that they can be reused (inherited) by any class for any subtype of A.

- Let A be a type of the form $\mu(X)[v_i{:}B_i\ ^{i\in I}, m_j{}^{+}{:}C_j\{X\}\ ^{j\in J}]$. As part of a class for A, a pre-method for m_j would have the type $\forall(X<:A)X\rightarrow C_j\{X\}$. For example:

  IncClass $\triangleq$
  [new^{+}: Inc,
  n: Int,
  inc: $\forall(X<:\text{Inc})X\rightarrow X$]

  IncDecClass $\triangleq$
  [new^{+}: IncDec,
  n: Int,
  inc: $\forall(X<:\text{IncDec})X\rightarrow X$,
  dec: $\forall(X<:\text{IncDec})X\rightarrow X$]

- A typical class of type IncClass reads:

  incClass : IncClass $\triangleq$
  [new = ς(classSelf: IncClass)
  [n = classSelf.n, inc = ς(self:Inc) classSelf.inc(Inc)(self)]
  n = 0,
  inc = pre-inc]

  The code for new is uniform: it assembles all the pre-methods into a new object.

Inheritance

- Inheritance is obtained by extracting a pre-method from a class and reusing it for constructing another class.

  For example, the pre-method pre-inc of type $\forall(X<:\text{Inc})X\rightarrow X$ in a class for Inc could be reused as a pre-method of type $\forall(X<:\text{IncDec})X\rightarrow X$ in a class for IncDec:

  incDecClass : IncDecClass $\triangleq$
  [new = ς(classSelf: IncDecClass)[...],
  n = 0,
  inc = incClass.inc,
  dec = ...]

- This example of inheritance requires the subtyping:

  $\forall(X<:\text{Inc})X\rightarrow X \ <: \ \forall(X<:\text{IncDec})X\rightarrow X$

  which follows from the subtyping rules for quantified types and function types:

  $\forall(X<:A)B <: \forall(X<:A')B'$ if $A'<:A$ and if $X<:A$ implies $B<:B'$
  $A\rightarrow B <: A'\rightarrow B'$ if $A' <: A$ and $B <: B'$

Inheritance from Subtyping

- In summary, inheritance from a class for Inc to a class for IncDec is enabled by the subtyping IncDec <: Inc.
- Unfortunately, inheritance is possible and desirable even in situations where such subtypings do not exist. These situations arise with binary methods.

Binary Methods

- Consider a recursive object type Max, with a field n and a binary method max.

  $$\text{Max} \triangleq \mu(X)[n:\text{Int}, \max^{+}:X \to X]$$

  Consider also a type MinMax with an additional binary method min:

  $$\text{MinMax} \triangleq \mu(Y)[n:\text{Int}, \max^{+}:Y \to Y, \min^{+}:Y \to Y]$$

- Problem:

  $$\text{MinMax} \not<: \text{Max}$$

  according to the rules we have adopted. Moreover, it would be unsound to assume MinMax <: Max.
- Hence, the development of classes and inheritance developed for Inc and IncDec falters in presence of binary methods.

LOOKING FOR A NEW RELATION

- A possible replacement for subtyping: *matching*.
  (Presented semi-formally.)

Matching

- Recently, Bruce *et al.* proposed axiomatizing a relation between recursive object types, called *matching*.
- We write A <# B to mean that A matches B; that is, that A is an "extended version" of B. We expect to have, for example:

  IncDec <# Inc
  MinMax <# Max

- In particular, we may write X <# A, where X is a variable. We may then quantify over all types that match a given one, as follows:

  $$\forall(X<\#A)B\{X\}$$

  We call $\forall(X<\#A)B$ *match-bounded quantification*, and say that occurrences of X in B are *match-bound*.

- Using match-bounded quantification, we can rewrite the polymorphic function pre-inc in terms of matching rather than subtyping:

  pre-inc : $\forall(X<\#\text{Inc})X \to X \triangleq$
  $\lambda(X<\#\text{Inc})\ \lambda(\text{self}:X)\ \text{self.n} := \text{self.n}+1$

  pre-inc(IncDec) : IncDec→IncDec

- Similarly, we can write a polymorphic version of the function pre-max:

  pre-max : $\forall(X<\#\text{Max})X \to X \to X \triangleq$
  $\lambda(X<\#\text{Max})\ \lambda(\text{self}:X)\ \lambda(\text{other}:X)$
  if self.n>other.n then self else other

  pre-max(MinMax) : MinMax→MinMax→MinMax

- Thus, the use of match-bounded quantification enables us to express the polymorphism of both pre-max and pre-inc: contravariant and covariant occurrences of Self are treated uniformly.

Matching and Subsumption

- A subsumption-like property does not hold for matching; A <# B is not quite as good as A <: B. (Fortunately, subsumption was not needed in the examples above.)

  a : A and A <# B need not imply a : B

- Thus, matching cannot completely replace subtyping. For example, forget that IncDec <: Inc and try to get by with IncDec <# Inc. We could not typecheck:

  inc : Inc→Inc ≜
  λ(x:Inc) x.n := x.n+1
  λ(x:IncDec) inc(x)

  We can circumvent this difficulty by turning inc into a polymorphic function of type ∀(X<#Inc)X→X, but this solution requires foresight, and is cumbersome:

  pre-inc : ∀(X<#Inc)X→X ≜
  λ(X<#Inc) λ(x:X) x.n := x.n+1
  λ(x:IncDec) pre-inc(IncDec)(x)

Matching and Classes

- We can now revise our treatment of classes, adapting it for matching.

