Object-Oriented Features

Luca Cardelli

joint work with Martín Abadi

Digital Equipment Corporation

Systems Research Center

ACM Sobótka ’96

Outline

- Topic: a foundation for object-oriented languages based on object calculi.
  - ~ Interesting object-oriented features.
  - ~ Modeling of those features.
- Plan:

  1) Class-Based Languages

  2) Object-Based Languages

  3) Subtyping -- Advanced Features

  4) A Language with Subtyping

  5) Matching -- Advanced Features

  6) A Language with Matching

LECTURE 1

- Intro.
- Class-Based Languages.

Easy Language Features

The early days

- Integers and floats (occasionally, also booleans and voids).
- Monomorphic arrays (Fortran).
- Monomorphic trees (Lisp).

The days of structured programming

- Product types (records in Pascal, structs in C).
- Union types (variant records in Pascal, unions in C).
- Function/procedure types (often with various restrictions).
- Recursive types (typically via pointers).

End of the easy part

- Languages with rich user-definable types (Pascal, Algol68).

Hard Language Features

Four major innovations

- Objects and Subtyping (Simula 67).
- Abstract types (CLU).
- Polymorphism (ML).
- Modules (Modula 2).

Despite much progress, nobody really knows yet how to combine all these ingredients into coherent language designs.

Confusion

These four innovations are partially overlapping and certainly interact in interesting ways. It is not clear which ones should be taken as more prominent. E.g.:

- Object-oriented languages have tried to incorporate type abstraction, polymorphism, and modularization all at once. As a result, o-o languages are (generally) a mess. Much effort has been dedicated to separating these notions back again.
- Claims have been made (at least initially) that objects can be subsumed by either higher-order functions and polymorphism (ML camp), by data abstraction (CLU camp), or by modularization (ADA camp). But later, subtyping features were adopted: ML => ML2000, CLU =>Theta, ADA => ADA'95.
- One hard fact is that full-blown polymorphism can subsume data abstraction. But this kind of polymorphism is more general than, e.g., ML's, and it is not yet clear how to handle it in practice.
- Modules can be used to obtain some form of polymorphism and data abstraction (ADA generics, C++ templates) (Modula 2 opaque types), but not in full generality.

O-O Programming

- Goals
  - ~ Data (state) abstraction.
  - ~ Polymorphism.
  - ~ Code reuse.

- Mechanisms
  - ~ Objects with *self* (packages of data and code).
  - ~ Subtyping and subsumption.
  - ~ Classes and inheritance.

Objects

- Objects and object types
- Objects are packages of data (*instance variables*) and code (*methods*).
- Object types describe the shape of objects.

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


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

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

Classes

- Classes are ways of describing and generating collections of objects of some type.

```
class cell for CellType is
    var contents: Integer := 0;
    method get(): Integer is return self.contents end;
    method set(n: Integer) is self.contents := n end;
end;
var myCell: CellType := new cell;
procedure double(aCell: CellType) is
    aCell.set(2 * aCell.get());
end;
```

Subclasses

- Subclasses are ways of describing classes incrementally, reusing code.

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

subclass reCell of cell for ReCellType is                    (Inherited:
    var backup: Integer := 0;                                  var contents
    override set(n: Integer) is                                method get)
        self.backup := self.contents;
        super.set(n);
    end;
    method restore() is 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.: $ReCellType \quad <: \quad CellType$)

- Subsumption rule

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

  (e.g.: $myReCell : CellType$)

- Subclass rule

  c can be a subclass of d only if $cType <: dType$

  (e.g.: *reCell* can indeed be declared as a subclass of *cell*)

Class-Based Languages

- The mainstream.
- We review only common, kernel properties.

Classes and Objects

- Classes are descriptions of objects. No clear distinction between classes and object types.

- Example: storage cells.

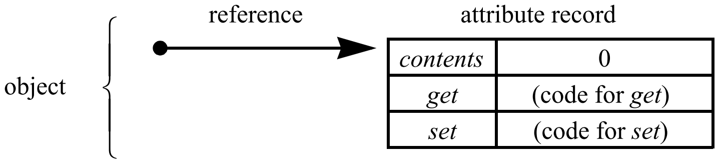
```
class cell is
    var contents: Integer := 0;
    method get(): Integer is
        return self.contents;
    end;
    method set(n: Integer) is
        self.contents := n;
    end;
end;
```

- Classes generate objects.

- Objects can refer to themselves.

Naive Storage Model

- Object = reference to a record of attributes.



Object Operations

- Object creation.
  - ~ *InstanceTypeOf(c)* indicates the type of an object of class *c*.

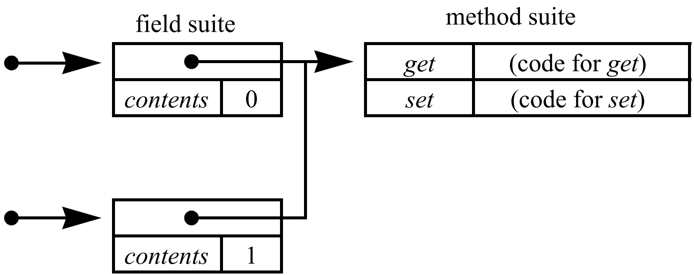
```
var myCell: InstanceTypeOf(cell) := new cell;
```

- Field selection.
- Field update.
- Method invocation.

```
procedure double(aCell: InstanceTypeOf(cell)) is
    aCell.set(2 * aCell.get());
end;
```

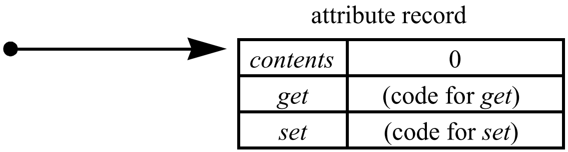
The Method-Suites Storage Model

- A more refined storage model for class-based languages.

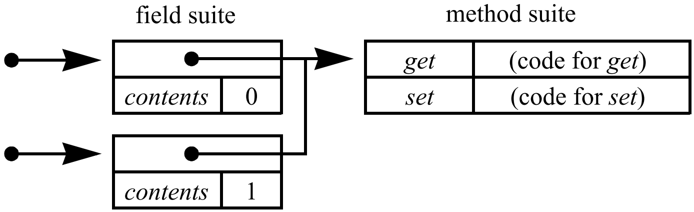


Embedding vs. Delegation

- In the naive storage model, methods are ***embedded*** in objects.



- In the methods-suites storage model, methods are ***delegated*** to the method suites.



- Naive and method-suites models are semantically equivalent for class-based languages.
- They are not equivalent (as we shall see) in object-based languages, where the difference between embedding and delegation is critical.

Method Lookup

- Method lookup is the process of finding the code to run on a method invocation *o*.*m*(…). The details depend on the language and the storage model.
- In class-based languages, method lookup gives the *illusion* that methods are embedded in objects (cf. *o*.*x*, *o*.*m*(...)), hiding storage model details.
- Self is always the *receiver*: the object that *appears* to contain the method.
- Features that would distinguish embedding from delegation implementations (e.g., method update) are usually avoided.

Subclasses and Inheritance

- A *subclass* is a differential description of a class.
- The *subclass relation* is the partial order induced by the subclass declarations.
- Example: restorable cells.

```
subclass reCell of cell is
    var backup: Integer := 0;
    override set(n: Integer) is
        self.backup := self.contents;
        super.set(n);
    end;
    method restore() is
        self.contents := self.backup;
    end;
end;
```

Subclasses and Self

- Because of subclasses, the meaning of **self** becomes dynamic.

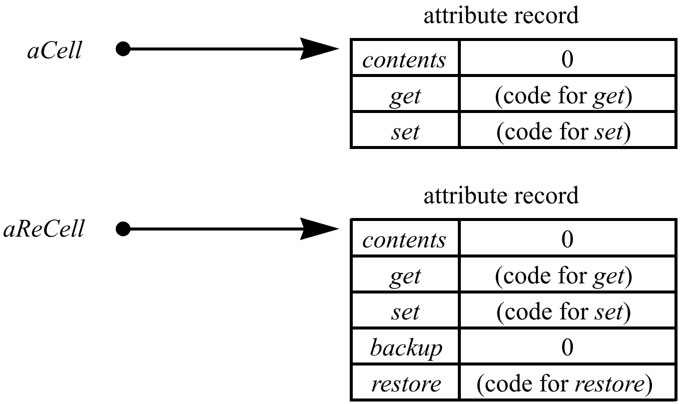
  self.*m*(...)

- Because of subclasses, the concept of **super** becomes useful.

  super.*m*(...)

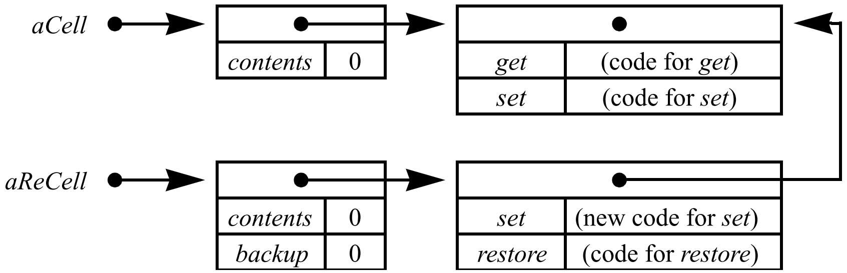
Subclasses and Naive Storage

- In the naive implementation, the existence of subclasses does not cause any change in the storage model.

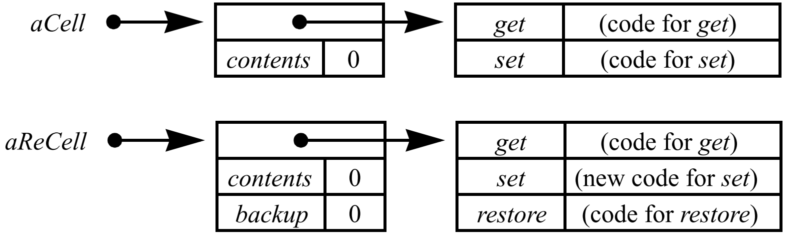


Subclasses and Method Suites

- Because of subclasses, the method-suites model has to be reconsidered. In dynamically-typed class-based languages, method suites are chained:



- In statically-typed class-based languages, however, the method-suites model can be maintained in its original form.



Embedding/Delegation View of Class Hierarchies

- Hierarchical method suites: *delegation* (of objects to suites) combined with *delegation* (of sub-suites to super-suites).
- Collapsed method suites: *delegation* (of objects to suites) combined with *embedding* (of super-suites in sub-suites).

Class-Based Summary

- In analyzing the meaning and implementation of class-based languages we end up inventing and analyzing sub-structures of objects and classes.
- These substructures are independently interesting: they have their own semantics, and can be combined in useful ways.
- What if these substructures were directly available to programmers?

LECTURE 2

- Object-Based Languages.
- O-O Summary.

Object-Based Languages

- Slow to emerge.
- Simple and flexible.
- Usually untyped.

- Just objects and dynamic dispatch.
- When typed, just object types and subtyping.
- Direct object-to-object inheritance.

An Object, All by Itself

- Classes are replaced by object constructors.
- Object types are immediately useful.

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

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

An Object Generator

- Procedures as object generators.

```
procedure newCell(m: Integer): Cell is
    object cell: Cell is
        var contents: Integer := m;
        method get(): Integer is return self.contents end;
        method set(n: Integer) is self.contents := n end;
    end;
    return cell;
end;

var cellInstance: Cell := newCell(0);
```

- Quite similar to classes!

Decomposing Class-Based Features

- General idea: decompose class-based notions and orthogonally recombine them.

- We have seen how to decompose simple classes into objects and procedures.

- We will now investigate how to decompose inheritance.
  - ~ Object generation by parameterization.
  - ~ Vs. object generation by cloning and mutation.

Prototypes and Clones

- Classes describe objects.
- Prototypes describe objects and *are* objects.
- Regular objects are clones of prototypes.

  var *cellClone*: *Cell* := **clone** *cellInstance*;

- **clone** is a bit like **new**, but operates on objects instead of classes.

Mutation of Clones

- Clones are customized by mutation (e.g., update).
- Field update.

```
cellClone.contents := 3;
```

- Method update.

```
cellClone.get :=
    method (): Integer is
        if self.contents < 0 then return 0 else return self.contents end;
    end;
```

- Self-mutation possible.

Self-Mutation

- Restorable cells with no *backup* field.

```
ObjectType ReCell is
    var contents: Integer;
    method get(): Integer;
    method set(n: Integer);
    method restore();
end;
```

- The *set* method updates the *restore* method!

```
object reCell: ReCell is
    var contents: Integer := 0;
    method get(): Integer is return self.contents end;
    method set(n: Integer) is
        let x = self.get();
        self.restore := method () is self.contents := x end;
        self.contents := n;
    end;
    method restore() is self.contents := 0 end;
end;
```

Forms of Mutation

- Method update is an example of a mutation operation. It is simple and statically typable.
- Forms of mutation include:
  - ~ Direct method update (Beta, NewtonScript, Obliq, Kevo, Garnet).
  - ~ Dynamically removing and adding attributes (Self, Act1).
  - ~ Swapping groups of methods (Self, Ellie).

Object-Based Inheritance

- Object generation can be obtained by procedures, but with no real notion of inheritance.
- Object inheritance can be achieved by cloning (reuse) and update (override), but with no shape change.
- How can one inherit with a change of shape?
- An option is object extension. But:
  - ~ Not easy to typecheck.
  - ~ Not easy to implement efficiently.
  - ~ Provided rarely or restrictively.

Donors and Hosts

- General object-based inheritance: building new objects by “reusing” attributes of existing objects.

- Two orthogonal aspects:

  - obtaining the attributes of a ***donor*** object, and

  - incorporating those attributes into a new ***host*** object.

- Four categories of object-based inheritance:

  - The attributes of a donor may be obtained ***implicitly*** or ***explicitly***.

  - Orthogonally, those attributes may be either ***embedded*** into a host, or ***delegated*** to a donor.

Implicit vs. Explicit Inheritance

- A difference in declaration.
- **Implicit inheritance**: one or more objects are designated as the donors (explicitly!), and their attributes are implicitly inherited.
- **Explicit inheritance**, individual attributes of one or more donors are explicitly designated and inherited.
- **Super** and **override** make sense for implicit inheritance, not for explicit inheritance.

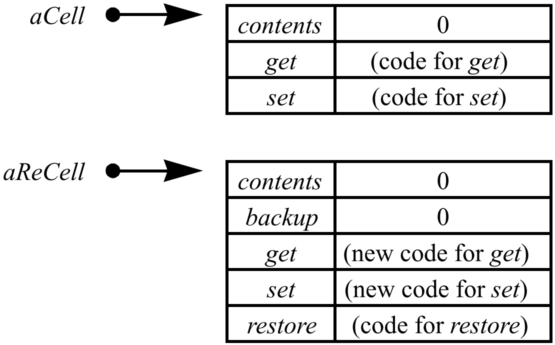
- Intermediate possibility: explicitly designate a named collection of attributes that, however, does not form a whole object. E.g. *mixin* inheritance.
- (We can see implicit and explicit inheritance, as the extreme points of a spectrum.)

Embedding vs. Delegation Inheritance

- A difference in execution.
- **Embedding inheritance**: the attributes inherited from a donor become part of the host (in principle, at least).
- **Delegation inheritance**: the inherited attributes remain part of the donor, and are accessed via an indirection from the host.
- Either way, self is the receiver.
- In embedding, host objects are independent of their donors. In delegation, complex webs of dependencies may be created.

Embedding

- Host objects contain copies of the attributes of donor objects.



contents	0
get	(code for *get*)
set	(code for *set*)

contents	0
backup	0
get	(new code for *get*)
set	(new code for *set*)
restore	(code for *restore*)

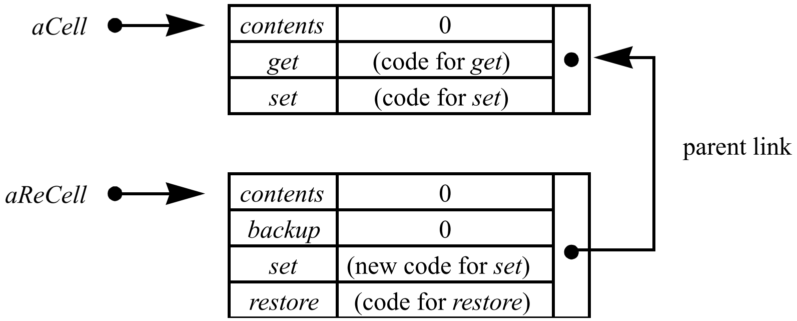
Embedding-Based Languages

- Embedding provides the simplest explanation of the standard semantics of **self** as the receiver.
- Embedding was described by Borning as part of one of the first proposals for prototype-based languages.
- Recently, it has been adopted by languages like Kevo and Obliq. We call these languages *embedding-based* (*concatenation-based*, in Kevo terminology).

Delegation

- Host objects contain *links* to the attributes of donor objects.
- Prototype-based languages that permit the sharing of attributes across objects are called *delegation-based*.
- Operationally, delegation is the redirection of field access and method invocation from an object or prototype to another, <u>in such a way that an object can be seen as an extension of another</u>.
- A crucial aspect of delegation inheritance is the interaction of donor links with the binding of **self**.

Delegation Inheritance



- Note: similar to hierarchical method suites.

Dynamic Inheritance

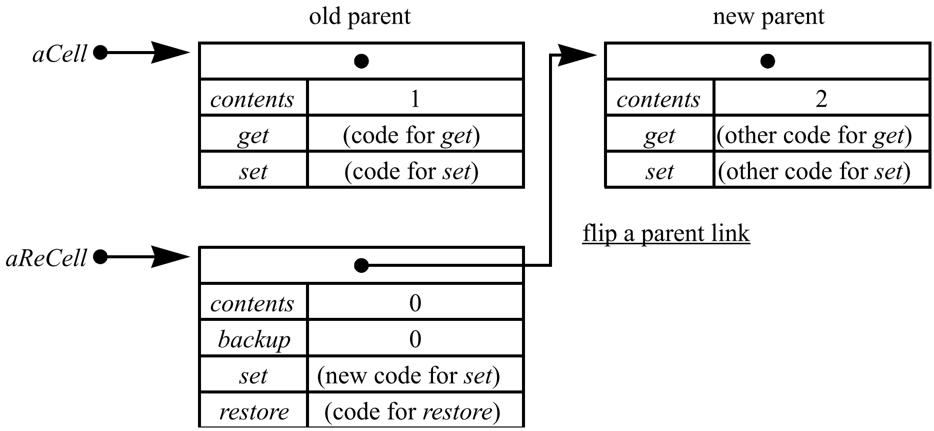
- Inheritance is called *static* when inherited attributes are fixed for all time.
- It is *dynamic* when the collection of inherited attributes can be updated dynamically (replaced, increased, decreased).

