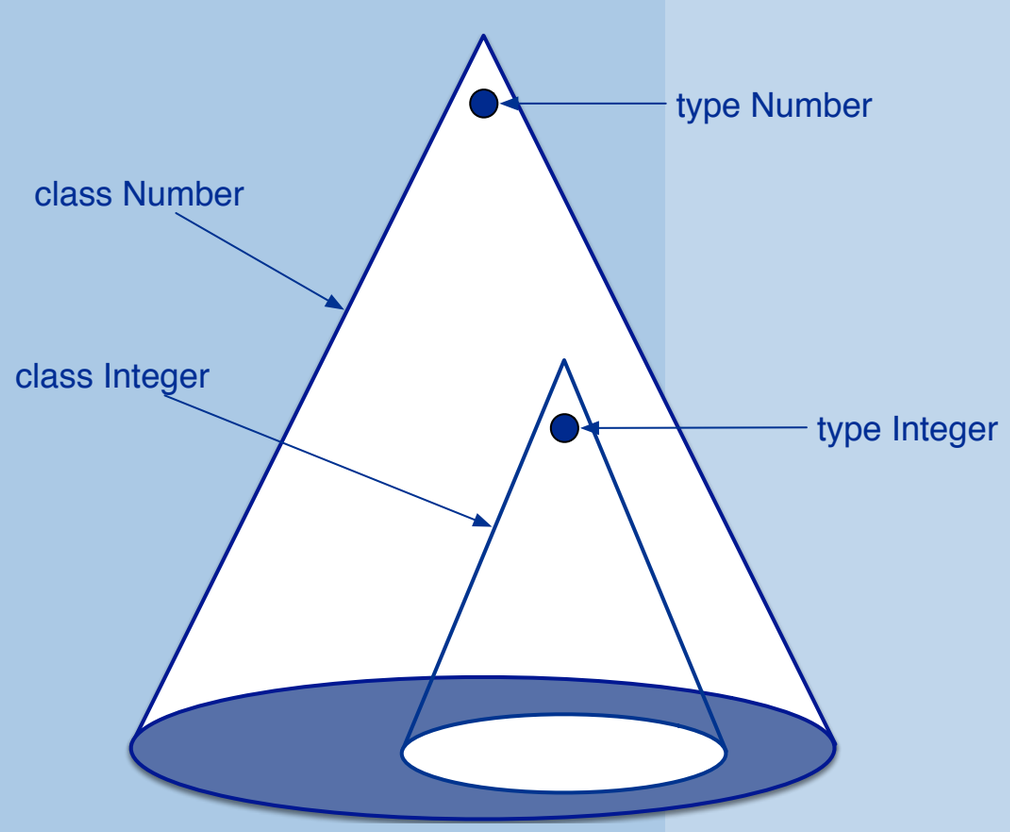
UNIVERSITÄT
BERN

9. Objects, Types and Classes

Oscar Nierstrasz



Based on material by Anthony Simons

Roadmap

1. *Objects and Type Compatibility*
   - The Principle of Substitutability
   - Types and Polymorphism
   - Type rules and type-checking
   - Object encodings and recursion

2. *Objects and Subtyping*
   - Subtypes, Covariance and Contravariance
   - Checking subtypes
   - Classes as families of types

Questions

- What is an object?
- What is a type?
- What is a subtype?
- What is the difference between a class and a type?
- What is the difference between inheritance and subtyping?
- When is a subclass also a subtype?

What is an object? Is it just a dictionary? Is it an autonomous agent?

What are classes and types? Are they just aspects of the same thing, or are types fundamentally different? What about dynamic class-based languages that don't have any formal notion of “type”?

Are inheritance and subtyping the same thing? Do they always coincide, or do they express very different things?

Roadmap

1. ***Objects and Type Compatibility***
   - The Principle of Substitutability
   - Types and Polymorphism
   - Type rules and type-checking
   - Object encodings and recursion

2. *Objects and Subtyping*
   - Subtypes, Covariance and Contravariance
   - Checking subtypes
   - Classes as families of types

Literature (I)

- Wilf LaLonde and John Pugh, *"Subclassing ≠ Subtyping ≠ Is-a,"* Journal of Object-Oriented Programming, vol. 3, no. 5, January 1991, pp. 57-62.

- Peter Wegner and Stanley B. Zdonik, *"Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like,"* Proceedings ECOOP '88 , S. Gjessing and K. Nygaard (Eds.), LNCS, vol. 322, Springer-Verlag, Oslo, August 15-17 1988, pp. 55-77.

- Barbara H. Liskov and Jeannette M. Wing, *"A behavioral notion of subtyping"*, In ACM Trans. Program. Lang. Syst. 16(6) p. 1811—1841, 1994.