  MaxClass ≜
  [new$^+$: Max,
  n: Int,
  max: ∀(X<#Max)X→X→X]
  MinMaxClass ≜
  [new$^+$: MinMax,
  n: Int,
  max: ∀(X<#MinMax)X→X→X,
  min: ∀(X<#MinMax)X→X→X]

- A typical class of type MaxClass reads:

  maxClass : MaxClass ≜
  [new = ς(classSelf: MaxClass)
  [n = classSelf.n, max = ς(self:Max) classSelf.max(Max)(self)],
  n = 0,
  max = pre-max]

Matching and Inheritance

- A typical (sub)class of type MinMaxClass reads:

  minMaxClass : MinMaxClass ≜
  [new = ς(classSelf: MinMaxClass)[...],
  n = 0,
  max = maxClass.max,
  min = ...]

- The implementation of max is taken from maxClass, that is, it is inherited. The inheritance typechecks assuming that

  ∀(X<#Max)X→X→X <: ∀(X<#MinMax)X→X→X

- Thus, we are still using some subtyping and subsumption as a basis for inheritance.

Advantages of Matching

Matching is attractive

- The fact that MinMax matches Max is reasonably intuitive.
- Matching handles contravariant Self and inheritance of binary methods.
- Matching is meant to be directly axiomatized as a relation between types.The typing rules of a programming language that includes matching can be explained directly.
- Matching is simple from the programmer's point to of view, in comparison with more elaborate type-theoretic mechanisms that could be used in its place.

However...

- The notion of matching is ad hoc (e.g., is defined only for object types).
- We still have to figure out the exact typing rules and properties matching.
- The rules for matching vary in subtle but fundamental ways in different languages.
- What principles will allow us to derive the "right" rules for matching?

MATCHING AS F-BOUNDED SUBTYPING

- An attempt to formalize matching as F-bounded subtyping.

Type Operators

- We introduce a theory of type operators that will enable us to express various formal relationships between types. Alternatives interpretations of matching will become available.
- A type operator is a function from types to types.

$\lambda(X)B\{X\}$	maps each type X to a corresponding type B{X}
$B(A)$	applies the operator B to the type A

  $(\lambda(X)B\{X\})(A) = B\{A\}$

- Notation for fixpoints:

F^*	abbreviates	$\mu(X)F(X)$	
A_{Op}	abbreviates	$\lambda(X)D\{X\}$	whenever $A \equiv \mu(X)D\{X\}$

- We obtain:

  $Max_{Op} \equiv \lambda(X)[n:Int, max^+:X\to X]$
  $MinMax_{Op} \equiv \lambda(Y)[n:Int, max^+:Y\to Y, min^+:Y\to Y]$

- The unfolding property of recursive types yields:

  $Max_{Op}{}^* = \mu(X)\, Max_{Op}(X) = \mu(X)\,[n:Int, max^+:X\to X] = Max$
  $Max_{Op}{}^* = Max_{Op}(\mu(X)\, Max_{Op}(X)) = Max_{Op}(Max)$

- Note that A_{Op} is defined in terms of the syntactic form $\mu(X)D\{X\}$ of A. In particular, the unfolding D{A} of A is not necessarily in a form such that $D\{A\}_{Op}$ is defined. Even if $D\{A\}_{Op}$ is defined, it need not equal A_{Op}. For example, consider:

  $D\{X\} \triangleq \mu(Y)\, X\to Y$
  $A \triangleq \mu(X)\, D\{X\}$
  $D\{A\} \equiv \mu(Y)\, A\to Y = A$
  $A_{Op} \equiv \lambda(X)\, D\{X\}$
  $D\{A\}_{Op} \equiv \lambda(Y)\, A\to Y \neq A_{Op}$

- Thus, we may have two types A and B such that $A = B$ but $A_{Op} \neq B_{Op}$ (when recursive types are taken equal up to unfolding). This is a sign of trouble to come.

F-bounded Subtyping

- F-bounded subtyping was invented to support parameterization in the absence of subtyping.
- The property:

  $A <: B_{Op}(A)$ (A is a pre-fixpoint of B_{Op})

  is seen as a statement that A extends B.
- This view is justified because, for example, a recursive object type A such that $A <: [n:Int, max^+:A\to A]$ often has the shape $\mu(Y)[n:Int, max^+:Y\to Y, \ldots]$.

- Both Max and MinMax are pre-fixpoints of Max_{Op}:

  Max <: Max_{Op}(Max) (= Max)
  MinMax (= [n:Int, max^+:MinMax→MinMax, min^+: ...])
  <: Max_{Op}(MinMax) (= [n:Int, max^+:MinMax→MinMax])

  So, we can parameterize over all types X with the property that X <: Max_{Op}(X).