Mode Switching

- Although dynamic inheritance is in general a dangerous feature, it enables rather elegant and disciplined programming techniques.

- In particular, *mode-switching* is the special case of dynamic inheritance where a collection of (inherited) attributes is *swapped* with a similar collection of attributes. (This is even typable.)

Delegation-Style Mode Switching



Embedding-Style Mode Switching

<u>flip a set of attributes</u>

old object

contents	0
backup	0
get	(code for *get*)
set	(code for *set*)
restore	(code for *restore*)

new object

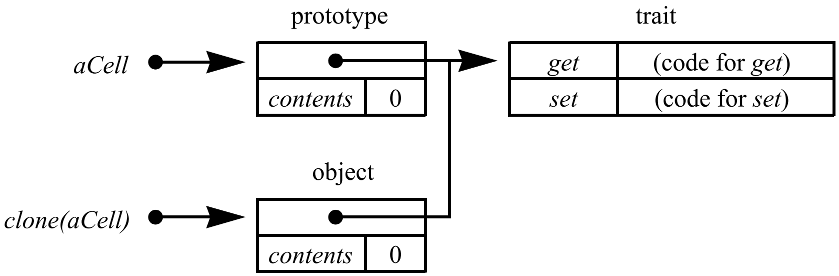
contents	0
backup	0
get	(other code for *get*)
set	(other code for *set*)
restore	(code for *restore*)

Traits: from Prototypes back to Classes?

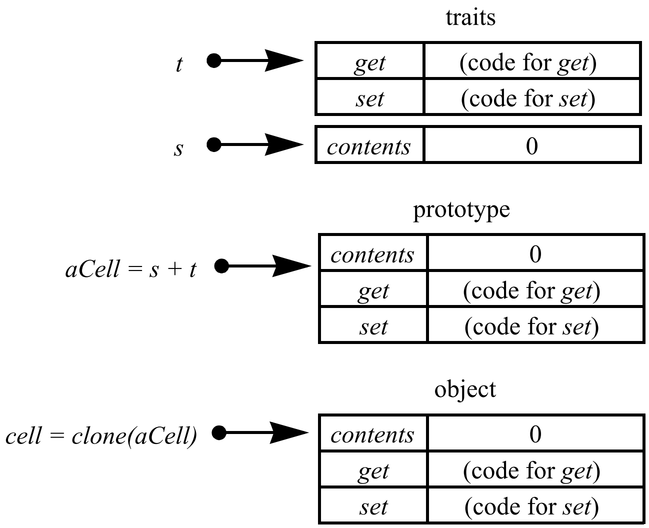
- Prototypes were initially intended to replace classes.
- Several prototype-based languages, however, seem to be moving towards a more traditional approach based on class-like structures.
- Prototypes-based languages like Omega, Self, and Cecil have evolved usage-based distinctions between objects.

Different Kinds of Objects

- Trait objects.
- Prototype objects.
- Normal objects.



Embedding-Style Traits



Traits are not Prototypes

- This separation of roles violates the original spirit of prototype-based languages: traits objects cannot function on their own. They typically lack instance variables.
- With the separation between traits and other objects, we seem to have come full circle back to class-based languages and to the separation between classes and instances.
- Trait-based techniques looks exactly like implementation techniques for classes.

Contributions of the Object-Based Approach

- The achievement of object-based languages is to make clear that classes are just one of the possible ways of generating objects with common properties.
- Objects are more primitive than classes, and they should be understood and explained before classes.
- Different class-like constructions can be used for different purposes; hopefully, more flexibly than in strict class-based languages.

SUMMARY

- Class-based: various implementation techniques based on embedding and/or delegation. Self is the receiver.
- Object-based: various language mechanisms based on embedding and/or delegation. Self is the receiver.
- Object-based can emulate class-based. (By traits, or by otherwise reproducing the implementations techniques of class-based languages.)

One Step Further

- Language analysis:
  - ~ Class-based langs. → Object-based langs. → Object calculi
- Language synthesis:
  - ~ Object calculi → Object-based langs. → Class-based langs.

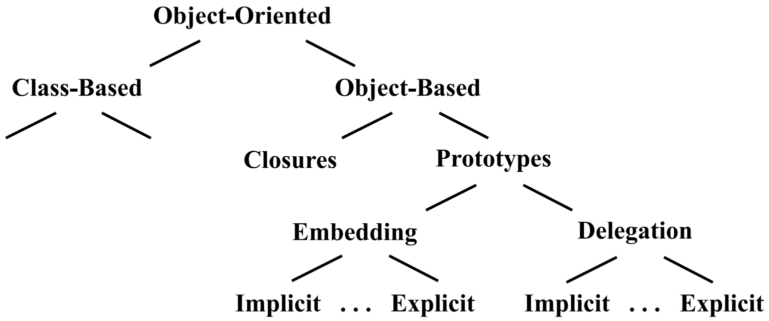
Our Approach to Modeling

- We have identified embedding and delegation as underlying many object-oriented features.
- In our object calculi, we choose embedding over delegation as the principal object-oriented paradigm.
- The resulting calculi can model classes well, although they are not class-based (since classes are not built-in).
- They can model delegation-style traits just as well, but not “true” delegation. (Object calculi for delegation exist but are more complex.)

Foundations

- Objects can emulate classes (by traits) and procedures (by “stack frame objects”).
- *Everything* can indeed be an object.

A Taxonomy


Object-Oriented
Class-Based
Object-Based
Closures
Prototypes
Embedding
Delegation
Implicit
. . .
Explicit
Implicit
. . .
Explicit

LECTURE 3

- Various advanced subtyping issues.
  - Subsumption andDynamic Dispatch.
  - Type Information Lost and Found.
  - Covariance, Contravariance, Invariance.
  - Method Specialization.
  - Self Type Specialization.
  - Type Parameters.
  - Extra: Inheritance, Subclasses, Subtypes and Object Types.
  - Extra: Distinguishing Subtyping from Subclassing.
- Subclassing without Subtying and Object Protocols. *(Skip if Lecture 5 is given.)*

Reminder: Classes

- Classes are descriptions of objects. No clear distinction between classes and object types.
- Example: storage cells.

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

Reminder: Subclasses

- A *subclass* is a differential description of a class.
- Example: restorable cells.

```
subclass reCell of cell is
    var backup: Integer := 0;
    override set(n: Integer) is
        self.backup := self.contents;
        super.set(n);
    end;
    method restore() is
        self.contents := self.backup;
    end;
end;
```

Subsumption and Dynamic Dispatch

- It may seem that subclasses are just a convenient mechanism to avoid rewriting definitions that already appear in superclasses. Much more comes into play with the associated notion of subsumption. Consider the definitions:

  var *myCell*: *InstanceTypeOf*(*cell*) := **new** *cell*;
  var *myReCell*: *InstanceTypeOf*(*reCell*) := **new** *reCell*;
  procedure *f*(*x*: *InstanceTypeOf*(*cell*)) **is** … **end**;

- Consider also the following code fragments, in the scope of those definitions:

  myCell := *myReCell*;

  f(*myReCell*);

- Here, an instance of class *reCell* is assigned to a variable holding instances of class *cell*. Similarly, an instance of class *reCell* is passed to a procedure *f* that expects instances of class *cell*. Both code fragments would be illegal in a language like Pascal, since the types *InstanceTypeOf*(*cell*) and *InstanceTypeOf*(*reCell*) do not match.

Polymorphism

- In object-oriented languages these code fragments are made legal by the following rule, which embodies what is often called (*subtype*) *polymorphism*:

  If *c'* is a subclass of *c*, and *o'* is an instance of *c'*, then *o'* is an instance of *c*.

  or, from the point of view of the typechecker:

  (1) If *c'* is a subclass of *c*, and *o'*: *InstanceTypeOf*(*c'*), then *o'*: *InstanceTypeOf*(*c*).

- We analyze statement (1) further, by introducing a reflexive and transitive subtype relation (<:) between *InstanceTypeOf* types. This subtype relation is intended, intuitively, as set inclusion between sets of values. For now we do not define the subtype relation precisely, but we assume that it satisfies two properties:

  (2) If $a : A$, and $A <: B$, then $a : B$.
  (3) *InstanceTypeOf*(*c'*) <: *InstanceTypeOf*(*c*) if and only if *c'* is a subclass of *c*.

  Together, (2) and (3) yield (1).

- Property (2), called *subsumption*, is the characteristic property of subtype relations. By subsumption, a value of type A can be viewed as a value of a supertype B. We say that the value is subsumed from type A to type B.
- Property (3), which we may call subclassing-is-subtyping, is the characteristic property of subclassing in classical class-based languages. Since inheritance is connected with subclassing, we may read (3) as an inheritance-is-subtyping property. More recent class-based languages adopt a different, inheritance-is-*not*-subtyping approach, as we shall see later.

Static Dispatch vs. Dynamic Dispatch

- With the introduction of subsumption, we have to reexamine the meaning of method invocation. For example, given the code:

  ```
  procedure g(x: InstanceTypeOf(cell)) is
      x.set(3);
  end;
  g(myReCell);
  ```

  we should determine what is the meaning of *x.set*(3) during the invocation of *g*. The declared type of *x* is *InstanceTypeOf*(*cell*), while its value is *myReCell*, which is an instance of *reCell*. Since the method *set* is overridden in *reCell*, there are two possibilities:

Static dispatch:	*x.set*(3)	executes the code of *set* from class *cell*
Dynamic dispatch:	*x.set*(3)	executes the code of *set* from class *reCell*

- Static dispatch is based on the compile-time type information available for *x*. Dynamic dispatch is based on the run-time value of *x*; we may say that *InstanceTypeOf*(*reCell*) is the *true type* of *x* during the execution of *g*(*myReCell*), and that the true type determines the choice of method.

- Dynamic dispatch is found in all object-oriented languages, to the point that it can be regarded as one of their defining properties.
- Dynamic dispatch is an important component of object abstraction: each object knows how to behave autonomously, so the context does not need to examine the object and decide which operation to apply.
- An interesting consequence of dynamic dispatch is that subsumption should have no run-time effect on objects.
- For example, if an application of subsumption from *InstanceTypeOf*(*reCell*) to *InstanceTypeOf*(*cell*) were to "coerce" a *reCell* to a *cell* by cutting off its additional attributes (*backup* and *restore*), then a dynamically dispatched invocation of *set* would fail. The fact that subsumption has no run-time effect is both good for efficiency and semantically necessary.

Type Information, Lost and Found

- Although subsumption has no run-time effect, it has the consequence of reducing static knowledge about the true type of an object.
- Imagine a root class with no attributes, such that all classes are subclasses of the root class. Then any object can be considered, by subsumption, as a member of the root class and can be regarded as a useless object with no attributes.
- Less drastically, when subsuming an object from *InstanceTypeOf*(*reCell*) to *InstanceTypeOf*(*cell*), the ability to access the field *backup* (as well as the method *restore*) is lost. This fact, however, does not make the field *backup* redundant because it is still used, through self, by the body of the overriding method *set*.
- So, attributes forgotten by subsumption can still be used thanks to dynamic dispatch.

- In a purist view of object-oriented methodology, dynamic dispatch is the only mechanism for taking advantage of attributes that have been forgotten by subsumption.
- This position is often taken on abstraction grounds: no knowledge should be obtainable about objects except by invoking their methods. In the purist approach, subsumption provides a simple and effective mechanism for hiding private attributes. If we create a *reCell* and, by subsumption, give it to a client as a *cell*, we can be sure that the client cannot directly affect the *backup* field.
- Most languages, however, provide some way of inspecting the type of an object and, consequently, of regaining access to its forgotten attributes.
- For example, a procedure with parameter *x* of type *InstanceTypeOf*(*cell*) could contain the following code. The **typecase** statement binds *x* to *c* or to *rc* depending on the true (run-time) type of *x*:

```
typecase x
    when rc: InstanceTypeOf(reCell) do … rc.restore() … ;
    when c: InstanceTypeOf(cell) do … c.set(3) … ;
end;
```

Previously inaccessible attributes can now be used in the *rc* branch.

- The **typecase** mechanism is useful, but it is considered impure for several methodological reasons (and also for theoretical ones).
  - ~ It violates the object abstraction, revealing information that may be regarded as private.
  - ~ It renders programs more fragile by introducing a form of dynamic failure when none of the branches apply.
  - ~ It makes code less extensible: when adding another subclass of *cell* one may have to revisit and extend the **typecase** statements in existing code. In the purist framework, the addition of a new subclass does not require recoding of existing classes. This is a good property, in particular because the source code of commercial libraries may not be available.

- Although **typecase** may be ultimately an unavoidable feature, its methodological drawbacks require that it be used prudently.
- The desire to reduce the uses of **typecase** has shaped much of the type structure of object-oriented languages.
  - ~ In particular, **typecase** on self is necessary for emulating objects in conventional languages by records of procedures; in contrast, the standard typing of methods in object-oriented languages avoids this need for **typecase**.
  - ~ More sophisticated typings of methods are aimed at avoiding **typecase** also on method results (using Self types) and on method arguments (using matching).

A Simple Formalization of Typecase

- The *typecase* construct evaluates a term to a result, and discriminates on the type of the result.

  $typecase\ a \mid (x{:}A)d_1 \mid d_2$

  If a yields a result of type A, then ($typecase\ a|(x{:}A)d_1|d_2$) returns d_1 with x replaced by this result. If a yields a result that does not have type A, then ($typecase\ a|(x{:}A)d_1|d_2$) returns d_2.

Typing rule for *typecase*

(Val Typecase)

$$\frac{E \vdash a : A' \qquad E, x{:}A \vdash d_1 : D \qquad E \vdash d_2 : D}{E \vdash typecase\ a|(x{:}A)d_1|d_2 : D}$$

- In programming languages that include *typecase*, the type of a value is represented using a tag attached to the value; *typecase* relies on this tag to perform run-time type discrimination.
- We adopt an operational semantics without such tags; in our semantics, *typecase* performs run-time type discrimination by constructing a typing derivation.
- Our rules do not immediately suggest an efficient implementation, but have the advantage of being simple and general, applying equally well to calculi with and without subtyping.

Operational semantics for *typecase*

(Red Typecase Match)

$$\frac{\vdash a \rightsquigarrow v' \qquad \emptyset \vdash v' : A \qquad \vdash d_1\{\!\{v'\}\!\} \rightsquigarrow v}{\vdash \mathit{typecase}\ a|(x{:}A)d_1\{x\}|d_2 \rightsquigarrow v}$$

(Red Typecase Else)

$$\frac{\vdash a \rightsquigarrow v' \qquad \emptyset \nvdash v' : A \qquad \vdash d_2 \rightsquigarrow v}{\vdash \mathit{typecase}\ a|(x{:}A)d_1|d_2 \rightsquigarrow v}$$

- These rules are unusual in that the result of a reduction depends on whether a typing is derivable. If $\emptyset \vdash v' : A$ is derivable, then the first rule is applicable. Otherwise (when $\emptyset \nvdash v' : A$), the second rule is applicable; thus the second rule has a negative assumption.

Covariance, Contravariance, and Invariance

- The type $A \times B$ is the type of pairs with left component of type A and right component of type B. The operations $fst(c)$ and $snd(c)$ extract the left and right components, respectively, of an element c of type $A \times B$.
- We say that $\times$ is a *covariant* operator (in both arguments), because $A \times B$ varies in the same sense as A or B:

  $A \times B <: A' \times B'$ provided that $A <: A'$ and $B <: B'$

  We can justify this property as follows:

Argument for the covariance of $A \times B$

A pair $\langle a,b \rangle$ with left component a of type A and right component b of type B, has type $A \times B$. If $A <: A'$ and $B <: B'$, then by subsumption we have $a:A'$ and $b:B'$, so that $\langle a,b \rangle$ has also type $A' \times B'$. Therefore, any pair of type $A \times B$ has also type $A' \times B'$ whenever $A <: A'$ and $B <: B'$. In other words, the inclusion $A \times B <: A' \times B'$ between product types is valid whenever $A <: A'$ and $B <: B'$.

- The type $A \rightarrow B$ is the type of functions with argument type A and result type B.
- We say that $\rightarrow$ is a *contravariant* operator in its left argument, because $A \rightarrow B$ varies in the opposite sense as A; the right argument is instead covariant:

$$A \rightarrow B <: A' \rightarrow B' \quad \text{provided that} \quad A' <: A \text{ and } B <: B'$$

Argument for the co/contravariance of $A \rightarrow B$

If $B<:B'$, then a function f of type $A \rightarrow B$ produces results of type B' by subsumption. If $A'<:A$, then f accepts also arguments of type A', since these have type A by subsumption. Therefore, every function of type $A \rightarrow B$ has also type $A' \rightarrow B'$ whenever $A'<:A$ and $B<:B'$. In other words, the inclusion $A \rightarrow B <: A' \rightarrow B'$ between function types is valid whenever $A'<:A$ and $B<:B'$.

- In the case of functions of multiple arguments, for example of type $(A_1 \times A_2) \rightarrow B$, we have contravariance in both A_1 and A_2. This is because product, which is covariant in both of its arguments, is found in a contravariant context.

- Consider pairs whose components can be updated; we indicate their type by $A \ast B$. Given $p{:}A \ast B$, $a{:}A$, and $b{:}B$, we have operations $\mathit{getLft}(p){:}A$ and $\mathit{getRht}(p){:}B$ that extract components, and operations $\mathit{setLft}(p,a)$ and $\mathit{setRht}(p,b)$ that destructively update components.
- The operator $\ast$ does not enjoy any covariance or contravariance properties:

$$A \ast B <: A' \ast B' \quad \text{provided that} \quad A=A' \text{ and } B=B'$$

We say that $\ast$ is an *invariant* operator (in both of its arguments).

Argument for the invariance of $A \ast B$

If $A<:A'$ and $B<:B'$, can we covariantly allow $A \ast B <: A' \ast B'$? If we adopt this inclusion, then from $p{:}A \ast B$ we obtain $p{:}A' \ast B'$, and we can perform $\mathit{setLft}(p,a')$, for any $a'{:}A'$. After that, $\mathit{getLft}(p)$ might return an element of type A' that is not an element of type A. Hence, the inclusion $A \ast B <: A' \ast B'$ is not sound.

Conversely, if $A''<:A$ and $B''<:B$, can we contravariantly allow $A \ast B <: A'' \ast B''$? From $p{:}A \ast B$ we now obtain $p{:}A'' \ast B''$, and we can incorrectly deduce that $\mathit{getLft}(p){:}A''$. Hence, the inclusion $A \ast B <: A'' \ast B''$ is not sound either.