  ∀(X<:Max_{Op}(X))B{X}

  This form of parameterization leads to a general typing of pre-max, and permits the inheritance of pre-max:

  pre-max : ∀(X<:Max_{Op}(X))X→X→X ≜
  λ(X<:Max_{Op}(X)) λ(self:X) λ(other:X)
  if self.n>other.n then self else other
  pre-max(Max) : Max→Max→Max
  pre-max(MinMax) : MinMax→MinMax→MinMax

The F-bounded Interpretation

- The central idea of the interpretation is:

  A <# B ≈ A <: B_{Op}(A)
  ∀(X<#A)B{X} ≈ ∀(X<:A_{Op}(X))B{X}

- However, this interpretation is not defined when the right-hand side of <# is a variable, as in the case of cascading quantifiers:

  ∀(X<#A) ∀(Y<#X) ... ≈ ?

  Since ∀(X<:A_{Op}(X)) ∀(Y<:X_{Op}(Y)) ... does not make sense the type structure supported by this interpretation is somewhat irregular: type variables are not allowed in places where object types are allowed.

Reflexivity and Transitivity

- We would expect A <# A to hold, e.g. to justifying the instantiation f(A) of a polymorphic function f : ∀(X<#A)B. We have:

  A <# A ≈ A <: A_{Op}(A)

  with A = A_{Op}(A) by the unfolding property of recursive types. However, if A is a type variable X, then X_{Op} is not defined, so X <: X_{Op}(X) does not make sense. Hence, reflexivity does not hold in general.

- If A, B, and C are object types of the source language, then we would expect that A <# B and B <# C imply A <# C; this would mean:

  A <: B_{Op}(A) and B <: C_{Op}(B) imply A <: C_{Op}(A)

  As in the case of reflexivity, we run into difficulties with type variables.

- Worse, transitivity fails even for closed types, with the following counterexample:

  A ≜ μ(X)[p^+: X→Int, q: Int]
  B ≜ μ(X)[p^+: X→Int]
  C ≜ μ(X)[p^+: B→Int]

  We have both A <# B and B <# C, but we do not have A <# C (because [p^+:A→Int, q:Int] <: [p^+:B→Int] fails).

  A = [p^+: A→Int, q: Int] <:
  B_{Op}(A) = [p^+: A→Int]

  B = [p^+: B→Int] <:
  C_{Op}(B) = [p^+: B→Int]

  A = [p^+: A→Int, q: Int] $\not<:$
  C_{Op}(A) = [p^+: B→Int]

- We can trace this problem back to the definition of D_{Op}, which depends on the exact syntax of the type D. Because of the syntactic character of that definition, two equal types may behave differently with respect to matching.

  In our example, we have B = C by the unfolding property of recursive types. Despite the equality B = C, we have A <# B but not A <# C !

Matching Self

- According to the F-bounded interpretation, two types that look rather different may match. Consider two types A and A′ such that:

  $$A \equiv \mu(X)[v_i{:}B_i\ ^{i\in I},\ m_j^+{:}C_j\{X\}\ ^{j\in J}]$$
  $$<\#\ \mu(X)[v_i{:}B_i\ ^{i\in I},\ m_j^+{:}C_j'\{X\}\ ^{j\in J'}] \equiv A'$$

  This holds when $A <: A'_{Op}(A)$, that is, when $[v_i{:}B_i\ ^{i\in I},\ m_j^+{:}C_j\{A\}\ ^{j\in J}] <: [v_i{:}B_i\ ^{i\in I},\ m_j^+{:}C_j'\{A\}\ ^{j\in J'}]$. It suffices that, for every $j\in J'$:

  $$C_j\{A\} <: C_j'\{A\}$$

- For example, we have:

  $$\mu(X)[v{:}Int,\ m^+{:}X] <\#\ \mu(X)[m^+{:}\ [v{:}Int]]$$

  The variable X on the left matches the type [v:Int] on the right. Since X is the Self variable, we may say that Self matches not only Self but also other types (here [v:Int]). This treatment of Self is both sound and flexible. On the other hand, it can be difficult for a programmer to see whether two types match.

MATCHING AS HIGHER-ORDER SUBTYPING

- A formalization of matching as higher-order subtyping.
- Inheritance of binary methods.

Higher-Order Subtyping

- Subtyping can be extended to operators, in a pointwise manner:

  $F <: G$ if, for all X, $F(X) <: G(X)$

- The property:

  $A_{Op} <: B_{Op}$ (A_{Op} is a suboperator of B_{Op})

  is seen as a statement that A extends B.

- We obtain:

$\text{Max}_{\text{Op}} \prec: \text{Max}_{\text{Op}}$
$\text{MinMax}_{\text{Op}} \quad (\forall X. \quad [n:\text{Int}, \text{max}^+:X\to X, \text{min}^+: X\to X]$
$\quad \prec: \text{Max}_{\text{Op}} \quad <: \quad [n:\text{Int}, \text{max}^+:X\to X])$

We can parameterize over all type operators X with the property that $X \prec: \text{Max}_{\text{Op}}$.

$\forall(X\prec:\text{Max}_{\text{Op}})B\{X\}$

We need to be careful about how X is used in B{X}, because X is now a type operator.

The idea is to take the fixpoint of X wherever necessary.

$\text{pre-max} : \forall(X\prec:\text{Max}_{\text{Op}})X^*\to X^*\to X^* \triangleq$
$\quad \lambda(X\prec:\text{Max}_{\text{Op}})\ \lambda(\text{self}:X^*)\ \lambda(\text{other}:X^*)$
$\quad\quad$ if self.n>other.n then self else other
$\text{pre-max}(\text{MinMax}_{\text{Op}}) : \text{MinMax}\to\text{MinMax}\to\text{MinMax}$

This typechecks, e.g.:

$X = X(X^*)$
$X \prec: \text{Max}_{\text{Op}} \Rightarrow X(X^*) <: \text{Max}_{\text{Op}}(X^*)$
$\text{self} : X^* \Rightarrow \text{self} : X(X^*) \Rightarrow \text{self} : \text{Max}_{\text{Op}}(X^*) \Rightarrow \text{self.n} : \text{Int}$

(In this derivation we have used the unfolding property $X^*=X(X^*)$, but we can do without it by introducing explicit fold/unfold terms.)

The Higher-Order Interpretation

- The central idea of the interpretation is:

$A <\# B \approx A_{\text{Op}} \prec: B_{\text{Op}}$
$\forall(X<\#A)B\{X\} \approx \forall(X\prec:A_{\text{Op}})B\{X^*\}$ (not quite)

We must be more careful about the B{X*} part, because X may occur both in type and operator contexts.

- We handle this problem by two translations for the two kinds of contexts:

$A <\# B \approx \mathit{Oper}\langle A\rangle \prec: \mathit{Oper}\langle B\rangle$
$\forall(X<\#A)B \approx \forall(X\prec:\mathit{Oper}\langle A\rangle)\mathit{Type}\langle B\rangle$

- The two translations, $\mathit{Type}\langle A\rangle$ and $\mathit{Oper}\langle A\rangle$, can be summarized as follows.

For object types of the source language, we set:

$\mathit{Oper}\langle X\rangle \approx$ (assuming that X is match-bound)
$\quad X$
$\mathit{Oper}\langle\mu(X)[v_i:B_i\ ^{i\in I}, m_j^+:C_j\{X\}\ ^{j\in J}]\rangle \approx$
$\quad \lambda(X)[v_i:\mathit{Type}\langle B_i\rangle\ ^{i\in I}, m_j^+:\mathit{Type}\langle C_j\{X\}\rangle\ ^{j\in J}]$

$\mathit{Type}\langle X\rangle \approx$ (when X is match-bound)
$\quad X^*$
$\mathit{Type}\langle\mu(X)[v_i:B_i\ ^{i\in I}, m_j^+:C_j\{X\}\ ^{j\in J}]\rangle \approx$
$\quad \mu(X)[v_i:\mathit{Type}\langle B_i\rangle\ ^{i\in I}, m_j^+:\mathit{Type}\langle C_j\{X\}\rangle\ ^{j\in J}]$

For other types, we set:

$\mathit{Type}\langle X\rangle \approx X$ (when X is not match-bound)
$\mathit{Type}\langle A\to B\rangle \approx \mathit{Type}\langle A\rangle\to\mathit{Type}\langle B\rangle$
$\mathit{Type}\langle\forall(X<\#A)B\rangle \approx \forall(X\prec:\mathit{Oper}\langle A\rangle)\mathit{Type}\langle B\rangle$

- For instance:

$$Type\langle \forall(X<\#Max)\ \forall(Y<\#X)\ X\rightarrow Y\rangle \approx \forall(X<:Max_{Op})\ \forall(Y<:X)\ X^*\rightarrow Y^*$$

This translation is well-defined on type variables, so there are no problems with cascading quantifiers.

- A note about unfolding of recursive types:
  - ~ The higher-order interpretation does not use the unfolding property of recursive types for the *target* language; instead, it uses explicit fold and unfold primitives.
  - ~ On the other hand, the higher-order interpretation is incompatible with the unfolding property of recursive types in the *source* language, because $Oper\langle\mu(X)A\{X\}\rangle$ and $Oper\langle A\{\mu(X)A\{X\}\}\rangle$ are in general different type operators.
  - ~ Technically, the unfolding property of recursive types is not an essential feature and it is the origin of complications; we are fortunate to be able to drop it throughout.

Reflexivity and Transitivity

- Reflexivity is now satisfied by all object types, including variables; for every object type A, we have:

$$A <\# A \quad \approx \quad Oper\langle A\rangle <: Oper\langle A\rangle$$

This follows from the reflexivity of $<:$.

- Similarly, transitivity is satisfied by all triples A,B, and C of object types, including variables:

$$A <\# B \text{ and } B <\# C \text{ imply } A <\# C \approx Oper\langle A\rangle <: Oper\langle B\rangle \text{ and } Oper\langle B\rangle <: Oper\langle C\rangle \text{ imply } Oper\langle A\rangle <: Oper\langle C\rangle$$

This follows from the transitivity of $<:$.