- Much controversy in the object-oriented community revolves around the question of whether the types of arguments of methods should vary covariantly or contravariantly from classes to subclasses.
- The properties of $\times$, $\rightarrow$, and ※ follow inevitably from our assumptions. The variance properties of method types follow inevitably by a similar analysis. We cannot take method argument types to vary covariantly, unless somehow we change the meaning of covariance, subtyping, or subsumption.
- According to our definitions, covariance of method argument types is statically unsound: if left unchecked, it may result in unpredictable behavior.
- Eiffel still favors covariance of method arguments. Unsound behavior was supposed to be caught by global flow analysis in Eiffel, but was never implemented.
- Covariance can be soundly adopted for multiple dispatch, but using a different set of type operators.

Method Specialization

- So far we have taken the simplest approach to overriding, requiring that an overriding method has exactly the same type as the overridden method.
- This condition can be relaxed to allow *method specialization*, that is, to allow an overriding method to adopt different argument and result types, specialized for the subclass.
- We still do not allow overriding and specialization of field types: fields are updatable, like the components of the type A※B, and therefore their types must be invariant.

- Suppose we use different argument and result types, *A'* and *B'*, when overriding *m*:

```
class c is
    method m(x: A): B is … end;
    method m_1(x_1: A_1): B_1 is … end;
end;

subclass c' of c is
    override m(x: A'): B' is … end;
end;
```

- In determining the admissible *A'* and *B'*, we are constrained by the possibility of subsumption between *InstanceTypeOf*(*c'*) and *InstanceTypeOf*(*c*).
- When *o'* of type *InstanceTypeOf*(*c'*) is subsumed into *InstanceTypeOf*(*c*), and *o'*.*m*(*a*) is invoked, it is acceptable for the argument to have static type *A* and for the result to have static type *B*. Therefore, it is sufficient to require that $B'<:B$ (covariantly) and that $A<:A'$ (contravariantly).
- This is called *method specialization on override*: the result type *B* is specialized to *B'*, while the parameter type *A* is generalized to *A'*, with the net effect that $A\to B$ is specialized to $A'\to B'$.

- There is another form of method specialization that happens implicitly by inheritance. The occurrences of **self** in the methods of *c* can be considered of type *InstanceTypeOf*(*c*), in the sense that all objects bound to **self** have type *InstanceTypeOf*(*c*) or a subtype of *InstanceTypeOf*(*c*).
- When the methods of *c* are inherited by *c'*, the same occurrences of **self** can similarly be considered of type *InstanceTypeOf*(*c'*). Thus, the type of **self** is silently specialized on inheritance (covariantly!).

- Another way to look at specialization is to consider methods as functions whose first parameter is self; we call such functions *pre-methods*.
- In the case of the method m_1 of c, this pre-method pm_1 would have type $(\mathit{InstanceTypeOf}(c) \times A_1) \rightarrow B_1$. This type is a subtype of $(\mathit{InstanceTypeOf}(c') \times A_1) \rightarrow B_1$, so pm_1 has this type too by subsumption. Then pm_1 has the type of a legal pre-method for c', and it can be inherited by c'.
- More generally, an inheritable pre-method of type $(\mathit{InstanceTypeOf}(c) \times A_1) \rightarrow B_1$ has by subsumption the type $(\mathit{InstanceTypeOf}(c') \times A_1') \rightarrow B_1'$ for any $A_1' <: A_1$ and $B_1 <: B_1'$. That is, the parameters, including self, may be specialized, and the results may be generalized.
- Note that the argument and result inclusions we have derived for inheritance are opposite to the ones for overriding: for inheritance, parameter types may be specialized, and result types may be generalized; for overriding, parameter types may be generalized, and result types may be specialized. In any case, the sound rules for method specialization, both for inheritance and for overriding, are a direct consequence of the constraints on subtyping and subsumption.

Self Type Specialization

- Method specialization adds flexibility to subclass definitions by allowing the result type of a method to vary. Another opportunity for flexibility arises when the result type of a method is, recursively, its class type. We consider a language construct devised for specializing such methods.
- Class definitions are often recursive, in the sense that the definition of a class *c* may contain occurrences of *InstanceTypeOf*(*c*). For example, we could have a class *c* containing a method *m* with result type *InstanceTypeOf*(*c*):

```
class c is
    var x: Integer := 0;
    method m(): InstanceTypeOf(c) is … self … end;
end;
subclass c’ of c is
    var y: Integer := 0;
end;
```

- On inheritance, recursive types are, by default, preserved exactly, just like other types. For instance, for *o'* of class *c'*, we have that *o'*.*m*() has type *InstanceTypeOf*(*c*) and not, for example, *InstanceTypeOf*(*c'*). In general, adopting *InstanceTypeOf*(*c'*) as the result type for the inherited method *m* in *c'* is unsound, because *m* may construct and return an instance of *c* that is not an instance of *c'*.
- Suppose, though, that *m* returns **self**, perhaps after modifying the field *x*. Then it would be sound to give the inherited method the result type *InstanceTypeOf*(*c'*), since **self** is always bound to the receiver of invocations. With this more precise typing, we would avoid subsequent uses of **typecase**: limiting the result type to *InstanceTypeOf*(*c*) constitutes an unwarranted loss of information.

- This argument leads to the notion of *Self types*. The keyword **Self** represents the type of **self**. Instead of assigning the result type *InstanceTypeOf*(*c*) to *m*, we now write:

```
class c is
    var x: Integer := 0;
    method m(): Self is … self … end;
end;
```

- The typing of the code of *m* relies on the assumptions that **Self** is a subtype of *InstanceTypeOf*(*c*), and that **self** has type **Self**. Field updates to **self** preserve its **Self** type. For soundness, the result of *m* must be shown to have type **Self**, and not just type *InstanceTypeOf*(*c*).
- When *c'* is declared as a subclass of *c*, the result type of *m* is still taken to be **Self**. However, **Self** is then regarded as a subtype of *InstanceTypeOf*(*c'*). Thus **Self**, as the result type of a method, is automatically specialized on subclassing.

- There are no drawbacks to extending classical class-based languages with **Self** in covariant positions, for example as the result type of methods. This extension increases expressive power and prevents loss of type information at no cost other than properly keeping track of the type of **self**.
- We can even allow **Self** as the type of fields; this is sound as long as those fields are updated only with **self** or updated versions of **self**.
- A natural next step is to allow **Self** in contravariant (argument) positions. This is what Eiffel set out to do. Unfortunately, contravariant uses of **Self** are unsound for subsumption, as can be demonstrated by simple counterexamples. As we have seen, types in argument positions of inherited methods may be generalized, not specialized.
- The proper handling of **Self** in contravariant positions is a major new development in class-based languages, and goes well beyond their classical features.

Type Parameters

- Type parameterization is a general technique for reusing the same piece of code at different types. It is becoming common in modern object-oriented languages, partially independently of object-oriented features.
- In conjunction with subtyping, type parameterization can be used to remedy some typing difficulties due to contravariance, for example in method specialization.
- Consider the following object types, where *Vegetables* <: *Food* (but not vice versa):

```
ObjectType Person is
    …
    method eat( food: Food);
end;
ObjectType Vegetarian is
    …
    method eat( food: Vegetables);
end;
```

- The intention is that a vegetarian is a person, so we would expect *Vegetarian* <: *Person*.
- However, this inclusion cannot hold because of the contravariance on the argument of the *eat* method. If we erroneously assume *Vegetarian* <: *Person*, then a vegetarian can be subsumed into *Person*, and can be made to eat meat.
- We can obtain some legal subsumptions between vegetarians and persons by converting the corresponding object types into type operators parameterized on the type of *food*:

```
ObjectOperator PersonEating[F <: Food] is
    …
    method eat( food: F);
end;
ObjectOperator VegetarianEating[F <: Vegetables] is
    …
    method eat( food: F);
end;
```

- The mechanism used here is called *bounded type parameterization*. The variable F is a type parameter, which can be instantiated with a type. A bound like $F <: \mathit{Vegetables}$ limits the possible instantiations of F to subtypes of *Vegetables*.
- So, $\mathit{VegetarianEating}[\mathit{Vegetables}]$ is a type; in contrast, $\mathit{VegetarianEating}[\mathit{Food}]$ is not well-formed. The type $\mathit{VegetarianEating}[\mathit{Vegetables}]$ is an instance of *VegetarianEating*, and is equal to the type *Vegetarian*.
- We have that:

$$\text{for all } F <: \mathit{Vegetables},\ \ \mathit{VegetarianEating}[F] <: \mathit{PersonEating}[F]$$

  because, for any $F <: \mathit{Vegetables}$, the two instances are included by the usual rules for subtyping.
- In particular, we obtain:

$$\mathit{Vegetarian} = \mathit{VegetarianEating}[\mathit{Vegetables}] <: \mathit{PersonEating}[\mathit{Vegetables}].$$

  This inclusion can be useful for subsumption: it asserts, correctly, that a vegetarian is a person that eats only vegetables.

- Related to bounded type parameters are *bounded abstract types* (also called *partially abstract types*). Bounded abstract types offer a different solution to the problem of making *Vegetarian* subtype of *Person*.
- We redefine our object types by adding the *F* parameter, subtype of *Food*, as one of the attributes:

```
ObjectType Person is
    type F <: Food;
    …
    var lunch: F;
    method eat( food: F);
end;
ObjectType Vegetarian is
    type F <: Vegetables;
    …
    var lunch: F;
    method eat( food: F);
end;
```

- The meaning of the type component *F*<:*Food* in *Person* is that, given a person, we know that it can eat some *Food*, but we do not know exactly of what kind. The *lunch* attribute provides some food that a person can eat.
- We can build an object of type *Person* by choosing a specific subtype of *Food*, for example *F*=*Dessert*, picking a dessert for the *lunch* field, and implementing a method with parameter of type *Dessert*. We have that the resulting object is a *Person*, by forgetting the specific *F* that we chose for its implementation.
- Now the inclusion *Vegetarian* <: *Person* holds. A vegetarian subsumed into *Person* can be safely fed the lunch it carries with it, because originally the vegetarian was constructed with *F*<:*Vegetables*.
- A limitation of this approach is that a person can be fed only the food it carries with it as a component of type *F*, and not some food obtained independently.

Inheritance, Subclassing, Subtyping

- One of the main characteristics of classical class-based languages is the strict correlation between inheritance, subclassing, and subtyping. A great economy of concepts and syntax is achieved by identifying these three relations.
- The identification also confers much flexibility in the use of subsumption: an object of a subclass, some of whose methods may have been inherited, can always be used in place of an object of a superclass by virtue of subtyping.
- There are situations, however, in which inheritance, subclassing, and subtyping conflict. Opportunities for code reuse, both by inheritance and by parameterization, turn out to be limited by the coincidence of these relations.
- Therefore, considerable attention has been devoted to separating them. The separation of subclassing from subtyping is becoming commonplace; other separations are more tentative.

Object Types

- In the original formulation of classes (in Simula, for example), the type description of objects is intermixed with the implementation of methods.
- This conflicts with the now widely recognized advantages of keeping specifications separate from implementations, particularly to enable separate code development within large teams of programmers.
- A relatively recent variation on the classical model addresses this problem. Separation between object specification and object implementation can be achieved by introducing types for objects that are independent of specific classes. This approach is supported by languages that provide both classes and interfaces.
- In class-based languages, the type of an object is related to its class, as in *InstanceTypeOf*(*cell*). It is peculiar that the type *InstanceTypeOf*(*cell*) should depend explicitly on an entity, the class *cell*, that describes some specific method code. None of the code from class *cell* is necessarily found in members of *InstanceTypeOf*(*cell*).
- Object types, instead, list attributes and their types, but not their implementations. They are suitable to appear in interfaces, and to be implemented separately and in more than one way.

- Recall the classes *cell* and *reCell*:

```
class cell is
        var contents: Integer := 0;
        method get(): Integer is return self.contents end;
        method set(n: Integer) is self.contents := n end;
end;
subclass reCell of cell is
        var backup: Integer := 0;
        override set(n: Integer) is
                self.backup := self.contents;
                super.set(n);
        end;
        method restore() is self.contents := self.backup end;
end;
```

- We introduce two object types *Cell* and *ReCell* that correspond to these classes. We write them as separate types (but we could introduce syntax to avoid repeating common components).

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

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

- We may still use *ObjectTypeOf*(*cell*) as a meta-notation for the object type *Cell*. This type can be mechanically extracted from class *cell*. Therefore, we may write either *o*: *ObjectTypeOf*(*cell*) or *o*: *Cell*.
- The main property we expect of *ObjectTypeOf* is that:

$$\textbf{new}\ c : \mathit{ObjectTypeOf}(c) \qquad\qquad \text{for any class } c$$

- Different classes *cell* and $cell_1$ may happen to produce the same object type *Cell*, equal to both *ObjectTypeOf*(*cell*) and *ObjectTypeOf*($cell_1$). Therefore, objects having type *Cell* are required only to satisfy a certain protocol, independently of attribute implementation.

Distinguishing Subclassing from Subtyping

- In class-based languages, the subtype relation is based on the subclass relation.
- When object types are independent of classes, we must provide an independent definition. There are several choices at this point: whether subtyping is determined by type structure or by type names in declarations, and in the former case what parts of the structure of types matter.
- Structural subtyping (subtyping determined by type structure) has desirable properties, such as supporting type matching in distributed and persistent systems. A disadvantage is the possibility of accidental matching of unrelated types. However, one can avoid such accidents by imposing distinctions on top of structural subtyping. In contrast, subtyping based on type names is hard to define precisely, and does not support structural subtyping.

- Here is a particularly simple form of structural subtyping. We assume, for two object types *O* and *O'*, that:

  O' <: *O*   if *O'* has the same components as *O* and possibly more

  where a component of an object type is the name of a field or a method and its associated type. So, for example, *ReCell* <: *Cell*.
- This definition implies that object types are invariant in their component types, although we could extend it to allow method specialization.
- When object types are defined recursively we need to be more careful about the definition of subtyping.

- With this definition of subtyping, object types naturally support multiple subtyping, because components are assumed unordered. For example, consider the object type:

```
ObjectType ReInteger is
    var contents: Integer;
    var backup: Integer;
    method restore();
end;
```

- Then we have both *ReCell* <: *Cell* and *ReCell* <: *ReInteger*.

- As a consequence of the new definition of subtyping we have:

(4) If *c'* is a subclass of *c* then *ObjectTypeOf*(*c'*) <: *ObjectTypeOf*(*c*).

- This holds simply because a subclass can only add new attributes to a class, and because we require that overriding methods preserve the existing method types.
- Property (4) is a reformulation for *ObjectTypeOf* of property (3), the subclassing-is-subtyping property. While property (3) is a double implication, the converse of (4) does not hold: there may be unrelated *c* and *c'* such that *ObjectTypeOf*(*c*)=*O* and *ObjectTypeOf*(*c'*)=*O'*, with *O'*<:*O*.
- Therefore, we have partially decoupled subclassing from subtyping, by relaxing the double implication (3) to the single implication (4). Subclassing still implies subtyping, so all the previous uses of subsumption are still allowed. But, since subsumption is based on subtyping and not subclassing, we now have even more freedom in subsumption.
- In conclusion, the notion of subclassing-is-subtyping can be weakened to subclassing-implies-subtyping without loss of expressiveness, and with a gain in separation between interfaces and implementations.

Subclassing without Subtyping

- We have seen how the partial decoupling of subtyping from subclassing increases the opportunities for subsumption.
- Another approach has emerged that increases the potential for inheritance by further separating subtyping from subclassing. This approach abandons completely the notion that subclassing implies subtyping (property (4)), and is known under the name *inheritance-is-not-subtyping*.
- It is largely motivated by the desire to handle contravariant (argument) occurrences of **Self** so as to allow inheritance of methods with arguments of type **Self**; these methods arise naturally in realistic examples.
- The price paid for this added flexibility in inheritance is decreased flexibility in subsumption. When **Self** is used liberally in contravariant positions, subclasses do not necessarily induce subtypes.

- Consider two types *Max* and *MinMax* for integers enriched with *min* and *max* methods. Each of these types is defined recursively:

```
ObjectType Max is
    var n: Integer;
    method max(other: Max): Max;
end;

ObjectType MinMax is
    var n: Integer;
    method max(other: MinMax): MinMax;
    method min(other: MinMax): MinMax;
end;
```

- Consider also two classes:

```
class maxClass is
    var n: Integer := 0;
    method max(other: Self): Self is
        if self.n>other.n then return self else return other end;
    end;
end;

subclass minMaxClass of maxClass is
    method min(other: Self): Self is
        if self.n<other.n then return self else return other end;
    end;
end;
```

- The methods *min* and *max* are called *binary* because they operate on two objects: **self** and *other*; the type of *other* is given by a contravariant occurrence of **Self**. Notice that the method *max*, which has an argument of type **Self**, is inherited from *maxClass* to *minMaxClass*.

- Intuitively the type *Max* corresponds to the class *maxClass*, and *MinMax* to *minMaxClass*. To make this correspondence more precise, we must define the meaning of *ObjectTypeOf* for classes containing occurrences of **Self**, so as to obtain *ObjectType-Of*(*maxClass*) = *Max* and *ObjectTypeOf*(*minMaxClass*) = *MinMax*.
- For these equations to hold, we map the use of **Self** in a class to the use of recursion in an object type. We also implicitly specialize **Self** for inherited methods; for example, we map the use of **Self** in the inherited method *max* to *MinMax*. In short, we obtain that any instance of *maxClass* has type *Max*, and any instance of *minMaxClass* has type *MinMax*.

- Although *minMaxClass* is a subclass of *maxClass*, *MinMax* cannot be a subtype of *Max*. Consider the class:

```
subclass minMaxClass' of minMaxClass is
    override max(other: Self): Self is
        if other.min(self)=other then return self else return other end;
    end;
end;
```

- For any instance *mm'* of *minMaxClass'* we have *mm'*:*MinMax*. If *MinMax* were a subtype of *Max*, then we would have also *mm'*:*Max*, and *mm'*.*max*(*m*) would be allowed for any *m* of type *Max*. Since *m* may not have a *min* attribute, the overridden *max* method of *mm'* may break. Therefore:

  MinMax <: *Max*   does not hold

- Thus, subclasses with contravariant occurrences of **Self** do not always induce subtypes.

Object Protocols

- Even when subclasses do not induce subtypes, we can find a relation between the type induced by a class and the type induced by one of its subclasses. It just so happens that, unlike subtyping, this relation does not enjoy the subsumption property. We now examine this new relation between object types.
- We cannot usefully quantify over the subtypes of *Max* because of the failure of subtyping. A parametric definition like:

ObjectOperator *P*[*M* <: *Max*] **is** … **end**;

- is not very useful; we could instantiate *P* by writing *P*[*Max*], but *P*[*MinMax*] would not be well-formed.