Matching Self

- With the higher-order interpretation, the relation:

$$A \equiv \mu(Self)[v_i{:}B_i\ ^{i\in I},\ m_j^+{:}C_j\{Self\}\ ^{j\in J}] <\# \mu(Self)[v_i{:}B_i\ ^{i\in I},\ m_j^+{:}C_j'\{Self\}\ ^{j\in J'}] \equiv A'$$

holds when the type operators corresponding to A and A′ are in the subtyping relation, that is, when:

$$[v_i{:}Type\langle B_i\rangle\ ^{i\in I},\ m_j^+{:}Type\langle C_j\{Self\}\rangle\ ^{j\in J}] <: [v_i{:}Type\langle B_i\rangle\ ^{i\in I},\ m_j^+{:}Type\langle C_j'\{Self\}\rangle\ ^{j\in J'}] \qquad \text{for an arbitrary Self}$$

For this, it suffices that, for every j in J′:

$$Type\langle C_j\{Self\}\rangle <: Type\langle C_j'\{Self\}\rangle$$

Since Self is μ-bound, all the occurrences of Self are translated as Self*. Then, an occurrence of Self* on the left can be matched only by a corresponding occurrence of Self* on the right, since Self is arbitrary. In short,:

Self matches only itself.

This property makes it easy for programmers to glance at two object types and tell whether they match.

Inheritance and Classes via Higher-Order Subtyping

- Applying our higher-order translation to MaxClass, we obtain:

$$\begin{array}{l}\text{MaxClass} \triangleq \\ \quad [\text{new}^{+}\text{: Max}, \\ \quad\ \text{n: Int}, \\ \quad\ \text{max: } \forall(X<:\text{Max}_{Op})X^{*}\to X^{*}\to X^{*}]\end{array}$$

The corresponding translation at the term level produces:

$$\begin{array}{l}\text{maxClass : MaxClass} \triangleq \\ \quad [\text{new} = \varsigma(\text{classSelf: MaxClass}) \\ \qquad \text{fold}(\\ \qquad\quad [\text{n} = \text{classSelf.n}, \\ \qquad\quad\ \text{max} = \varsigma(\text{self:Max}_{Op}(\text{Max})) \\ \qquad\qquad\qquad \text{classSelf.max}(\text{Max}_{Op})(\text{fold(self)})]), \\ \quad \text{n} = 0, \\ \quad \text{max} = \text{pre-max}]\end{array}$$

$$\begin{array}{l}\text{pre-max} : \forall(X<:\text{Max}_{Op})X^{*}\to X^{*}\to X^{*} \triangleq \\ \quad \lambda(X<:\text{Max}_{Op})\ \lambda(\text{self:}X^{*})\ \lambda(\text{other:}X^{*}) \\ \qquad \text{if unfold(self).n>unfold(other).n then self else other}\end{array}$$

It is possible to check that pre-max is well typed.

The instantiations pre-max(Max_{Op}) and pre-max(MinMax_{Op}) are both legal. Since pre-max has type $\forall(X<:\text{Max}_{Op})X^{*}\to X^{*}\to X^{*}$, this pre-method can be used as a component of a class of type MaxClass.

Moreover, a higher-order version of the rule for quantifier subtyping yields:

$$\forall(X<:\text{Max}_{Op})X^{*}\to X^{*}\to X^{*} \ <: \ \forall(X<:\text{MinMax}_{Op})X^{*}\to X^{*}\to X^{*}$$

so pre-max has type $\forall(X<:\text{MinMax}_{Op})X^{*}\to X^{*}\to X^{*}$ by subsumption, and hence pre-max can be reused as a component of a class of type MinMaxClass.

- Note. We expect following typings:

  if X<#Inc and x:X then x.n : Int
  if X<#Inc and x:X and b:Int then x.n:=b : X

The higher-order interpretation induces the following term translations:

  if $X<:\text{Inc}_{Op}$ and x:X* then unfold(x).n : Int
  if $X<:\text{Inc}_{Op}$ and x:X* and b:Int then fold(unfold(x).n:=b) : X*

For the first typing, we have unfold(x):X(X*). Moreover, from $X<:\text{Inc}_{Op}$ we obtain $X(X^{*}) <: \text{Inc}_{Op}(X^{*}) = [\text{n:Int, inc:}X^{*}]$. Therefore, unfold(x):[n:Int, inc:X*], and unfold(x).n:Int.

For the second typing, we have again unfold(x):X(X*) with $X(X^{*}) <: [\text{n:Int, inc:}X^{*}]$. We then use a typing rule for field update in the target language. This rule says that if a:A, c:C, and $A <: [\text{v:C,...}]$ then (a.v:=c) : A. In our case, we have unfold(x):X(X*), b:Int, and $X(X^{*}) <: [\text{n:Int, inc:}X^{*}]$. We obtain (unfold(x).n:=b) : X(X*). Finally, by folding, we obtain fold(unfold(x).n:=b) : X*.

APPLICATIONS

- A language based on matching should be given a set of type rules based on the source type system.
- The rules can be proven sound by a judgment-preserving translation into an object-calculus with higher-order subtyping.