- Still, any object that supports the *MinMax* protocol, in an intuitive sense, supports also the *Max* protocol. There seems to be an opportunity for some kind of *subprotocol* relation that may allow useful parameterization. In order to find this subprotocol relation, we introduce two type operators, *MaxProtocol* and *MinMaxProtocol*:

```
ObjectOperator MaxProtocol[X] is
    var n:Integer;
    method max(other: X): X;
end;
ObjectOperator MinMaxProtocol[X] is
    var n:Integer;
    method max(other: X): X;
    method min(other: X): X;
end;
```

- Generalizing from this example, we can always pass uniformly from a recursive type *T* to an operator *T-Protocol* by abstracting over the recursive occurrences of *T*. The operator *T-Protocol* is a function on types; taking the fixpoint of *T-Protocol* yields back *T*.
- We find two formal relationships between *Max* and *MinMax*. First, *MinMax* is a post-fixpoint of *MaxProtocol*, that is:

$$MinMax <: MaxProtocol[MinMax]$$

- Second, let $\prec:$ denote the higher-order subtype relation between type operators:

$$P \prec: P' \quad \text{iff} \quad P[T] <: P'[T] \quad \text{for all types } T$$

- Then, the protocols of *Max* and *MinMax* satisfy:

$$MinMaxProtocol \prec: MaxProtocol$$

- Either of these two relationships can be taken as our hypothesized notion of subprotocol:

$$S \text{ subprotocol } T \quad \text{if} \quad S <: T\text{-}Protocol[S]$$

or

$$S \text{ subprotocol } T \quad \text{if} \quad S\text{-}Protocol \prec: T\text{-}Protocol$$

- The second relationship expresses a bit more directly the fact that there exists a subprotocol relation, and that this is in fact a relation between operators, not between types.
- Whenever we have some property common to several types, we may think of parameterizing over these types. So we may adopt one of the following forms of parameterization:

ObjectOperator $P_1[X <: MaxProtocol[X]]$ **is** … **end**;
ObjectOperator $P_2[P \prec: MaxProtocol]$ **is** … **end**;

- Then we can instantiate P_1 to $P_1[MinMax]$, and P_2 to $P_2[MinMaxProtocol]$.

- These two forms of parameterization seem to be equally expressive in practice. The first one is called *F-bounded parameterization*. The second form is *higher-order bounded parameterization*, defined via pointwise subtyping of type operators.
- Instead of working with type operators, a programming language supporting subprotocols may conveniently define a *matching* relation (denoted by <#) directly over types. The properties of the matching relation are designed to correspond to the definition of subprotocol. Depending on the choice of subprotocol relation, we have:

$$S <\# T \quad \text{if} \quad S <: T\text{-}Protocol[S] \qquad \text{(F-bounded interpretation)}$$

$$\text{or}$$

$$S <\# T \quad \text{if} \quad S\text{-}Protocol \prec: T\text{-}Protocol \qquad \text{(higher-order interpretation)}$$

- With either definition we have *MinMax* <# *Max*.

- Matching does not enjoy a subsumption property (that is, $S <\# T$ and $s : S$ do not imply that $s : T$); however, matching is useful for parameterizing over all the types that match a given one:

  ObjectOperator $P_3[X <\# Max]$ **is** … **end**;

- The instantiation $P_3[MinMax]$ is legal.
- In summary, even in the presence of contravariant occurrences of **Self**, and in absence of subtyping, there can be inheritance of binary methods like *max*. Unfortunately, subsumption is lost in this context, and quantification over subtypes is no longer very useful. These disadvantages are partially compensated by the existence of a subprotocol relation, and by the ability to parameterize with respect to this relation.

LECTURE 4

- A Language with Subtyping.

The Language O–1

Synthesis of a Language

- O–1 is a language built out of constructs from object calculi.
  - ~ The main purpose of O–1 is to help us assess the contributions of object calculi.
  - ~ In addition, O–1 embodies a few intriguing language-design ideas.
  - ~ We have studied more advanced languages that include Self types and parametric polymorphism.

Some Features of O–1

- Both class-based and object-based constructs.
- First-order object types with subtyping and variance annotations.
- Classes with single inheritance.
- Method overridding and specialization.
- Recursion.
- Typecase. (To compensate for, e.g., lack of Self types.)
- Separation of interfaces from implementations.
- Separation of inheritance from subtyping.

Some Non-Features of O–1

- No public/private/protected/abstract, etc.,
- No cloning,
- No basic types, such as integers,
- No arrays and other data structures,
- No procedures,
- No concurrency.

Syntax of Types

Syntax of O–1 types

$A,B ::=$	types
X	type variable
Top	the biggest type
Object$(X)[l_i \upsilon_i{:}B_i \ ^{i \in 1..n}]$	object type (l_i distinct)
Class(A)	class type

- Roughly, we may think **Object** = μ.
  But the fold/unfold coercions do not appear in the syntax of O–1.
- Usually, $^+$ variance is for methods, and ^{o} variance is for fields.

Syntax of Programs

Syntax of O–1 terms

a,b,c ::=	terms
x	variable
object$(x{:}A)\ l_i{=}b_i\ ^{i\in 1..n}$ **end**	direct object construction
$a.l$	field selection / method invocation
$a.l := b$	update with a term
$a.l :=$ **method**$(x{:}A)\ b$ **end**	update with a method
new c	object construction from a class
root	root class
subclass of $c{:}C$ **with**$(x{:}A)$	subclass
$l_i{=}b_i\ ^{i\in n+1..n+m}$	additional attributes
override $l_i{=}b_i\ ^{i\in Ovr\subseteq 1..n}$ **end**	overridden attributes
$c\hat{\ }l(a)$	class selection
typecase a **when** $(x{:}A)b_1$ **else** b_2 **end**	typecase

- Superclass attributes are inherited “automatically”. (No copying premethods by hand as in the encodings of classes.)
- Inheritance “by hand” still possible by class selection c^$l(a)$.
- Classes are first-class values.
- Parametric classes can be written as functions that return classes.

- We could drop the object-based constructs (object construction and method update). The result would be a language expressive enough for traditional class-based programming.
- Alternatively, we could drop the class-based construct (root class, subclass, new, and class selection), obtaining an object-based language.
- Classes, as well as objects, are first-class values. A parametric class can be obtained as a function that returns a class.

Abbreviations

Root $\triangleq$

  Class(**Object**(X)[])

class with$(x{:}A)$ $l_i{=}b_i\ ^{i\in 1..n}$ **end** $\triangleq$

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

subclass of $c{:}C$ **with** $(x{:}A)$ … **super**.l … **end** $\triangleq$

  subclass of $c{:}C$ **with** $(x{:}A)$ … $c\hat{\ }l(x)$ … **end**

object$(x{:}A)$ … l **copied from** c … **end** $\triangleq$

  object$(x{:}A)$ … $l{=}c\hat{\ }l(x)$ … **end**

N.B.: conversely, **subclass** could be defined from **class** and $c\hat{\ }l$.

Examples

- We assume basic types (*Bool*, *Int*) and function types ($A \to B$, contravariant in A and covariant in B).

$$Point \triangleq \textbf{Object}(X)[x\colon Int,\ eq^{+}\colon X \to Bool,\ mv^{+}\colon Int \to X]$$

$$CPoint \triangleq \textbf{Object}(X)[x\colon Int,\ c\colon Color,\ eq^{+}\colon Point \to Bool,\ mv^{+}\colon Int \to Point]$$

- $CPoint <: Point$

- The type of *mv* in *CPoint* is $Int \to Point$.
  One can explore the effect of changing it to $Int \to X$.

- The type of *eq* in *CPoint* is $Point \to Bool$.
  If we were to change it to $X \to Bool$ we would lose the subtyping $CPoint <: Point$.

Class(Point)

```
pointClass : Class(Point)   ≜
    class with (self: Point)
        x = 0,
        eq = fun(other: Point) self.x = other.x end,
        mv = fun(dx: Int) self.x := self.x+dx end
    end
```

Class(CPoint)

```
$cPointClass$ : **Class**($CPoint$)   $\triangleq$
    **subclass of** $pointClass$: **Class**($Point$)
    **with** ($self$: $CPoint$)
        $c = black$
    **override**
        $eq$ = **fun**($other$: $Point$)
                **typecase** $other$
                **when** ($other'$: $CPoint$) **super**.$eq$($other'$) $and\ self.c = other'.c$
                **else** $false$
                **end**
            **end**
    **end**
```

Comments

- The class *cPointClass* inherits *x* and *mv* from its superclass *pointClass*.
- Although it could inherit *eq* as well, *cPointClass* overrides this method as follows.
  - ~ The definition of *Point* requires that *eq* work with any argument *other* of type *Point*.
  - ~ In the *eq* code for *cPointClass*, the typecase on *other* determines whether *other* has a color.
  - ~ If so, *eq* works as in *pointClass* and in addition tests the color of *other*.
  - ~ If not, *eq* returns *false*.

- We can use *cPointClass* to create color points of type *CPoint*:

  $cPoint : CPoint \triangleq \textbf{new}\ cPointClass$

- Calls to *mv* lose the color information.

- In order to access the color of a point after it has been moved, a typecase is necessary:

```
movedColor : Color  ≜
      typecase cPoint.mv(1)
      when (cp: CPoint) cp.c
      else black
      end
```

Typing

- The rules of O–1 are based on the following judgments:

Judgments

$E \vdash \diamond$	environment E is well-formed
$E \vdash A$	A is a well-formed type in E
$E \vdash A <: B$	A is a subtype of B in E
$E \vdash \upsilon A <: \upsilon' B$	A is a subtype of B in E, with variance annotations υ and υ'
$E \vdash a : A$	a has type A in E

- The rules for environments are standard:

Environments

(Env $\emptyset$)
$$\frac{}{\emptyset \vdash \diamond}$$

(Env X<:)
$$\frac{E \vdash A \qquad X \notin dom(E)}{E, X <: A \vdash \diamond}$$

(Env x)
$$\frac{E \vdash A \qquad x \notin dom(E)}{E, x : A \vdash \diamond}$$

Type Formation Rules

Types

(Type X)

$$\frac{E', X{<:}A, E'' \vdash \diamond}{E', X{<:}A, E'' \vdash X}$$

(Type Top)

$$\frac{E \vdash \diamond}{E \vdash \mathbf{Top}}$$

(Type Object) (l_i distinct, $\upsilon_i \in \{^{\circ}, ^{-}, ^{+}\}$)

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

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

$$\frac{E \vdash A}{E \vdash \mathbf{Class}(A)}$$

Subtyping Rules

- Note that there is no rule for subtyping class types.

Subtyping

(Sub Refl)
$$\frac{E \vdash A}{E \vdash A <: A}$$

(Sub Trans)
$$\frac{E \vdash A <: B \qquad E \vdash B <: C}{E \vdash A <: C}$$

(Sub X)
$$\frac{E', X{<:}A, E'' \vdash \diamond}{E', X{<:}A, E'' \vdash X <: A}$$

(Sub Top)
$$\frac{E \vdash A}{E \vdash A <: \mathbf{Top}}$$

(Sub Object) (where $A \equiv \mathbf{Object}(X)[l_i \upsilon_i{:}B_i\{X\}^{\,i\in 1..n+m}]$, $A' \equiv \mathbf{Object}(X')[l_i \upsilon_i'{:}B_i'\{X'\}^{\,i\in 1..n}]$)
$$\frac{E \vdash A \qquad E \vdash A' \qquad E, X{<:}A' \vdash \upsilon_i\, B_i\{X\} <: \upsilon_i'\, B_i'\{\!\{A'\}\!\} \qquad \forall\, i\in 1..n}{E \vdash A <: A'}$$

(Sub Invariant)
$$\frac{E \vdash B}{E \vdash {}^{o}B <: {}^{o}B}$$

(Sub Covariant)
$$\frac{E \vdash B <: B' \qquad \upsilon\in\{{}^{o},{}^{+}\}}{E \vdash \upsilon\, B <: {}^{+}B'}$$

(Sub Contravariant)
$$\frac{E \vdash B' <: B \qquad \upsilon\in\{{}^{o},{}^{-}\}}{E \vdash \upsilon\, B <: {}^{-}B'}$$

Term Typing Rules

Terms

(Val Subsumption)

$$\frac{E \vdash a : A \qquad E \vdash A <: B}{E \vdash a : B}$$

(Val x)

$$\frac{E', x{:}A, E'' \vdash \diamond}{E', x{:}A, E'' \vdash x : A}$$

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

$$\frac{E, x{:}A \vdash b_i : B_i\{\!\{A\}\!\} \qquad \forall i \in 1..n}{E \vdash \textbf{object}(x{:}A)\ l_i{=}b_i^{\,i \in 1..n}\ \textbf{end} : A}$$

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

$$\frac{E \vdash a : A \qquad \upsilon_j \in \{^{\circ}, ^{+}\} \qquad j\in 1..n}{E \vdash a.l_j : B_j\{\!\{A\}\!\}}$$

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

$$\frac{E \vdash a : A \qquad E \vdash b : B_j\{\!\{A\}\!\} \qquad \upsilon_j \in \{^{\circ}, ^{-}\} \qquad j\in 1..n}{E \vdash a.l_j := b : A}$$

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

$$\frac{E \vdash a : A \qquad E, x{:}A \vdash b : B_j\{\!\{A\}\!\} \qquad \upsilon_j \in \{^{\circ}, ^{-}\} \qquad j\in 1..n}{E \vdash a.l_j := \textbf{method}(x{:}A)b\ \textbf{end} : A}$$

(Val New)

$$\frac{E \vdash c : \mathbf{Class}(A)}{E \vdash \mathbf{new}\ c : A}$$

(Val Root)

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

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

$$\frac{E \vdash a : A \qquad E \vdash c : \mathbf{Class}(A) \qquad j \in 1..n}{E \vdash c\hat{\ }l_j(a) : B_j\{\!\{A\}\!\}}$$

(Val Typecase)

$$\frac{E \vdash a : A' \qquad E, x{:}A \vdash b_1 : D \qquad E \vdash b_2 : D}{E \vdash \mathbf{typecase}\ a\ \mathbf{when}\ (x{:}A)b_1\ \mathbf{else}\ b_2\ \mathbf{end} : D}$$

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

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

- Legend

A subclass object type; A' superclass object type; Ovr indices of overriden methods.

$E \vdash A <: A'$ The “class rule”:

“type generated by subclass <: type generated by superclass”

Allows “method specialiazation” $l_i^+{:}B_i <: l_i^+{:}B_i'$ for $i \in Ovr$

$E \vdash B_i'\{\!\{A'\}\!\} <: B_i\{\!\{A\}\!\}$ Toghether with $A <: A'$ requires type invariance for an inherited method. If this condition does not hold, the method must be overridden.

$E, x{:}A \vdash b_i : B_i\{\!\{A\}\!\}$ Checking the bodies of overridden and additional method.

Translation

- We give a translation into a functional calculus (with all the features described earlier).
- A similar translation could be given into an appropriate imperative calculus.
- At the level of types, the translation is simple.
  - ~ We write $\langle\!\langle A\rangle\!\rangle$ for the translation of A.
  - ~ We map an object type **Object**$(X)[l_i \upsilon_i{:}B_i \ ^{i\in 1..n}]$ to a recursive object type $\mu(X)[l_i \upsilon_i{:}\langle\!\langle B_i\rangle\!\rangle \ ^{i\in 1..n}]$.
  - ~ We map a class type **Class**(**Object**$(X)[l_i \upsilon_i{:}B_i\{X\} \ ^{i\in 1..n}]$) to an object type that contains components for pre-methods and a *new* component.

Translation of Types

Translation of O–1 types

$\langle\!\langle X \rangle\!\rangle \triangleq X$

$\langle\!\langle \mathbf{Top} \rangle\!\rangle \triangleq \mathit{Top}$

$\langle\!\langle \mathbf{Object}(X)[l_i \upsilon_i{:}B_i\ ^{i\in 1..n}] \rangle\!\rangle \triangleq \mu(X)[l_i \upsilon_i{:}\langle\!\langle B_i \rangle\!\rangle\ ^{i\in 1..n}]$

$\langle\!\langle \mathbf{Class}(A) \rangle\!\rangle \triangleq [\mathit{new}^+{:}\langle\!\langle A \rangle\!\rangle,\ l_i^+{:}\langle\!\langle A \rangle\!\rangle \rightarrow \langle\!\langle B_i \rangle\!\rangle\{\!\{\langle\!\langle A \rangle\!\rangle\}\!\}\ ^{i\in 1..n}]$
where $A \equiv \mathbf{Object}(X)[l_i \upsilon_i{:}B_i\{X\}\ ^{i\in 1..n}]$

Translation of O–1 environments

$\langle\!\langle \emptyset \rangle\!\rangle \triangleq \emptyset$

$\langle\!\langle E, X{<:}A \rangle\!\rangle \triangleq \langle\!\langle E \rangle\!\rangle, X{<:}\langle\!\langle A \rangle\!\rangle$

$\langle\!\langle E, x{:}A \rangle\!\rangle \triangleq \langle\!\langle E \rangle\!\rangle, x{:}\langle\!\langle A \rangle\!\rangle$

Translation of Terms

- The translation is guided by the type structure.
- The translation maps a class to a collection of pre-methods plus a *new* method.
  - For a class **subclass of** *c'* … **end**, the collection of pre-methods consists of the pre-methods of *c'* that are not overridden, plus all the pre-methods given explicitly.
  - The *new* method assembles the pre-methods into an object; **new** *c* is interpreted as an invocation of the *new* method of $\langle\!\langle c \rangle\!\rangle$.
  - The construct $c\hat{\ }l(a)$ is interpreted as the extraction and the application of a pre-method.