The Language O–3

Syntax of O–3

A,B ::=	types
X	type variable
Top	maximum type
Object(X)[l_iᵥ$_i$:B_i{X} $^{i\in 1..n}$]	object type
Class(A)	class type
All(X<:#A)B	match-quantified type

a,b,c ::=	terms
x	variable
object(x:X=A) l_i=b_i{X,x} $^{i\in 1..n}$ **end**	direct object construction
$a.l$	field/method selection
$a.l$:= **method**(x:X<:#A) b **end**	update
new c	object construction from a class
root	root class
subclass of c:C **with**(x:X<:#A)	subclass
l_i=b_i{X,x} $^{i\in n+1..n+m}$	additional attributes
override l_i=b_i{X,x} $^{i\in Ovr\subseteq 1..n}$ **end**	overridden attributes
c^l(A,a)	class selection
fun(X<:#A) b **end**	match-polymorphic abstraction
b(A)	match-polymorphic instantiation

- Convenient abbreviations:

Root ≜
  Class(**Object**(X)[])

class with(x:X<:#A) l_i=b_i{X,x} $^{i\in 1..n}$ **end** ≜
  subclass of root:**Root with**(x:X<:#A) l_i=b_i{X,x} $^{i\in 1..n}$ **override end**

subclass of c:C **with**(x:X<:#A) ... **super**.l ... **end** ≜
  subclass of c:C **with**(x:X<:#A) ... c^l(X,x) ... **end**

object(x:X=A) ... l **copied from** c ... **end** ≜
  object(x:X=A) ... l=c^l(X,x) ... **end**

$a.l$:= b ≜ where $X,x\notin$FV(b) and a:A,
  $a.l$:= **method**(x:X<:#A) b **end** with A clear from context

Judgments

$E \vdash \diamond$	environment E is well formed
$E \vdash A$	A is well formed type in E
$E \vdash A :: Obj$	A is a well formed object in E
$E \vdash A <: B$	A is a subtype of B in E
$E \vdash A <\!\!\# B$	A matches B in E
$E \vdash a : A$	a has type A in E

Environments

$$\frac{}{\emptyset \vdash \diamond}\ (\text{Env } \emptyset) \qquad \frac{E \vdash A \quad X\notin dom(E)}{E, X<:A \vdash \diamond}\ (\text{Env } X<:) \qquad \frac{E \vdash A :: Obj \quad X\notin dom(E)}{E, X<\!\!\#A \vdash \diamond}\ (\text{Env } X<\!\!\#) \qquad \frac{E \vdash A \quad x\notin dom(E)}{E, x:A \vdash \diamond}\ (\text{Env } x)$$

Types

(Type Obj) $\dfrac{E \vdash A :: Obj}{E \vdash A}$ (Type X) $\dfrac{E', X{<:}A, E'' \vdash \diamond}{E', X{<:}A, E'' \vdash X}$ (Type Top) $\dfrac{E \vdash \diamond}{E \vdash \mathbf{Top}}$

(Type Class) (where $A \equiv \mathbf{Object}(X)[l_i \upsilon_i{:}B_i\ ^{i \in 1..n}]$) $\dfrac{E, X{<:}\#A \vdash B_i \quad \forall i \in 1..n}{E \vdash \mathbf{Class}(A)}$

(Type All<#) $\dfrac{E, X{<:}\#A \vdash B}{E \vdash \mathbf{All}(X{<:}\#A)B}$

Object Types

(Obj X) $\dfrac{E', X{<:}\#A, E'' \vdash \diamond}{E', X{<:}\#A, E'' \vdash X :: Obj}$ (Obj Object) (l_i distinct, $\upsilon_i \in \{^o, ^-, ^+\}$) $\dfrac{E, X{<:}\mathbf{Top} \vdash B_i \quad \forall i \in 1..n}{E \vdash \mathbf{Object}(X)[l_i \upsilon_i{:}B_i\ ^{i \in 1..n}] :: Obj}$

The judgments for types and object types are connected by the (Type Obj) rule.

Subtyping

(Sub Refl) $\dfrac{E \vdash A}{E \vdash A{<:}A}$ (Sub Trans) $\dfrac{E \vdash A{<:}B \quad E \vdash B{<:}C}{E \vdash A{<:}C}$ (Sub X) $\dfrac{E', X{<:}A, E'' \vdash \diamond}{E', X{<:}A, E'' \vdash X{<:}A}$ (Sub Top) $\dfrac{E \vdash A}{E \vdash A{<:}\mathbf{Top}}$

(Sub Object)

$$\frac{E \vdash \mathbf{Object}(X)[l_i \upsilon_i{:}B_i\ ^{i \in 1..n+m}] \quad E \vdash \mathbf{Object}(Y)[l_i \upsilon_i'{:}B_i'\ ^{i \in 1..n}] \quad E, Y{<:}\mathbf{Top}, X{<:}Y \vdash \upsilon_i B_i <: \upsilon_i' B_i' \quad \forall i \in 1..n \quad E, X{<:}\mathbf{Top} \vdash B_i \quad \forall i \in n+1..m}{E \vdash \mathbf{Object}(X)[l_i \upsilon_i{:}B_i\ ^{i \in 1..n+m}] <: \mathbf{Object}(Y)[l_i \upsilon_i'{:}B_i'\ ^{i \in 1..n}]}$$