(Simplified) Translation of O–1 terms

$\langle\!\langle x \rangle\!\rangle \triangleq x$

$\langle\!\langle \textbf{object}(x{:}A)\ l_i{=}b_i\ ^{i\in 1..n}\ \textbf{end} \rangle\!\rangle \triangleq [l_i{=}\varsigma(x{:}\langle\!\langle A \rangle\!\rangle)\langle\!\langle b_i \rangle\!\rangle\ ^{i\in 1..n}]$

$\langle\!\langle a.l \rangle\!\rangle \triangleq \langle\!\langle a \rangle\!\rangle.l$

$\langle\!\langle a.l := b \rangle\!\rangle \triangleq \langle\!\langle a \rangle\!\rangle.l{:=}\langle\!\langle b \rangle\!\rangle$

$\langle\!\langle a.l := \textbf{method}(x{:}A)\ b\ \textbf{end} \rangle\!\rangle \triangleq \langle\!\langle a \rangle\!\rangle.l \Leftarrow \varsigma(x{:}\langle\!\langle A \rangle\!\rangle)\langle\!\langle b \rangle\!\rangle$

$\langle\!\langle$**new** $c\rangle\!\rangle \triangleq \langle\!\langle c\rangle\!\rangle.new$

$\langle\!\langle$**root**$\rangle\!\rangle \triangleq [new=[]]$

$\langle\!\langle$**subclass of** c':**Class**(A') **with**$(x{:}A)$ $l_i{=}b_i\ ^{i\in n+1..n+m}$ **override** $l_i{=}b_i\ ^{i\in Ovr}$ **end**$\rangle\!\rangle \triangleq$

$[new{=}\varsigma(z{:}\langle\!\langle \mathbf{Class}(A)\rangle\!\rangle)[l_i{=}\varsigma(s{:}\langle\!\langle A\rangle\!\rangle)z.l_i(s)\ ^{i\in 1..n+m}],$

$l_i{=}\langle\!\langle c'\rangle\!\rangle.l_i\ ^{i\in 1..n-Ovr},$

$l_i{=}\lambda(x{:}\langle\!\langle A\rangle\!\rangle)\langle\!\langle b_i\rangle\!\rangle\ ^{i\in Ovr\cup n+1..n+m}]$

$\langle\!\langle c\hat{\ }l(a)\rangle\!\rangle \triangleq \langle\!\langle c\rangle\!\rangle.l(\langle\!\langle a\rangle\!\rangle)$

$\langle\!\langle$**typecase** a **when** $(x{:}A)b_1$ **else** b_2 **end**$\rangle\!\rangle \triangleq$ $typecase\ \langle\!\langle a\rangle\!\rangle \mid (x{:}\langle\!\langle A\rangle\!\rangle)\langle\!\langle b_1\rangle\!\rangle \mid \langle\!\langle b_2\rangle\!\rangle$

Soundness

- If E ⊢ J is valid in O–1, then ⟪E ⊢ J⟫ is valid in the object calculus.
- The object subtyping rule relies on the following rule for recursive types:

(Sub Rec')

$$\frac{E \vdash \mu(X)A\{X\} \qquad E \vdash \mu(Y)B\{Y\} \qquad E, X{<:}\mu(Y)B\{Y\} \vdash A\{X\} <: B\{\!\{\mu(Y)B\{Y\}\}\!\}}{E \vdash \mu(X)A\{X\} <: \mu(Y)B\{Y\}}$$

- The most interesting case is for **subclass**. We need to check:

$\langle\!\langle$**subclass of** c':**Class**(A') **with**$(x{:}A)$ $l_i{=}b_i$ $^{i\in n+1..n+m}$ **override** $l_i{=}b_i$ $^{i\in Ovr}$ **end**$\rangle\!\rangle$
: $\langle\!\langle$**Class**$(A)\rangle\!\rangle$

That is:

$$\begin{array}{l} [\mathit{new}{=}\varsigma(z{:}\langle\!\langle \mathbf{Class}(A)\rangle\!\rangle)[l_i{=}\varsigma(s{:}\langle\!\langle A\rangle\!\rangle)z.l_i(s)^{\ i\in 1..n+m}], \\ \qquad l_i{=}\langle\!\langle c'\rangle\!\rangle.l_i^{\ i\in 1..n-Ovr}, \\ \qquad l_i{=}\lambda(x{:}\langle\!\langle A\rangle\!\rangle)\langle\!\langle b_i\rangle\!\rangle^{\ i\in Ovr\cup n+1..n+m}] \\ : [\mathit{new}^{+}{:}\langle\!\langle A\rangle\!\rangle,\ l_i^{+}{:}\langle\!\langle A\rangle\!\rangle\to\langle\!\langle B_i\rangle\!\rangle\{\langle\!\langle A\rangle\!\rangle\}^{\ i\in 1..n}] \end{array}$$

- *new* checks by computation.
- $i\in Ovr\cup n+1..n+m$ checks by one the (Val Subclass) hypotheses.
- $i\in 1..n-Ovr$ (inherited methods) checks as follows:

  $\langle\!\langle c'\rangle\!\rangle : \langle\!\langle \mathbf{Class}(A')\rangle\!\rangle$ by hypothesis. Hence:

  $\langle\!\langle c'\rangle\!\rangle.l_i : \langle\!\langle A'\rangle\!\rangle\to\langle\!\langle B_i'\rangle\!\rangle\{\langle\!\langle A'\rangle\!\rangle\}$. Moreover:

  $\langle\!\langle A'\rangle\!\rangle\to\langle\!\langle B_i'\rangle\!\rangle\{\langle\!\langle A'\rangle\!\rangle\} <: \langle\!\langle A\rangle\!\rangle\to\langle\!\langle B_i\rangle\!\rangle\{\langle\!\langle A\rangle\!\rangle\}$ directly from hypotheses. So:

  $\langle\!\langle c'\rangle\!\rangle.l_i : \langle\!\langle A\rangle\!\rangle\to\langle\!\langle B_i\rangle\!\rangle\{\langle\!\langle A\rangle\!\rangle\}$ by subsumption.

Usefulness of the Translation

- The translation validates the typing rules of O–1. That is, if $E \vdash J$ is valid in O–1, then $\langle\!\langle E \vdash J \rangle\!\rangle$ is valid in the object calculus.
- The translation served as an important guide in finding sound typing rules for O–1, and for "tweaking" them to make them both simpler and more general.
- In particular, typing rules for subclasses are so inherently complex that it is difficult to "guess" them correctly without the aid of some interpretation.
- Thus, we have succeeded in using object calculi as a platform for explaining a relatively rich object-oriented language and for validating its type rules.

LECTURE 5

- Matching Issues.

MATCHING

- 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.

When Subtyping Works

- A simple treatment of objects, classes, and inheritance is possible for covariant Self types (only).

Object Types

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

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

- A typical object of type Inc is:

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

Subtyping

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

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

- For object types, we have the subtyping rule:

  $[v_i{:}B_i{}^{i\in I}, m_j{}^{+}{:}C_j{}^{j\in J}] \;<:\; [v_i{:}B_i{}^{i\in I'}, m_j{}^{+}{:}C_j'{}^{j\in J'}]$

  if $C_j <: C_j'$ for all $j\in J'$, with $I'\subseteq I$ and $J'\subseteq J$

- For recursive types we have the subtyping rule:

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

  if $X <: Y$ implies $A\{X\} <: B\{Y\}$

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

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

  if $X <: Y$ implies $C_j\{X\} <: C_j'\{Y\}$ for all $j\in J'$, with $I'\subseteq I$ and $J'\subseteq J$

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

  IncDec <: Inc

Pre-Methods

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

$$\text{pre-inc} : \forall(X<:\text{Inc})X\rightarrow X \quad \triangleq$$
$$\lambda(X<:\text{Inc})\ \lambda(\text{self}:X)\ \text{self.n} := \text{self.n}+1 \qquad \text{(using a ``structural'' rule)}$$

- We call a code fragment such as pre-inc a *pre-method*.
- N.B. it is not enough to have pre-inc : Inc→Inc if we want to inherit this pre-method in IncDec classes. Here polymorphism is essential.
- N.B. the body of pre-inc is typed by means of a *structural* rule for update, which is essential in many examples involving bounded quantification.
- We can specialize pre-inc to implement the method inc of type Inc or IncDec:

$$\text{pre-inc(Inc)} : \text{Inc}\rightarrow\text{Inc}$$
$$\text{pre-inc(IncDec)} : \text{IncDec}\rightarrow\text{IncDec}$$

- 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 $\forall(X<:A)X\to C_j\{X\}$. For example:

$$\begin{aligned}
&\mathrm{IncClass} \triangleq \\
&\quad [\mathrm{new}^+ : \mathrm{Inc}, \\
&\quad\ \ n: \mathrm{Int}, \\
&\quad\ \ \mathrm{inc}: \forall(X<:\mathrm{Inc})X\to X]
\end{aligned}$$

N.B.: $\mathrm{inc}: \mathrm{Inc}\to\mathrm{Inc}$
would not allow inheritance

$$\begin{aligned}
&\mathrm{IncDecClass} \triangleq \\
&\quad [\mathrm{new}^+ : \mathrm{IncDec}, \\
&\quad\ \ n: \mathrm{Int}, \\
&\quad\ \ \mathrm{inc}: \forall(X<:\mathrm{IncDec})X\to X, \\
&\quad\ \ \mathrm{dec}: \forall(X<:\mathrm{IncDec})X\to X]
\end{aligned}$$

- A typical class of type IncClass reads:

$$\begin{aligned}
&\mathrm{incClass} : \mathrm{IncClass} \triangleq \\
&\quad [\mathrm{new} = \varsigma(\mathrm{classSelf}: \mathrm{IncClass}) \\
&\qquad\qquad [n = \mathrm{classSelf}.n,\ \mathrm{inc} = \varsigma(\mathrm{self}{:}\mathrm{Inc})\ \mathrm{classSelf}.\mathrm{inc}(\mathrm{Inc})(\mathrm{self})] \\
&\quad\ \ n = 0, \\
&\quad\ \ \mathrm{inc} = \text{pre-inc}]
\end{aligned}$$

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 $\forall(X<:Inc)X\rightarrow X$ in a class for Inc could be re-used as a pre-method of type $\forall(X<:IncDec)X\rightarrow X$ in a class for IncDec:

$$
\begin{array}{l}
incDecClass : IncDecClass \;\triangleq \\
\quad [new = \varsigma(classSelf{:}\ IncDecClass)[...], \\
\quad\; n = 0, \\
\quad\; inc = incClass.inc, \\
\quad\; dec = ...]
\end{array}
$$

- This example of inheritance requires the subtyping:

$$\forall(X<:Inc)X\rightarrow X \quad <: \quad \forall(X<:IncDec)X\rightarrow X$$

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

$$
\begin{array}{ll}
\forall(X<:A)B <: \forall(X<:A')B' & \text{if } A'<:A \text{ and if } X<:A \text{ implies } B<:B' \\
A\rightarrow B <: A'\rightarrow B' & \text{if } A' <: A \text{ and } B <: B'
\end{array}
$$

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.

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

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

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

- Problem:

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

  according to the rules we have adopted, since :

  $$Y <: X \;\;\not\Rightarrow\;\; Y\to Y <: X\to X \qquad \text{for } \mathrm{max}^{+}$$

  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

- If subtyping doesn’t work, maybe some other relation between types will.
- A possible replacement for subtyping: *matching*.

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*.

- For recursive object types we have:

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

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

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

$$\text{pre-inc(IncDec)} : \text{IncDec}\to\text{IncDec}$$

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

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

$$\text{pre-max(MinMax)} : \text{MinMax}\to\text{MinMax}\to\text{MinMax} \qquad \text{(didn't hold with <:)}$$

- 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:

  $\text{inc} : \text{Inc}\to\text{Inc} \;\triangleq$
  $\quad\lambda(x{:}\text{Inc})\ x.n := x.n+1$

  $\lambda(x{:}\text{IncDec})\ \text{inc}(x)$

  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:

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

  $\lambda(x{:}\text{IncDec})\ \text{pre-inc}(\text{IncDec})(x)$

Matching and Classes

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

$$
\begin{aligned}
&\mathrm{MaxClass} \triangleq\\
&\quad [\mathrm{new}^{+}: \mathrm{Max},\\
&\quad\ \mathrm{n}: \mathrm{Int},\\
&\quad\ \mathrm{max}: \forall(X {<\#} \mathrm{Max})X \to X \to X]
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{MinMaxClass} \triangleq\\
&\quad [\mathrm{new}^{+}: \mathrm{MinMax},\\
&\quad\ \mathrm{n}: \mathrm{Int},\\
&\quad\ \mathrm{max}: \forall(X {<\#} \mathrm{MinMax})X \to X \to X,\\
&\quad\ \mathrm{min}: \forall(X {<\#} \mathrm{MinMax})X \to X \to X]
\end{aligned}
$$

- A typical class of type MaxClass reads:

$$
\begin{aligned}
&\mathrm{maxClass} : \mathrm{MaxClass} \triangleq\\
&\quad [\mathrm{new} = \varsigma(\mathrm{classSelf}: \mathrm{MaxClass})\\
&\qquad\qquad [\mathrm{n} = \mathrm{classSelf.n},\ \ \mathrm{max} = \varsigma(\mathrm{self}{:}\mathrm{Max})\ \mathrm{classSelf.max}(\mathrm{Max})(\mathrm{self})],\\
&\quad\ \mathrm{n} = 0,\\
&\quad\ \mathrm{max} = \text{pre-max}]
\end{aligned}
$$

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

$$\forall(X<\#Max)X\to X\to X \quad <: \quad \forall(X<\#MinMax)X\to X\to X$$

- 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 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 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:

$$F \prec: G \qquad \text{if, for all } X, \quad F(X) <: G(X)$$

- The property:

$$A_{Op} \prec: B_{Op} \qquad (A_{Op} \text{ is a suboperator of } B_{Op})$$

is seen as a statement that A extends B.

$$\begin{aligned}&MinMax_{Op} \;\equiv\; \lambda(X)\,[n{:}Int,\ max^{+}{:}X\to X,\ min^{+}{:}\ X\to X]\\ &\prec: \quad \lambda(X)\,[n{:}Int,\ max^{+}{:}X\to X,\ min^{+}{:}\ X\to X] \;\equiv\; Max_{Op}\end{aligned}$$

- We obtain:

$$\begin{array}{lll} \mathrm{Max}_{\mathrm{Op}} \prec: \mathrm{Max}_{\mathrm{Op}} & & \\ \mathrm{MinMax}_{\mathrm{Op}} & (\forall X. & [n:\mathrm{Int},\ \mathrm{max}^{+}:X\to X,\ \mathrm{min}^{+}: X\to X] \\ \quad \prec: \mathrm{Max}_{\mathrm{Op}} & \quad <: & [n:\mathrm{Int},\ \mathrm{max}^{+}:X\to X]) \end{array}$$

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

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

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.

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

$$\text{pre-max}(MinMax_{Op}) : MinMax \to MinMax \to MinMax$$

This typechecks, e.g.:

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

(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:

$$A <\# B \quad \approx \quad A_{Op} \prec: B_{Op}$$

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

  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:

$$A <\# B \quad \approx \quad \mathit{Oper}\langle A\rangle \prec: \mathit{Oper}\langle B\rangle$$

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

- The two translations, *Type*⟨A⟩ and *Oper*⟨A⟩, can be summarized as follows.

  For object types of the source language, we set:

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

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

  For other types, we set:

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

- For instance:

$$\mathit{Type}\langle\forall(X{<}\#\mathrm{Max})\ \forall(Y{<}\#X)\ X\to Y\rangle \ \approx$$
$$\forall(X{\prec}{:}\mathrm{Max}_{\mathrm{Op}})\ \forall(Y{\prec}{:}X)\ X^*\to Y^*$$

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:

$$\mathrm{A} <\# \mathrm{A} \qquad \approx \qquad \mathit{Oper}\langle \mathrm{A}\rangle \prec: \mathit{Oper}\langle \mathrm{A}\rangle$$

  This follows from the reflexivity of $\prec:$.

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

$$\begin{aligned} &\mathrm{A} <\# \mathrm{B} \quad \text{and} \quad \mathrm{B} <\# \mathrm{C} \quad \text{imply} \quad \mathrm{A} <\# \mathrm{C} \quad \approx \\ &\qquad \mathit{Oper}\langle \mathrm{A}\rangle \prec: \mathit{Oper}\langle \mathrm{B}\rangle \quad \text{and} \quad \mathit{Oper}\langle \mathrm{B}\rangle \prec: \mathit{Oper}\langle \mathrm{C}\rangle \\ &\qquad\qquad \text{imply} \quad \mathit{Oper}\langle \mathrm{A}\rangle \prec: \mathit{Oper}\langle \mathrm{C}\rangle \end{aligned}$$

  This follows from the transitivity of $\prec:$.

Matching Self

- With the higher-order interpretation, the relation:

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

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

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

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

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

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:

$$
\begin{aligned}
&\mathrm{MaxClass} \quad \triangleq \\
&\quad [\mathrm{new}^{+}: \mathrm{Max}, \\
&\quad\ \ \mathrm{n}: \mathrm{Int}, \\
&\quad\ \ \mathrm{max}: \forall(X \prec: \mathrm{Max}_{\mathrm{Op}})X^{*} \rightarrow X^{*} \rightarrow X^{*}]
\end{aligned}
$$

The corresponding translation at the term level produces:

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

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

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

The instantiations pre-max(Max_{Op}) and pre-max($\text{MinMax}_{\text{Op}}$) are both legal. Since pre-max has type $\forall(X \prec: \text{Max}_{\text{Op}})X^* \to X^* \to X^*$, 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:

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

so pre-max has type $\forall(X \prec: \text{MinMax}_{\text{Op}})X^* \to X^* \to X^*$ 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 $X \prec: Inc_{Op}$ and x:X* then unfold(x).n : Int

  if $X \prec: Inc_{Op}$ 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 $X \prec: Inc_{Op}$ we obtain X(X*) <: Inc_{Op}(X*) = [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*) <: [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 <: [v:C,...] then (a.v:=c) : A. In our case, we have unfold(x):X(X*), b:Int, and X(X*) <: [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.

LECTURE 6

- A Language with Matching (28 slides).
- Translations Summary (4 slides).
- Conclusions (4 slides).

A Language with Matching

- Many features of O–3 are familiar: for example, object types, class types, and single inheritance.
- The main new feature is a matching relation, written <#. The matching relation is defined only between object types, and between variables bounded by object types.
- An object type $A \equiv$ **Object**$(X)[\ldots]$ matches another object type $B \equiv$ **Object**$(X)[\ldots]$ (written A <# B) when all the object components of B are literally present in A, including any occurrence of the common variable X.

Object$(X)[l{:}X\to X,\ m{:}X]$ <# **Object**$(X)[l{:}X\to X]$	Yes
Object$(X)[l{:}X\to X,\ m{:}X]$ $<:$ **Object**$(X)[l{:}X\to X]$	No

- Matching is the basis for inheritance in O–3. That is, if A <# B, then a method of a class for B may be inherited as a method of a class for A.
- In particular, binary methods can be inherited. For example, a method l of a class for **Object**$(X)[l{:}X\to X]$ can be inherited as a method of a class for **Object**$(X)[l{:}X\to X,\ m{:}X]$.