(Sub All<#) $\dfrac{E \vdash A' <\# A \quad E, X{<:}\#A' \vdash B <: B'}{E \vdash \mathbf{All}(X{<:}\#A)B <: \mathbf{All}(X{<:}\#A')B'}$

(Sub Invariant) $\dfrac{E \vdash B}{E \vdash {}^o B <: {}^o B}$ (Sub Covariant) $\dfrac{E \vdash B <: B' \quad \upsilon \in \{^o, ^+\}}{E \vdash \upsilon B <: {}^+ B'}$ (Sub Contravariant) $\dfrac{E \vdash B' <: B \quad \upsilon \in \{^o, ^-\}}{E \vdash \upsilon B <: {}^- B'}$

Matching

(Match Refl) $\dfrac{E \vdash A :: Obj}{E \vdash A{<:}\#A}$ (Match Trans) $\dfrac{E \vdash A{<:}\#B \quad E \vdash B{<:}\#C}{E \vdash A{<:}\#C}$ (Match X) $\dfrac{E', X{<:}\#A, E'' \vdash \diamond}{E', X{<:}\#A, E'' \vdash X{<:}\#A}$

(Match Object) (l_i distinct)

$$\frac{E, X{<:}\mathbf{Top} \vdash \upsilon_i B_i <: \upsilon_i' B_i' \quad \forall i \in 1..n \quad E, X{<:}\mathbf{Top} \vdash B_i \quad \forall i \in n+1..m}{E \vdash \mathbf{Object}(X)[l_i \upsilon_i{:}B_i\ ^{i \in 1..n+m}] <\# \mathbf{Object}(X)[l_i \upsilon_i'{:}B_i'\ ^{i \in 1..n}]}$$

Terms

(Val Subsumption) $\dfrac{E \vdash a : A \quad E \vdash A <: B}{E \vdash a : B}$ (Val x) $\dfrac{E', x{:}A, E'' \vdash \diamond}{E', x{:}A, E'' \vdash x : A}$

(Val Object) (where $A \equiv \mathbf{Object}(X)[l_i \upsilon_i{:}B_i\{X\}\ ^{i \in 1..n}]$) $\dfrac{E, x{:}A \vdash b_i[\![A]\!] : B_i[\![A]\!] \quad \forall i \in 1..n}{E \vdash \mathbf{object}(x{:}X{=}A)\ l_i{=}b_i\{X\}\ ^{i \in 1..n}\ \mathbf{end} : A}$

(Val Select) (where $A \equiv \mathbf{Object}(X)[l_i \upsilon_i{:}B_i\{X\}\ ^{i \in 1..n}]$) $\dfrac{E \vdash a : A' \quad E \vdash A' <\# A \quad \upsilon_j \in \{^o, ^+\} \quad j \in 1..n}{E \vdash a.l_j : B_j[\![A']\!]}$

(Val Method Update) (where $A \equiv \mathbf{Object}(X)[l_i \upsilon_i{:}B_i\ ^{i \in 1..n}]$) $\dfrac{E \vdash a : A' \quad E \vdash A' <\# A \quad E, X{<:}\#A', x{:}X \vdash b : B_j \quad \upsilon_j \in \{^o, ^-\} \quad j \in 1..n}{E \vdash a.l_j := \mathbf{method}(x{:}X{<:}\#A')b\ \mathbf{end} : A'}$

(Val New) $\dfrac{E \vdash c : \mathbf{Class}(A)}{E \vdash \mathbf{new}\ c : A}$

(Val Root)

$$\frac{E \vdash \diamond}{E \vdash \textbf{root} : \textbf{Class}(\textbf{Object}(X)[])}$$

(Val Subclass) (where $A \equiv \textbf{Object}(X)[l_i \upsilon_i : B_i \ ^{i \in 1..n+m}]$, $A' \equiv \textbf{Object}(X')[l_i \upsilon_i' : B_i' \ ^{i \in 1..n}]$, $Ovr \subseteq 1..n$)

$$\frac{\begin{array}{c} E \vdash \textbf{Class}(A) \quad E \vdash c' : \textbf{Class}(A') \quad E \vdash A <\# A' \\ E, X <\# A \vdash B_i' <: B_i \quad \forall i \in 1..n-Ovr \\ E, X <\# A, x : X \vdash b_i : B_i \quad \forall i \in Ovr \cup n+1..n+m \end{array}}{\begin{array}{c} E \vdash \textbf{subclass of } c' : \textbf{Class}(A') \textbf{ with}(x : X <\# A)\ l_i = b_i \ ^{i \in n+1..n+m} \textbf{ override } l_i = b_i \ ^{i \in Ovr} \textbf{ end} \\ : \textbf{Class}(A) \end{array}}$$

(Val Class Select) (where $A \equiv \textbf{Object}(X)[l_i \upsilon_i : B_i\{X\} \ ^{i \in 1..n}]$)

$$\frac{E \vdash a : A' \quad E \vdash A' <\# A \quad E \vdash c : \textbf{Class}(A) \quad j \in 1..n}{E \vdash c \hat{} l_j(A', a) : B_j[\![A']\!]}$$