- Matching does not support subsumption: when a has type A and A <# B, it is not sound in general to infer that a has type B.
- With the loss of subsumption, it is often necessary to parameterize over all types that match a given type. For example, a function with type $(\mathbf{Object}(X)[l{:}X\rightarrow X])\rightarrow C$ may have to be rewritten, for flexibility, with type $\mathbf{All}(Y{<}\#\mathbf{Object}(X)[l{:}X\rightarrow X])Y\rightarrow C$, enabling the application to an object of type $\mathbf{Object}(X)[l{:}X\rightarrow X,\ m{:}X]$.
- No subtype relation appears in the syntax of O–3, although subtyping is still used in its type rules.
- We will have that if A and B are object types and $A <: B$, then A <# B. Moreover, if all occurrences of Self in B are covariant and A <# B, then $A <: B$.

The Language O–3

Syntax of O–3

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

$a,b,\mathrm{c} ::=$	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
$c\hat{}l(A,a)$	class selection
fun$(X{<}\#A)\ b$ **end**	match-polymorphic abstraction
$b(A)$	match-polymorphic instantiation

Convenient abbreviations

Root $\triangleq$

  Class(**Object**(X)[])

class with(x:X<#A) l_i=b_i{X,x} $^{i\in 1..n}$ **end** $\triangleq$

  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** $\triangleq$

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

object(x:X=A) ... l **copied from** c ... **end** $\triangleq$

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

$a.l$:= b $\triangleq$ where $X,x \notin$ FV(b) and a:A,

  $a.l$:= **method**(x:X<#A) b **end** with A clear from context

Example: Points

$$Point \triangleq \mathbf{Object}(X)[x: Int,\ eq^{+}: X \rightarrow Bool,\ mv^{+}: Int \rightarrow X]$$

$$CPoint \triangleq \mathbf{Object}(X)[x: Int,\ c: Color,\ eq^{+}: X \rightarrow Bool,\ mv^{+}: Int \rightarrow X]$$

- These definitions freely use covariant and contravariant occurrences of Self types. The liberal treatment of Self types in O–3 yields *CPoint* <# *Point*.
- In O–1, the same definitions of *Point* and *CPoint* are valid, but they are less satisfactory because *CPoint* <: *Point* fails; therefore the O–1 definitions adopt a different type for *eq*.
- In O–2, contravariant occurrences of Self types are illegal; therefore the O–2 definitions have a different type for *eq*, too.

- We define two classes *pointClass* and *cPointClass* that correspond to the types *Point* and *CPoint*, respectively:

```
pointClass : Class(Point)  ≜
    class with (self: X<#Point)
        x = 0,
        eq = fun(other: X) self.x = other.x end,
        mv = fun(dx: Int) self.x := self.x+dx end
    end
cPointClass : Class(CPoint)  ≜
    subclass of pointClass: Class(Point)
    with (self: X<#CPoint)
        c = black
    override
        eq = fun(other: X) super.eq(other) and self.c = other.c end,
        mv = fun(dx: Int) super.mv(dx).c := red end
    end
```

- The subclass *cPointClass* could have inherited both *mv* and *eq*. However, we chose to override both of these methods in order to adapt them to deal with colors.
- In contrast with the corresponding programs in O–1 and O–2, no uses of typecase are required in this code. The use of typecase is not needed for accessing the color of a point after moving it. (Typecase is needed in O–1 but not in O–2.) Specifically, the overriding code for *mv* does not need a typecase on the result of **super**.*mv*(*dx*) in the definition of *cPointClass*.
- Other code that moves color points does not need a typecase either:

$$cPoint : CPoint \quad \triangleq \quad \mathbf{new}\ cPointClass$$

$$movedColor : Color \quad \triangleq \quad cPoint.mv(1).c$$

- Moreover, O–3 allows us to specialize the binary method *eq* as we have done in the definition of *cPointClass* (unlike O–2). This specialization does not require dynamic typing: we can write **super**.*eq*(*other*) without first doing a typecase on *other*.
- Thus the treatment of points in O–3 circumvents the previous needs for dynamic typing. The price for this is the loss of the subtyping *CPoint* <: *Point*, and hence the loss of subsumption between *CPoint* and *Point*.

Example: Binary Trees

```
Bin  ≜
    Object(X)[isLeaf: Bool, lft: X, rht: X, consLft: X→X, consRht: X→X]
binClass : Class(Bin)  ≜
    class with(self: X<#Bin)
        isLeaf = true,
        lft = self.lft,
        rht = self.rht,
        consLft = fun(lft: X) ((self.isLeaf := false).lft := lft). rht := self end,
        consRht = fun(rht: X) ((self.isLeaf := false).lft := self ). rht := rht end
    end
leaf : Bin  ≜
    new binClass
```

- The definition of the object type *Bin* is the same one we could have given in O–1, but it would be illegal in O–2 because of the contravariant occurrences of *X*.
- The method bodies rely on some new facts about typing; in particular, if *self* has type *X* and *X*<#*Bin*, then *self*.*lft* and *self*.*isLeaf*:=*false* have type *X*.

- Let us consider now a type *NatBin* of binary trees with natural number components.

$$NatBin \triangleq \mathbf{Object}(X)[n{:}\ Nat,\ isLeaf{:}\ Bool,\ lft{:}\ X,\ rht{:}\ X,\ consLft{:}\ X{\to}X,\ consRht{:}\ X{\to}X]$$

- We have *NatBin* <# *Bin*, although *NatBin* </: *Bin*.
- If *b* has type *Bin* and *nb* has type *NatBin*, then *b.consLft*(*b*) and *nb.consLft*(*nb*) are allowed, but *b.consLft*(*nb*) and *nb.consLft*(*b*) are not.
- The methods *consLft* and *consRht* can be used as binary operations on any pair of objects whose common type matches *Bin*. O–3 allows inheritance of *consLft* and *consRht*. A class for *NatBin* may inherit *consLft* and *consRht* from *binClass*.
- Because *NatBin* is not a subtype of *Bin*, generic operations must be explicitly parameterized over all types that match *Bin*. For example, we may write:

$$selfCons : \mathbf{All}(X{<\#}Bin)X{\to}X \triangleq$$
$$\mathbf{fun}(X{<\#}Bin)\ \mathbf{fun}(x{:}\ X)\ x.consLft(x)\ \mathbf{end\ end}$$

$$selfCons(NatBin)(nb) : NatBin \qquad \text{for } nb : NatBin$$

- Explicit parameterization must be used systematically in order to guarantee future flexibility in usage, especially for object types that contain binary methods.

Example: Cells

- In this version, the proper methods are indicated with variance annotations $^+$; the *contents* and *backup* attributes are fields.

$Cell \triangleq$

$\quad \mathbf{Object}(X)[contents: Nat,\ get^+: Nat,\ set^+: Nat \to X]$

$cellClass : \mathbf{Class}(Cell) \triangleq$

```
    class with(self: X<#Cell)
        contents = 0,
        get = self.contents,
        set = fun(n: Nat) self.contents := n end
    end
```

$ReCell \triangleq$

$\quad$ **Object**$(X)[contents\colon Nat,\ get^{+}\colon Nat,\ set^{+}\colon Nat \to X,\ backup\colon Nat,\ restore^{+}\colon X]$

$reCellClass$: **Class**$(ReCell) \triangleq$

```
    subclass of cellClass:Class(Cell)
    with(self: X<#ReCell)
        backup = 0,
        restore = self.contents := self.backup
    override
        set = fun(n: Nat) cellClass^set(X, self.backup := self.contents)(n) end
    end
```

- We can also write a version of *ReCell* that uses method update instead of a *backup* field:

```
ReCell'  ≜
    Object(X)[contents: Nat, get⁺: Nat, set⁺: Nat→X, restore: X]
reCellClass' : Class(ReCell')  ≜
    subclass of cellClass:Class(Cell)
    with(self: X<#ReCell')
        restore = self.contents := 0
    override
        set = fun(n: Nat)
                    let   m = self.contents
                    in    cellClass^set(X,
                                self.restore := method(y: X) y.contents := m end)
                          (n)
                    end
                end
    end
```

- We obtain *ReCell* <: *Cell* and *ReCell'* <: *Cell*, because of the covariance of *X* and the positive variance annotations on the method types of *Cell* where *X* occurs.
- On the other hand, we have also *ReCell* <# *Cell* and *ReCell'* <# *Cell*, and this does not depend on the variance annotations.
- A generic doubling function for all types that match *Cell* can be written as follows:

double : **All**(*X*<#*Cell*) *X*→*X* ≜
    fun(*X*<#*Cell*) **fun**(*x*: *X*) *x.set*(2**x.get*) **end end**

Rules for O–3

Judgments

$E \vdash \diamond$	environment E is well formed
$E \vdash A$	A is a well formed type in E
$E \vdash A :: Obj$	A is a well formed object type 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

(Env $\emptyset$)
$$\frac{}{\emptyset \vdash \diamond}$$

(Env X<:)
$$\frac{E \vdash A \qquad X \notin dom(E)}{E, X<:A \vdash \diamond}$$

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

(Env x)
$$\frac{E \vdash A \qquad x \notin dom(E)}{E, x{:}A \vdash \diamond}$$

Types

(Type Obj)

$$\frac{E \vdash A :: Obj}{E \vdash A}$$

(Type X)

$$\frac{E', X{<:}A, E'' \vdash \diamond}{E', X{<:}A, E'' \vdash X}$$

(Type Top)

$$\frac{E \vdash \diamond}{E \vdash \mathbf{Top}}$$

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

$$\frac{E, X{<\#}A \vdash B_i \qquad \forall i \in 1..n}{E \vdash \mathbf{Class}(A)}$$

(Type All<#)

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

Object Types

(Obj X)

$$\frac{E', X{<\#}A, E'' \vdash \diamond}{E', X{<\#}A, E'' \vdash X :: Obj}$$

(Obj Object) (l_i distinct, $\upsilon_i \in \{^{\circ}, ^{-}, ^{+}\}$)

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

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

Subtyping

(Sub Refl)

$$\frac{E \vdash A}{E \vdash A <: A}$$

(Sub Trans)

$$\frac{E \vdash A <: B \qquad E \vdash B <: C}{E \vdash A <: C}$$

(Sub X)

$$\frac{E', X <: A, E'' \vdash \diamond}{E', X <: A, E'' \vdash X <: A}$$

(Sub Top)

$$\frac{E \vdash A}{E \vdash A <: \mathbf{Top}}$$

(Sub Object)

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

(Sub All<#)

$$\frac{E \vdash A' \mathrel{<\#} A \qquad E, X \mathrel{<\#} A' \vdash B <: B'}{E \vdash \mathbf{All}(X \mathrel{<\#} A)B <: \mathbf{All}(X \mathrel{<\#} A')B'}$$

(Sub Invariant)

$$\frac{E \vdash B}{E \vdash {}^{o} B <: {}^{o} B}$$

(Sub Covariant)

$$\frac{E \vdash B <: B' \qquad \upsilon \in \{^{o},^{+}\}}{E \vdash \upsilon\, B <: {}^{+} B'}$$

(Sub Contravariant)

$$\frac{E \vdash B' <: B \qquad \upsilon \in \{^{o},^{-}\}}{E \vdash \upsilon\, B <: {}^{-} B'}$$

Matching

(Match Refl)

$$\frac{E \vdash A :: Obj}{E \vdash A \mathrel{<\#} A}$$

(Match Trans)

$$\frac{E \vdash A \mathrel{<\#} B \qquad E \vdash B \mathrel{<\#} C}{E \vdash A \mathrel{<\#} C}$$

(Match X)

$$\frac{E', X \mathrel{<\#} A, E'' \vdash \diamond}{E', X \mathrel{<\#} A, E'' \vdash X \mathrel{<\#} A}$$

(Match Object) (l_i distinct)

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

Terms

(Val Subsumption)

$$\frac{E \vdash a : A \qquad E \vdash A <: B}{E \vdash a : B}$$

(Val x)

$$\frac{E', x{:}A, E'' \vdash \diamond}{E', x{:}A, E'' \vdash x : A}$$

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

$$\frac{E, x{:}A \vdash b_i\{\!\{A\}\!\} : B_i\{\!\{A\}\!\} \qquad \forall i \in 1..n}{E \vdash \mathbf{object}(x{:}X{=}A)\ l_i{=}b_i\{X\}\ ^{i \in 1..n}\ \mathbf{end} : A}$$

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

$$\frac{E \vdash a : A' \qquad E \vdash A' <\# A \qquad \upsilon_j \in \{^{o}, ^{+}\} \qquad j \in 1..n}{E \vdash a.l_j : B_j\{\!\{A'\}\!\}}$$

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

$$\frac{E \vdash a : A' \qquad E \vdash A' <\# A \qquad E, X{<\#}A', x{:}X \vdash b : B_j \qquad \upsilon_j \in \{^{o}, ^{-}\} \qquad j \in 1..n}{E \vdash a.l_j := \mathbf{method}(x{:}X{<\#}A')b\ \mathbf{end} : A'}$$

(Val New)

$$\frac{E \vdash c : \mathbf{Class}(A)}{E \vdash \mathbf{new}\ c : A}$$

(Val Root)

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

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

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

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

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

(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<#)

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

Translation of O–3 (Sketch)

Syntax of $Ob_{\omega<:\mu}$

$K,L ::=$	kinds
Ty	types
$K \Rightarrow L$	operators from K to L
$A,B ::=$	constructors
X	constructor variable
Top	the biggest constructor at kind Ty
$[l_i \upsilon_i : B_i \ ^{i \in 1..n}]$	object type (l_i distinct, $\upsilon_i \in \{^o, ^-, ^+\}$)
$\forall(X<:A::K)B$	bounded universal type
$\mu(X)A$	recursive type
$\lambda(X::K)B$	operator
$B(A)$	operator application

$a,b ::=$	terms
x	variable
$[l_i{=}\varsigma(x_i{:}A_i)b_i\ ^{i\in 1..n}]$	object formation (l_i distinct)
$a.l$	method invocation
$a.l \Leftarrow \varsigma(x{:}A)b$	method update
$\lambda(X{<:}A{::}K)b$	constructor abstraction
$b(A)$	constructor application
$fold(A,a)$	recursive fold
$unfold(a)$	recursive unfold

- The symbol $\cong$ means "informally translates to", with $\cong_{Ty}$ for translations that yield types, and $\cong_{Op}$ for translations that yield operators.
- We represent the translation of a term a by $\underline{a}$, the type translation of a type A by $\underline{A}$, and its operator translation by $\underline{\underline{A}}$.
- We say that a variable X is subtype-bound when it is introduced as $X<:A$ for some A; we say that X is match-bound when it is introduced as $X<\#A$ for some A.

Translation summary

$X \quad \cong_{Op} \quad X$ (where X is match-bound in the environment)

$\mathbf{Object}(X)[l_i \upsilon_i{:}B_i \;^{i\in 1..n}] \quad \cong_{Op} \quad \lambda(X)[l_i \upsilon_i{:}\underline{B_i} \;^{i\in 1..n}]$

$X \quad \cong_{Ty} \quad X$ (when X is subtype-bound in the environment)

$X \quad \cong_{Ty} \quad X^*$ (when X is match-bound in the environment)

$\mathbf{Top} \quad \cong_{Ty} \quad Top$

$\mathbf{Object}(X)[l_i \upsilon_i{:}B_i \;^{i\in 1..n}] \quad \cong_{Ty} \quad (\lambda(X)[l_i \upsilon_i{:}\underline{B_i} \;^{i\in 1..n}])^*$

$$\textbf{Class}(A) \quad \cong_{Ty} \quad [new^{+}{:}\underline{A},\ l_i^{+}{:}\forall(X{\prec:}\underline{\underline{A}})X^{*}\rightarrow\underline{B_i}\ ^{i\in 1..n}]$$

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

$$\textbf{All}(X{<}\#A)B \quad \cong_{Ty} \quad \forall(X{\prec:}\underline{\underline{A}})\underline{B}$$

$$x \quad \cong \quad x$$

$$\textbf{object}(x{:}A)\ l_i{=}b_i\{x\}\ ^{i\in 1..n}\ \textbf{end} \quad \cong \quad fold(\underline{A},[l_i{=}\varsigma(x{:}\underline{\underline{A}}(\underline{A}))\underline{b_i}\{\!\{fold(\underline{A},x)\}\!\}\ ^{i\in 1..n}])$$

$$a.l_j \quad \cong \quad unfold(\underline{a}).l_j$$

$$a.l_j := \textbf{method}(x{:}A')b\{x\}\ \textbf{end} \quad \cong$$

$$fold(\underline{A}',unfold(\underline{a}).l_j\Leftarrow\varsigma(x{:}\underline{\underline{A}}'(\underline{A}'))\underline{b}\{\!\{\,fold(\underline{A}',x)\}\!\})$$

$$\textbf{new}\ c \quad \cong \quad \underline{c}.new$$

$$\textbf{root} \quad \cong \quad [new{=}\varsigma(z{:}[new^{+}{:}\mu(X)[]])fold(\mu(X)[],[])]$$

$$\textbf{subclass of}\ c'{:}C'\ \textbf{with}(x{:}X{<}\#A)\ l_i{=}b_i\ ^{i\in n+1..n+m}\ \textbf{override}\ l_i{=}b_i\ ^{i\in Ovr}\ \textbf{end} \quad \cong$$

$$[new{=}\varsigma(z{:}\underline{C})fold(\underline{A},[l_i{=}\varsigma(s{:}\underline{\underline{A}}(\underline{A}))z.l_i(\underline{\underline{A}})(\,fold(\underline{A},s))\ ^{i\in 1..n+m}])$$

$$l_i{=}\varsigma(z{:}\underline{C})\ \underline{c}'.l_i\ ^{i\in 1..n-Ovr},$$

$$l_i{=}\varsigma(z{:}\underline{C})\lambda(X{\prec:}\underline{\underline{A}})\lambda(x{:}X^{*})\underline{b_i}\ ^{i\in Ovr\cup n+1..n+m}]$$

$$\text{where } C \equiv \textbf{Class}(A)$$

$$c\hat{\ }l_j(A',a) \quad \cong \quad \underline{c}.l_j(\underline{\underline{A}}')(\underline{a})$$

$\textbf{fun}(X{<}\#A)b\ \textbf{end} \quad \cong \quad \lambda(X{\prec}{:}\underline{\underline{A}})\underline{b}$

$b(A') \quad \cong \quad \underline{b}(\underline{\underline{A}}')$

Summary

- 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:

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

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

- 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.

TRANSLATIONS

- In order to give insight into type rules for object-oriented languages, translations must be judgment-preserving (in particular, type and subtype preserving).
- Translating object-oriented languages directly to typed λ-calculi is just too hard. Object calculi provide a good stepping stone in this process, or an alternative endpoint.
- Translating object calculi into λ-calculi means, intuitively, "programming in object-oriented style within a procedural language". This is the hard part.

Untyped Translations

- Give insights into the nature of object-oriented computation.
- Objects = records of functions.

$$\begin{tikzcd}
& \text{o-o language} \arrow[dl] \arrow[dr] & \\
\varsigma\text{-calculus} \arrow[rr, shift left] & & \lambda\text{-calculus} \arrow[ll, shift left]
\end{tikzcd}$$

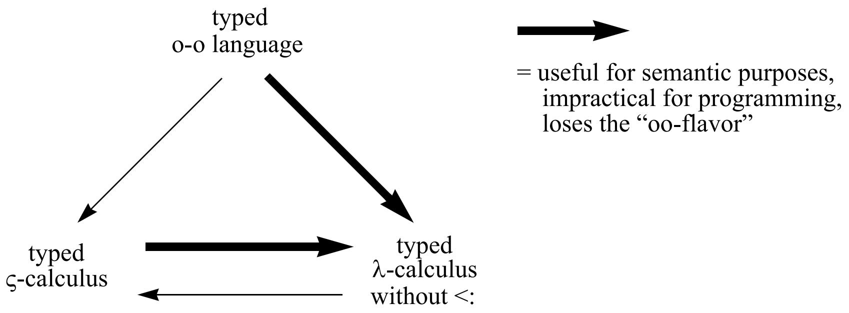
$\longrightarrow$ = easy translation

Type-Preserving Translations

- Give insights into the nature of object-oriented typing and subsumption/coercion.

- Object types = recursive records-of-functions types.

$$[l_i{:}B_i\ ^{i\in 1..n}] \quad \triangleq \quad \mu(X)\langle l_i{:}X\rightarrow B_i\ ^{i\in 1..n}\rangle$$

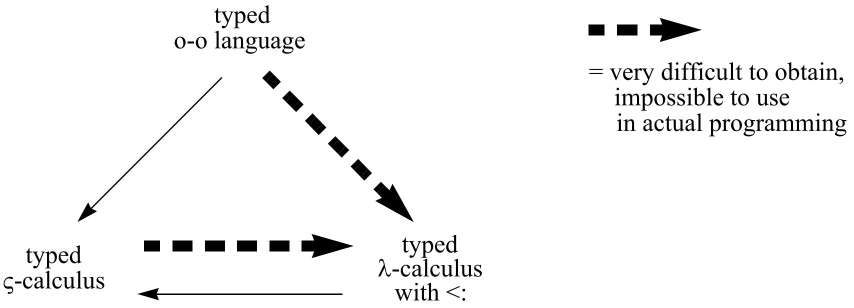


= useful for semantic purposes,
impractical for programming,
loses the “oo-flavor”

Subtype-Preserving Translations

- Give insights into the nature of subtyping for objects.
- Object types = recursive bounded existential types (!!).

$$[l_i{:}B_i\ ^{i\in 1..n}] \ \triangleq\ \mu(Y)\exists(X{<:}Y)\langle r{:}X,\ l_i^{sel}{:}X\to B_i\ ^{i\in 1..n},\ l_i^{upd}{:}(X\to B_i)\to X\ ^{i\in 1..n}\rangle$$


typed o-o language
typed ς-calculus
typed λ-calculus with <:
= very difficult to obtain, impossible to use in actual programming

CONCLUSIONS

- Foundations
  - ~ Subtype-preserving translations of object calculi into λ-calculi are hard.
  - ~ In contrast, subtype-preserving translations of λ-calculi into object-calculi can be easily obtained.
  - ~ In this sense, object calculi are a more convenient foundation for object-oriented programming than λ-calculi.

- Language design

  ~ Object calculi are a good basis for designing rich object-oriented type systems (including polymorphism, Self types, etc.).

  ~ Object-oriented languages can be shown sound by fairly direct translations into object calculi.

- Potential future areas
  - ~ Typed ς-calculi should be a good simple foundation for studying object-oriented specification and verification.
  - ~ They should also give us a formal platform for studying object-oriented concurrent languages (as opposed to “ordinary” concurrent languages).

References

- http://www.research.digital.com/SRC/personal/Luca_Cardelli/TheoryOfObjects.html
- M.Abadi, L.Cardelli: **A Theory of Objects**. Springer, 1996.

EXTRA LECTURE

- Operationally Sound Update (28 Slides).

Operationally Sound Update

Luca Cardelli

Digital Equipment Corporation

Systems Research Center

Outline

- The type rules necessary for “sufficiently polymorphic” update operations on records and objects are based on unusual operational assumptions.

- These update rules are sound operationally, but not denotationally (in standard models). They arise naturally in type systems for programming, and are not easily avoidable.

- Thus, we have a situation where operational semantics is clearly more advantageous than denotational semantics.

- However (to please the semanticists) I will show how these operationally-based type systems can be translated into type systems that are denotationally sound.

The polymorphic update problem

L.Cardelli, P.Wegner

"*The need for bounded quantification arises very frequently in object-oriented programming. Suppose we have the following types and functions:*

type $Point = [x: Int, y: Int]$
value $moveX_0 = \lambda(p: Point, dx: Int)\ p.x := p.x + dx;\ p$
value $moveX = \lambda(P <: Point)\ \lambda(p: P, dx: Int)\ p.x := p.x + dx;\ p$

It is typical in (type-free) object-oriented programming to reuse functions like moveX on objects whose type was not known when moveX was defined. If we now define:

type $Tile = [x: Int, y: Int, hor: Int, ver: Int]$

we may want to use moveX to move tiles, not just points."

$Tile <: Point$

$moveX_0([x=0, y=0, hor=1, ver=1], 1).hor$ fails

$moveX(Tile)([x=0, y=0, hor=1, ver=1], 1).hor$ succeeds

- In that paper, bounded quantification was justified as a way of handling polymorphic update, and was used in the context of *imperative* update.

- The examples were inspired by object-oriented applications. Object-oriented languages combine subtyping and polymorphism with state encapsulation, and hence with imperative update. Some form of polymorphic update is inevitable.

- Simplifying the situation a bit, let's consider the equivalent example in a functional setting. We might hope to achieve the following typing:

$$\begin{aligned} bump &\triangleq \lambda(P <: Point)\, \lambda(p{:}\, P)\, p.x := p.x + 1 \\ bump &: \forall(P <: Point)\, P \rightarrow P \end{aligned}$$

But ...

There is no bump there!

Neither semantically

J.Mitchell

In standard models, the type $\forall(P<:Point)P\to P$ contains only the identity function.

Consider $\{p\}$ for any $p\in Point$. If $f : \forall(P<:Point)P\to P$, then $f(\{p\}) : \{p\}\to\{p\}$, therefore f must map every point to itself, and must be the identity.

Nor parametrically

M.Abadi, L.Cardelli, G.Plotkin

By parametricity (for bounded quantifiers), we can show that if $f : \forall(P<:Point)P\to P$, then $\forall(P<:Point)\ \forall(x:P)\ f(P)(x) =_P x$. Thus f is an identity.

Nor by standard typing rules

As shown next ...

The simple rule for update

(Val Simple Update)

$$\frac{E \vdash a : [l_i{:}B_i \ ^{i\in 1..n}] \qquad E \vdash b : B_j \qquad j\in 1..n}{E \vdash a.l_j{:=}b : [l_i{:}B_i \ ^{i\in 1..n}]}$$

- According to this rule, bump does not typecheck as desired:

  $$bump \ \triangleq \ \lambda(P <: Point)\, \lambda(p{:}\, P)\, p.x := p.x + 1$$

  We must go from $p{:}P$ to $p{:}Point$ by subsumption before we can apply the rule. Therefore we obtain only:

  $$bump : \forall(P <: Point)\, P \rightarrow Point$$

The "structural" rule for update

(Val Structural Update)

$$\frac{E \vdash a : A \qquad E \vdash A <: [l_i{:}B_i{}^{i \in 1..n}] \qquad E \vdash b : B_j \qquad j \in 1..n}{E \vdash a.l_j{:=}b : A}$$

- According to this rule, bump typechecks as desired, using the special case where A is a type variable.

$$bump \triangleq \lambda(P <: Point)\, \lambda(p{:}\, P)\, p.x := p.x + 1$$

$$bump : \forall(P <: Point)\, P \to P$$

- Therefore, (Val Structural Update) is not sound in most semantic models, because it populates the type $\forall(P{<:}Point)P{\to}P$ with a non-identity function.

- However, (Val Structural Update) is in practice highly desirable, so the interesting question is under which conditions it is sound.

Can’t allow too many subtypes

- Suppose we had:

$$BoundedPoint \triangleq \{x: 0..9, y: 0..9\}$$
$$BoundedPoint <: Point$$

  then:

$$bump(BoundedPoint)(\{x=9, y=9\}) : BoundedPoint$$

  unsound!

- To recover from this problem, the subtyping rule for records/objects must forbid certain subtypings:

(Sub Object)

$$\frac{E \vdash B_i \qquad \forall i \in 1..m}{E \vdash [l_i{:}B_i \;^{i\in 1..n+m}] <: [l_i{:}B_i \;^{i\in 1..n}]}$$

- Therefore, for soundness, the rule for structural updates makes implicit assumptions about the subtype relationships that may exist.

Relevant rules for structural update

(Sub Object)

$$\frac{E \vdash B_i \quad \forall i \in 1..m}{E \vdash [l_i{:}B_i{}^{i\in 1..n+m}] <: [l_i{:}B_i{}^{i\in 1..n}]}$$

(Val Subsumption)

$$\frac{E \vdash a : A \quad E \vdash A <: B}{E \vdash a : B}$$

(Val Object)

$$\frac{E \vdash b_i : B_i \quad \forall i \in 1..n}{E \vdash [l_i{=}b_i{}^{i\in 1..n}] : [l_i{:}B_i{}^{i\in 1..n}]}$$

(Val Structural Update)

$$\frac{E \vdash a : A \quad E \vdash A <: [l_i{:}B_i{}^{i\in 1..n}] \quad E \vdash b : B_j \quad j \in 1..n}{E \vdash a.l_j{:=}b : A}$$

(Red Update)

$$\frac{\vdash a \rightsquigarrow [l_i{=}v_i{}^{i\in 1..n}] \quad \vdash b \rightsquigarrow v \quad j \in 1..n}{\vdash a.l_j{:=}b \rightsquigarrow [l_j{=}v,\ l_i{=}v_i{}^{i\in 1..n-\{j\}}]}$$

The structural subtyping lemmas

Lemma (Structural subtyping)

If $E \vdash [l_i{:}B_i\ ^{i \in I}]{<:}C$ then either $C \equiv Top$, or $C \equiv [l_i{:}B_i\ ^{i \in J}]$ with $J \subseteq I$.

If $E \vdash C{<:}[l_i{:}B_i\ ^{i \in J}]$ then either $C \equiv [l_i{:}B_i\ ^{i \in I}]$ with $J \subseteq I$,

or $C \equiv X_1$ and E contains a chain $X_1 <: \ldots <: X_p <: [l_i{:}B_i\ ^{i \in I}]$ with $J \subseteq I$.

Proof

By induction on the derivations of $E \vdash [l_i{:}B_i\ ^{i \in I}]{<:}C$ and $E \vdash C{<:}[l_i{:}B_i\ ^{i \in I}]$.

□

Soundness by subject reduction

Theorem (Subject reduction)

If $\varnothing \vdash a : A$ and $\vdash a \leadsto v$ then $\varnothing \vdash v : A$.

Proof By induction on the derivation of $\vdash a \leadsto v$.

Case (Red Update)

$$\frac{\vdash c \leadsto [l_i{=}z_i\ {}^{i\in 1..n}] \qquad \vdash b \leadsto w \qquad j\in 1..n}{\vdash c.l_j{:=}b \leadsto [l_j{=}w,\ l_i{=}z_i\ {}^{i\in 1..n-\{j\}}]}$$

By hypothesis $\varnothing \vdash c.l_j{:=}b : A$. This must have come from (1) an application of (Val Structural Update) with assumptions $\varnothing \vdash c : C$, and $\varnothing \vdash C <: D$ where $D \equiv [l_j{:}B_j, ...]$, and $\varnothing \vdash b : B_j$, and with conclusion $\varnothing \vdash c.l_j{:=}b : C$, followed by (2) a number of subsumption steps implying $\varnothing \vdash C <: A$ by transitivity.

By induction hypothesis, since $\varnothing \vdash c : C$ and $\vdash c \leadsto z \equiv [l_i{=}z_i\ {}^{i\in 1..n}]$, we have $\varnothing \vdash z : C$.

By induction hypothesis, since $\varnothing \vdash b : B_j$ and $\vdash b \leadsto w$, we have $\varnothing \vdash w : B_j$.

Now, $\varnothing \vdash z : C$ must have come from (1) an application of (Val Object) with assumptions $\varnothing \vdash z_i : B_i'$ and $C' \equiv [l_i'{:}B_i'\ {}^{i\in 1..n}]$, and with conclusion $\varnothing \vdash z : C'$, followed by (2) a number of subsumption steps implying $\varnothing \vdash C' <: C$ by transitivity. By transitivity, $\varnothing \vdash C' <: D$. Hence by the Structural Subtyping Lemma, we must have $B_j \equiv B_j'$. Thus $\varnothing \vdash w : B_j'$. Then, by (Val Object), we obtain $\varnothing \vdash [l_j{=}w,\ l_i{=}z_i\ ^{i\in 1..n-\{j\}}] : C'$. Since $\varnothing \vdash C' <: A$ by transitivity, we have $\varnothing \vdash [l_j{=}w,\ l_i{=}z_i\ ^{i\in 1..n-\{j\}}] : A$ by subsumption.

Other structural rules

- Rules based on structural assumptions (structural rules, for short) are not restricted to record/ object update. They also arise in:
  - ~ method invocation with Self types,
  - ~ object cloning,
  - ~ class encodings,
  - ~ unfolding recursive types.

- The following is one of the simplest examples of the phenomenon (although not very useful in itself):

A structural rule for product types

M.Abadi

- The following rule for pairing enables us to mix two pairs a and b of type C into a new pair of the same type. The only assumption on C is that it is a subtype of a product type $B_1 \times B_2$.

$$\frac{E \vdash C <: B_1 \times B_2 \qquad E \vdash a : C \qquad E \vdash b : C}{E \vdash \langle fst(a), snd(b) \rangle : C}$$

  The soundness of this rule depends on the property that every subtype of a product type $B_1 \times B_2$ is itself a product type $C_1 \times C_2$.

- This property is true operationally for particular systems, but fails in any semantic model where subtyping is interpreted as the subset relation. Such a model would allow the set $\{a,b\}$ as a subtype of $B_1 \times B_2$ whenever a and b are elements of $B_1 \times B_2$. If a and b are different, then $\langle fst(a), snd(b) \rangle$ is not an element of $\{a,b\}$. Note that $\{a,b\}$ is not a product type.

A structural rule for recursive types

M.Abadi, L.Cardelli, R.Viswanathan

- In the paper "An Interpretation of Objects and Object types" we give a translation of object types into ordinary types:

$$[l_i{:}B_i{}^{i\in 1..n}] \;\triangleq\; \mu(Y)\,\exists(X<:Y)\,\langle r{:}X,\ l_i^{sel}{:}X\to B_i{}^{i\in 1..n},\ l_i^{upd}{:}(X\to B_i)\to X^{\,i\in 1..n}\rangle$$

  this works fine for non-structural rules.

- In order to validate a structural update rule in the source calculus, we need a structural update rule in the target calculus. It turns out that the necessary rule is the following, which is operationally sound:

$$\frac{E \vdash C <: \mu(X)B\{X\} \qquad E \vdash a : C}{E \vdash unfold(a) : B\{\!\{C\}\!\}}$$

A structural rule for method invocation

- In the context of object types with Self types:

(Val Select)

$$\frac{E \vdash a : A \qquad E \vdash A <: Obj(X)[l_i{:}B_i\{X\}^{\ i\in 1..n}] \qquad j\in 1..n}{E \vdash a.l_j : B_j\{\!\{A\}\!\}}$$

This structural rule is necessary to “encapsulate” structural update inside methods:

$A \;\triangleq\; Obj(X)[n{:}\ Int,\ bump{:}\ X]$

$\lambda(Y <: A)\ \lambda(y{:}\ Y)\ y.bump$

$:\ \ \forall(Y <: A)\ Y \to Y$

Structural rules and class encodings

Types of the form $\forall(X<:A)X\rightarrow B\{X\}$ are needed also for defining classes as collections of pre-methods. Each pre-method must work for all possible subclasses, parametrically in self, so that it can be inherited.

$$A \triangleq Obj(X)[l_i{:}B_i\{X\}^{\ i\in 1..n}]$$

$$Class(A) \triangleq [new{:}\ A,\ l_i{:}\ \forall(X<:A)X\rightarrow B_i\{X\}^{\ i\in 1..n}]$$

$$Bump \triangleq Obj(X)[n{:}\ Int,\ bump{:}\ X]$$

$$Class(Bump) \triangleq [new{:}\ Bump,\ bump{:}\ \forall(X<:Bump)X\rightarrow X]$$

$$c : Class(Bump) \triangleq$$

$$[new = \varsigma(c{:}\ Class(Bump))\ [n = 0,\ bump = \varsigma(s{:}\ Bump)\ c.bump(Bump)(s)],$$
$$bump = \lambda(X<:Bump)\ \lambda(x{:}X)\ x.n{:=}x.n{+}1\}]$$

A structural rule for cloning

- In the context of imperative object calculi:

(Val Clone)

$$\frac{E \vdash a : A \qquad E \vdash A <: [l_i{:}B_i \ ^{i \in 1..n}] \qquad j \in 1..n}{E \vdash clone(a) : A}$$

This structural rule is necessary for bumping and returning a clone instead of the original object:

$$bump \quad \triangleq \quad \lambda(P <: Point)\ \lambda(p{:}\ P)\ clone(p).x := p.x + 1$$

$$bump : \forall(P <: Point)\ P \to P$$

Comments

- Structural rules are quite satisfactory. The operational semantics is the right one, the typing rules are the right ones for writing useful programs, and the rules are sound for the semantics.

- We do not have a denotational semantics (yet?). (The paper "Operations on Records" by L.Cardelli and J.Mitchell contains a limited model for structural update; no general models seems to be known.)

- Even without a denotational semantcs, there is an operational semantics from which one could, hopefully, derive a theory of typed equality.

- Still, I would like to understand in what way a type like $\forall(X<:\text{Point})X\rightarrow X$ does not mean what most people in this room might think.

- Insight may come from translating a calculus with structural rules, into one without structural rules for which we have a standard semantics.