(Val Fun<#)

$$\frac{E, X <\# A \vdash b : B}{E \vdash \textbf{fun}(X <\# A)\ b \textbf{ end} : \textbf{All}(X <\# A)B}$$

(Val Appl<#)

$$\frac{E \vdash b : \textbf{All}(X <\# A)B\{X\} \quad E \vdash A' <\# A}{E \vdash b(A') : B[\![A']\!]}$$

CONCLUSIONS

- There are situations in programming where one would like to parameterize over all "extensions" of a recursive object type, rather than over all its subtypes.
- Both F-bounded subtyping and higher-order subtyping can be used in explaining the matching relation.

  We have presented two interpretations of matching:

  $A <\# B \quad \approx \quad A <: B_{Op}(A)$ (F-bounded interpretation)

  $A <\# B \quad \approx \quad A_{Op} \prec: B_{Op}$ (higher-order interpretation)

- Both interpretations can be soundly adopted, but they require different assumptions and yield different rules. The higher-order interpretation validates reflexivity and transitivity.

  Technically, the higher-order interpretation need not assume the equality of recursive types up to unfolding (which seems to be necessary for the F-bounded interpretation). This leads to a simpler underlying theory, especially at higher order.
- Thus, we believe that the higher-order interpretation is preferable; it should be a guiding principle for programming languages that attempt to capture the notion of type extension.

- Matching achieves "covariant subtyping" for Self types and inheritance of binary methods at the cost not validating subsumption.
- Subtyping is still useful when subsumption is needed. Moreover, matching is best understood as higher-order subtyping. Therefore, subtyping is still needed as a fundamental concept, even though the syntax of a programming language may rely only on matching.

EXTRA

Unsoundness of Naive Object Subtyping with Binary Methods

Max ≜ μ(X)[n:Int, max^+:X→X]
MinMax ≜ μ(Y)[n:Int, max^+:Y→Y, min^+:Y→Y]

Consider:

m : Max ≜ [n = 0, max = ...]
mm : MinMax ≜
[n = 0, min = ... ,
max = ς(s:MinMax) λ(o:MinMax)
if o.min(o).n > s.n then o else s]

Assume MinMax <: Max, then:

mm : Max (by subsumption)
mm.max(m) : Max

But:

mm.max(m) ⇝ if m.min(m).n > mm.n then m else mm ⇝ CRASH!

Unsoundness of Covariant Object Types

With record types, it is unsound to admit covariant subtyping of record components in presence of imperative field update. With object types, the essence of that couterexample can be reproduced even in a purely functional setting.

U ≜ [] The unit object type.
L ≜ [l:U] An object type with just l.
$L <: U$

P ≜ [x:U, f:U]
Q ≜ [x:L, f:U]
Assume $Q <: P$ by an (erroneous) covariant rule for object subtyping

$q : Q$ ≜ [x = [l=[]], f = ς(s:Q) $s.x.l$]
then $q : P$ by subsumption with $Q <: P$
hence $q.x$:=[] : P that is [x = [], f = ς(s:Q) $s.x.l$] : P

But ($q.x$:=[]).f fails!

Unsoundness of Method Extraction

It is unsound to have an operation that extracts a method as a function.

(Val Extract) (where $A \equiv [l_i{:}B_i\ ^{i\in 1..n}]$)

$$\frac{E \vdash a : A \qquad j \in 1..n}{E \vdash a.l_j : A \to B_j}$$

(Eval Extract) (where $A \equiv [l_i{:}B_i\ ^{i\in 1..n}]$, $a \equiv [l_i{=}\varsigma(x_i{:}A')b_i\ ^{i\in 1..n+m}]$)

$$\frac{E \vdash a : A \qquad j \in 1..n}{E \vdash a.l_j \leftrightarrow \lambda(x_j{:}A)b_j : A \to B_j}$$

P ≜ [f:[]]
Q ≜ [f:[], y:[]] $Q <: P$

$p : P$ ≜ [f=[]]
$q : Q$ ≜ [f=ς(s:Q)$s.y$, y=[]]
then $q : P$ by subsumption with $Q <: P$
hence $q.f : P \to []$ that is λ(s:Q)$s.y$: $P \to []$

But $q.f(p)$ fails!

Unsoundness of a Naive Recursive Subtyping Rule

Assume:

$A \equiv \mu(X)X \to Nat <: \mu(X)X \to Int \equiv B$

Let:		Type-erased:
$f : Nat \to Nat$	(*given*)	
$a : A = fold(A, \lambda(x{:}A)\ 3)$		$= \lambda(x)\ 3$
$b : B = fold(B, \lambda(x{:}B)\ -3)$		$= \lambda(x)\ -3$
$c : A = fold(A, \lambda(x{:}A)\ f(unfold(x)(a)))$		$= \lambda(x)\ f(x(a))$

By subsumption:

$c : B$

Hence:

$unfold(c)(b) : Int$ Well-typed! $= c(b)$

But:

$unfold(c)(b) = f(-3)$ Error!