Translating away structural rules

- The "Penn translation" can be used to map $F_{<:}$ into F by threading *coercion* functions.
- Similarly, we can map an $F_{<:}$-like calculus with structural rules into a normal $F_{<:}$-like calculus by threading *update* functions (*c.f.* M.Hofmann and B.Pierce: Positive <:).
- Example :

$$f : \forall(X <: [l{:}\ Int])\ X \to X \ \triangleq$$
$$\lambda(X <: [l{:}\ Int])\ \lambda(x{:}\ X)\ x.l := 3$$

$$f([l{:}\ Int])$$

(N.B. the update $x.l{:=}3$ uses the structural rule)

translates to:

$$f : \forall(X <: [l{:}\ Int])\ [l{:}\ X \to Int \to X] \to X \to X \ \triangleq$$
$$\lambda(X <: [l{:}\ Int])\ \lambda(\pi_X{:}\ [l{:}\ X \to Int \to X\ ])\ \lambda(x{:}\ X)\ \pi_X.l(x)(3)$$

$$f([l{:}\ Int])\ ([l = \lambda(x{:}\ [l{:}\ Int])\ \lambda(y{:}\ Int)\ x.l := y])$$

(N.B. the update $x.l{:=}$y uses the non-structural rule)

- Next I discuss a simplified, somewhat ad-hoc, calculus to formalize the main ideas for this translation.

Syntax

$A,B ::=$	types
X	type variable
$[l_i{:}B_i\ ^{i\in 1..n}]$	object type (l_i distinct)
$A\rightarrow B$	function types
$\forall(X{<:}[l_i{:}B_i\ ^{i\in 1..n}])B$	bounded universal type
$a,b ::=$	terms
x	variable
$[l_i{=}\varsigma(x_i{:}A_i)b_i\ ^{i\in 1..n}]$	object (l_i distinct)
$a.l$	method invocation
$a.l\Leftarrow\varsigma(x{:}A)b$	method update
$\lambda(x{:}A)b$	function
$b(a)$	application
$\lambda(X{<:}[l_i{:}B_i\ ^{i\in 1..n}])b$	polymorphic function
$b(A)$	polymorphic instantiation

- We consider method update instead of field update ($a_A.l{:=}b \ \triangleq\ a.l\Leftarrow\varsigma(x{:}A)b$).
- We do not consider object types with Self types.
- We do not consider arbitrary bounds for type variables, only object-type bounds.

Environments

(Env $\emptyset$)

$$\frac{}{\emptyset \vdash \diamond}$$

(Env x)

$$\frac{E \vdash A \qquad x \notin dom(E)}{E, x{:}A \vdash \diamond}$$

(Env X<:) (where $A \equiv [l_i{:}B_i \,^{i \in 1..n}]$)

$$\frac{E \vdash A \qquad X \notin dom(E)}{E, X{<:}A \vdash \diamond}$$

Types

(Type X<:)

$$\frac{E', X{<:}A, E'' \vdash \diamond}{E', X{<:}A, E'' \vdash X}$$

(Type Object) (l_i distinct)

$$\frac{E \vdash B_i \qquad \forall i \in 1..n}{E \vdash [l_i{:}B_i \,^{i \in 1..n}]}$$

(Type Arrow)

$$\frac{E \vdash A \qquad E \vdash B}{E \vdash A \to B}$$

(Type All<:)

$$\frac{E, X{<:}A \vdash B}{E \vdash \forall(X{<:}A)B}$$

Subtyping

(Sub Refl)

$$\frac{E \vdash A}{E \vdash A <: A}$$

(Sub Trans)

$$\frac{E \vdash A <: B \qquad E \vdash B <: C}{E \vdash A <: C}$$

(Sub X)

$$\frac{E', X{<:}A, E'' \vdash \diamond}{E', X{<:}A, E'' \vdash X{<:}A}$$

(Sub Object) (l_i distinct)

$$\frac{E \vdash B_i \qquad \forall i{\in}1..n{+}m}{E \vdash [l_i{:}B_i{}^{i\in 1..n+m}] <: [l_i{:}B_i{}^{i\in 1..n}]}$$

(Sub Arrow)

$$\frac{E \vdash A' <: A \qquad E \vdash B <: B'}{E \vdash A{\rightarrow}B <: A'{\rightarrow}B'}$$

(Sub All)

$$\frac{E \vdash A' <: A \qquad E, X{<:}A' \vdash B <: B'}{E \vdash \forall(X{<:}A)B <: \forall(X{<:}A')B'}$$

Typing

(Val Subsumption)

$$\frac{E \vdash a : A \qquad E \vdash A <: B}{E \vdash a : B}$$

(Val x)

$$\frac{E', x{:}A, E'' \vdash \diamond}{E', x{:}A, E'' \vdash x{:}A}$$

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

$$\frac{E, x_i{:}A \vdash b_i : B_i \qquad \forall i \in 1..n}{E \vdash [l_i{=}\varsigma(x_i{:}A)b_i\ ^{i \in 1..n}] : A}$$

(Val Select)

$$\frac{E \vdash a : [l_i{:}B_i\ ^{i \in 1..n}] \qquad j \in 1..n}{E \vdash a.l_j : B_j}$$

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

$$\frac{E \vdash a : A \qquad E, x{:}A \vdash b : B_j \qquad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x{:}A)b : A}$$

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

$$\frac{E \vdash a : X \qquad E \vdash X <: A \qquad E, x{:}X \vdash b : B_j \qquad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x{:}X)b : X}$$

(Val Fun)

$$\frac{E, x{:}A \vdash b : B}{E \vdash \lambda(x{:}A)b : A \rightarrow B}$$

(Val Appl)

$$\frac{E \vdash b : A \rightarrow B \qquad E \vdash a : A}{E \vdash b(a) : B}$$

(Val Fun2<:)

$$\frac{E, X <: A \vdash b : B}{E \vdash \lambda(X <: A)b : \forall(X <: A)B}$$

(Val Appl2<:) (where $A' \equiv [l_i{:}B_i\ ^{i \in 1..n}]$ or $A' \equiv Y$)

$$\frac{E \vdash b : \forall(X <: A)B\{X\} \qquad E \vdash A' <: A}{E \vdash b(A') : B\{\!\{A'\}\!\}}$$

- The source system for the translation is the one given above. The target system is the one given above minus the (Val Update *X*) rule.
- Derivations in the source system can be translated to derivations that do not use (Val Update *X*). The following tables give a slightly informal summary of the translation on derivations.

Translation of Environments

$$\langle\!\langle \phi \rangle\!\rangle \triangleq \phi$$

$$\langle\!\langle E, x{:}A \rangle\!\rangle \triangleq \langle\!\langle E \rangle\!\rangle, x{:}\langle\!\langle A \rangle\!\rangle$$

$$\langle\!\langle E, X{<:}[l_i{:}B_i\ ^{i\in 1..n}] \rangle\!\rangle \triangleq \langle\!\langle E \rangle\!\rangle, X{<:}\langle\!\langle [l_i{:}B_i\ ^{i\in 1..n}] \rangle\!\rangle, \pi_X{:}[l_i{:}X\rightarrow(X\rightarrow\langle\!\langle B_i \rangle\!\rangle)\rightarrow X\ ^{i\in 1..n}]$$

where each l_i: $X\rightarrow(X\rightarrow\langle\!\langle B_i \rangle\!\rangle)\rightarrow X$ is an updator that takes an object of type X, takes a pre-method for X (of type $X\rightarrow\langle\!\langle B_i \rangle\!\rangle$), updates the i-th method of the object, and returns the modified object of type X.

Translation of Types

$$\langle\!\langle X \rangle\!\rangle \triangleq X$$

$$\langle\!\langle [l_i{:}B_i\ ^{i\in 1..n}] \rangle\!\rangle \triangleq [l_i{:}\langle\!\langle B_i \rangle\!\rangle\ ^{i\in 1..n}]$$

$$\langle\!\langle A\to B \rangle\!\rangle \triangleq \langle\!\langle A \rangle\!\rangle\to\langle\!\langle B \rangle\!\rangle$$

$$\langle\!\langle \forall(X{<:}[l_i{:}B_i\ ^{i\in 1..n}])B \rangle\!\rangle \triangleq \forall(X{<:}\langle\!\langle [l_i{:}B_i\ ^{i\in 1..n}] \rangle\!\rangle)[l_i{:}X\to(X\to\langle\!\langle B_i \rangle\!\rangle)\to X\ ^{i\in 1..n}]\to\langle\!\langle B \rangle\!\rangle$$

- N.B. the translation preserves subtyping. In particular:

$$\langle\!\langle \forall(X{<:}[l_i{:}B_i\ ^{i\in 1..n}])B \rangle\!\rangle \ <: \ \langle\!\langle \forall(X{<:}[l_i{:}B_i\ ^{i\in 1..n+m}])B \rangle\!\rangle$$

since:

$$\forall(X{<:}\langle\!\langle [l_i{:}B_i\ ^{i\in 1..n}] \rangle\!\rangle)\ [l_i{:}X\to(X\to\langle\!\langle B_i \rangle\!\rangle)\to X\ ^{i\in 1..n}]\to\langle\!\langle B \rangle\!\rangle \ <:$$
$$\forall(X{<:}\langle\!\langle [l_i{:}B_i\ ^{i\in 1..n+m}] \rangle\!\rangle)\ [l_i{:}X\to(X\to\langle\!\langle B_i \rangle\!\rangle)\to X\ ^{i\in 1..n+m}]\to\langle\!\langle B \rangle\!\rangle$$

- We have a calculus with polymorphic update where quantifier and arrow types are contravariant on the left (*c.f.* Positive Subtyping).

Translation of Terms

$\langle\!\langle x \rangle\!\rangle \triangleq x$

$\langle\!\langle [l_i{=}(x_i{:}A_i)b_i \ ^{i\in 1..n}] \rangle\!\rangle \triangleq [l_i{=}\varsigma(x_i{:}\langle\!\langle A_i \rangle\!\rangle)\langle\!\langle b_i \rangle\!\rangle \ ^{i\in 1..n}]$

$\langle\!\langle a.l_j \rangle\!\rangle \triangleq \langle\!\langle a \rangle\!\rangle.l_j$

$\langle\!\langle a.l{\Leftarrow}\varsigma(x{:}A)b \rangle\!\rangle \triangleq \langle\!\langle a \rangle\!\rangle.l{\Leftarrow}(x{:}\langle\!\langle A \rangle\!\rangle)\langle\!\langle b \rangle\!\rangle)$ for (Val Update Obj)

$\langle\!\langle a.l{\Leftarrow}\varsigma(x{:}X)b \rangle\!\rangle \triangleq \pi_X.l(\langle\!\langle a \rangle\!\rangle)(\lambda(x{:}X)\langle\!\langle b \rangle\!\rangle)$ for (Val Update X)

$\langle\!\langle \lambda(x{:}\mathrm{A})b \rangle\!\rangle \triangleq \lambda(x{:}\langle\!\langle A \rangle\!\rangle)\langle\!\langle b \rangle\!\rangle$

$\langle\!\langle b(a) \rangle\!\rangle \triangleq \langle\!\langle b \rangle\!\rangle(\langle\!\langle a \rangle\!\rangle)$

$\langle\!\langle \lambda(X{<:}[l_i{:}B_i \ ^{i\in 1..n}])b \rangle\!\rangle \triangleq$

$\quad \lambda(X{<:}\langle\!\langle [l_i{:}B_i \ ^{i\in 1..n}] \rangle\!\rangle)\ \lambda(\pi_X{:}[l_i{:}X{\to}(X{\to}\langle\!\langle B_i \rangle\!\rangle){\to}X \ ^{i\in 1..n}])\ \langle\!\langle b \rangle\!\rangle$

$\langle\!\langle b(A) \rangle\!\rangle \triangleq$ for $A = [l_i{:}B_i \ ^{i\in 1..n}]$

$\quad \langle\!\langle b \rangle\!\rangle(\langle\!\langle A \rangle\!\rangle)\ ([l_i = \lambda(x_i{:}\langle\!\langle A \rangle\!\rangle)\ \lambda(f{:}\langle\!\langle A \rangle\!\rangle{\to}\langle\!\langle B_i \rangle\!\rangle)\ x.l_i{\Leftarrow}\varsigma(z{:}\langle\!\langle A \rangle\!\rangle)f(z) \ ^{i\in 1..n}])$

$\langle\!\langle b(Y) \rangle\!\rangle \triangleq \langle\!\langle b \rangle\!\rangle(Y)(\pi_Y)$

Conclusions

- Structural rules for polymorphic update are sound for operational semantics. They work equally well for functional and imperative semantics.

- Structural rules can be translated into non structural rules. I have shown a translation for a restricted form of quantification.

- Theories of equality for systems with structural rules have not been studied directly yet. Similarly, theories of equality induced by the translation have not been studied.

Extra Slides

Unsoundness of Naive Object Subtyping with Binary Methods

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

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

Consider:

$m : \text{Max} \triangleq [n = 0, \text{max} = \dots]$

$mm : \text{MinMax} \triangleq$

$\quad [n = 0, \text{min} = \dots,$

$\quad \text{max} = \varsigma(s:\text{MinMax})\ \lambda(o:\text{MinMax})$

$\qquad \text{if } o.\text{min}(o).n > s.n \text{ then } o \text{ else } s]$

Assume MinMax <: Max, then:

$mm : \text{Max}$ (by subsumption)

$mm.\text{max}(m) : \text{Max}$

But (Eiffel, O_2, ...):

$mm.\text{max}(m) \leadsto \text{if } m.\text{min}(m).n > mm.n \text{ then } m \text{ else } mm \leadsto \text{CRASH!}$

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.

$U \triangleq []$	The unit object type.
$L \triangleq [l{:}U]$	An object type with just l.
$L <: U$	

$P \triangleq [x{:}U, f{:}U]$

$Q \triangleq [x{:}L, f{:}U]$

Assume $Q <: P$ by an (erroneous) covariant rule for object subtyping

$q : Q \triangleq [x = [l{=}[]], f = \varsigma(s{:}Q)\, s.x.l]$

then	$q : P$	by subsumption with $Q <: P$
hence	$q.x{:=}[] : P$	that is $[x = [], f = \varsigma(s{:}Q)\, s.x.l] : P$
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 $A \equiv [l_i{:}B_i\ ^{i\in 1..n}]$)

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

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

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

$P \triangleq [f{:}[]]$

$Q \triangleq [f{:}[], y{:}[]]$ $\qquad Q <: P$

$p : P \triangleq [f{=}[]]$

$q : Q \triangleq [f{=}\varsigma(s{:}Q)s.y, y{=}[]]$

then $\quad q : P$ $\qquad$ by subsumption with $Q <: P$

hence $\quad q\cdot f : P\to[]$ $\qquad$ that is $\lambda(s{:}Q)s.y : P\to[]$

But $\quad q.f(p)$ $\qquad$ 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!

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:

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

- The unfolding property of recursive types yields:

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

- 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:

$$\begin{aligned}
D\{X\} &\triangleq \mu(Y)\ X\to Y \\
A &\triangleq \mu(X)\ D\{X\} \\
D\{A\} &\equiv \mu(Y)\ A\to Y && = A \\
A_{Op} &\equiv \lambda(X)\ D\{X\} \\
D\{A\}_{Op} &\equiv \lambda(Y)\ A\to Y && ¦\ A_{Op}
\end{aligned}$$

- Thus, we may have two types A and B such that $A = B$ but $A_{Op}\ ¦\ B_{Op}$ (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 $A <: [n:Int, max^{+}:A\rightarrow A]$ often has the shape $\mu(Y)[n:Int, max^{+}:Y\rightarrow Y, ...]$.

- Both Max and MinMax are pre-fixpoints of $\mathrm{Max}_{\mathrm{Op}}$:

$$\begin{array}{ll}
\mathrm{Max} <: \mathrm{Max}_{\mathrm{Op}}(\mathrm{Max}) & (= \mathrm{Max}) \\
\mathrm{MinMax} & (= [n{:}\mathrm{Int},\ \mathrm{max}^{+}{:}\mathrm{MinMax}\rightarrow\mathrm{MinMax},\ \mathrm{min}^{+}{:}\ \ldots\ ]) \\
\quad <: \mathrm{Max}_{\mathrm{Op}}(\mathrm{MinMax}) & (= [n{:}\mathrm{Int},\ \mathrm{max}^{+}{:}\mathrm{MinMax}\rightarrow\mathrm{MinMax}])
\end{array}$$

So, we can parameterize over all types X with the property that $X <: \mathrm{Max}_{\mathrm{Op}}(X)$.

$$\forall(X<:\mathrm{Max}_{\mathrm{Op}}(X))B\{X\}$$

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

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

$$\begin{array}{l}
\text{pre-max}(\mathrm{Max}) : \mathrm{Max}\rightarrow\mathrm{Max}\rightarrow\mathrm{Max} \\
\text{pre-max}(\mathrm{MinMax}) : \mathrm{MinMax}\rightarrow\mathrm{MinMax}\rightarrow\mathrm{MinMax}
\end{array}$$

The F-bounded Interpretation

- The central idea of the interpretation is:

$$A <\# B \quad \approx \quad A <: B_{Op}(A)$$
$$\forall(X<\#A)B\{X\} \quad \approx \quad \forall(X<:A_{Op}(X))B\{X\}$$

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

$$\forall(X<\#A)\ \forall(Y<\#X)\ ... \quad \approx \quad ?$$

  Since $\forall(X<:A_{Op}(X))\ \forall(Y<:X_{Op}(Y))$ ... 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:

$$A <\# A \qquad \approx \qquad A <: A_{Op}(A)$$

  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:

$$A <: B_{Op}(A) \quad \text{and} \quad B <: C_{Op}(B) \quad \text{imply} \quad A <: C_{Op}(A)$$

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

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

$$\begin{aligned} A &\triangleq \mu(X)[p^{+}: X\rightarrow Int,\ q: Int] \\ B &\triangleq \mu(X)[p^{+}: X\rightarrow Int] \\ C &\triangleq \mu(X)[p^{+}: B\rightarrow Int] \end{aligned}$$

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

$$\begin{aligned} A &= [p^{+}: A\rightarrow Int,\ q: Int] && <: \\ B_{Op}(A) &= [p^{+}: A\rightarrow Int] \end{aligned}$$

$$\begin{aligned} B &= [p^{+}: B\rightarrow Int] && <: \\ C_{Op}(B) &= [p^{+}: B\rightarrow Int] \end{aligned}$$

$$\begin{aligned} A &= [p^{+}: A\rightarrow Int,\ q: Int] && \not<: \\ C_{Op}(A) &= [p^{+}: B\rightarrow Int] \end{aligned}$$

- 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.