- **Anthony J. H. Simons*, "The Theory of Classification"*, Parts 1-3, Journal of Object Technology, 2002-2003, www.jot.fm.**

This lecture is mainly based on Simons' JOT series, with some additional material from the other papers. All material can be found here:

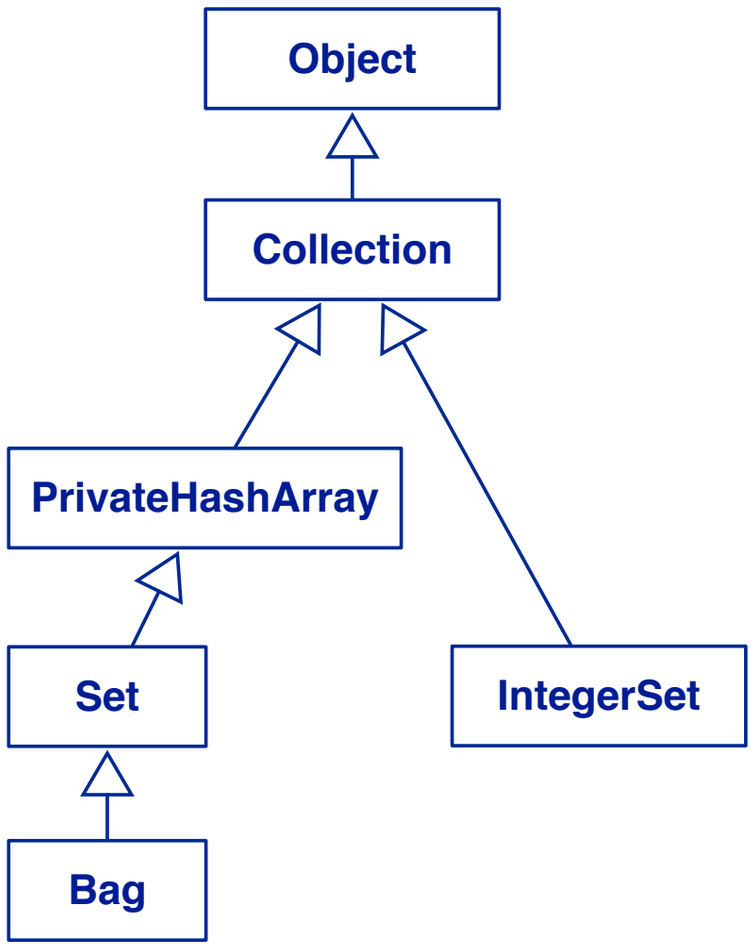
http://scg.unibe.ch/teaching/pl/resources

Roadmap

1. *Objects and Type Compatibility*
   - **The Principle of Substitutability**
   - Types and Polymorphism
   - Type rules and type-checking
   - Object encodings and recursion

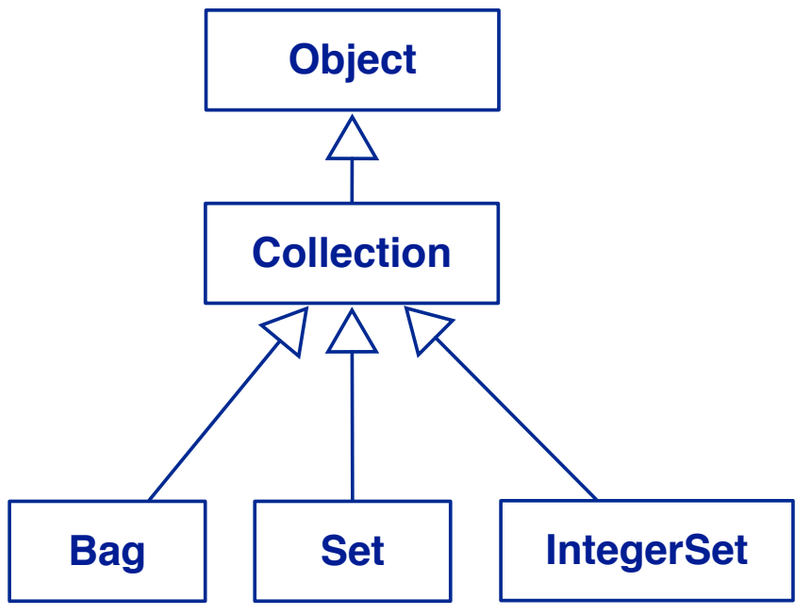
2. *Objects and Subtyping*
   - Subtypes, Covariance and Contravariance
   - Checking subtypes
   - Classes as families of types

Prelude: Subclassing ≠ Subtyping ≠ Is-a



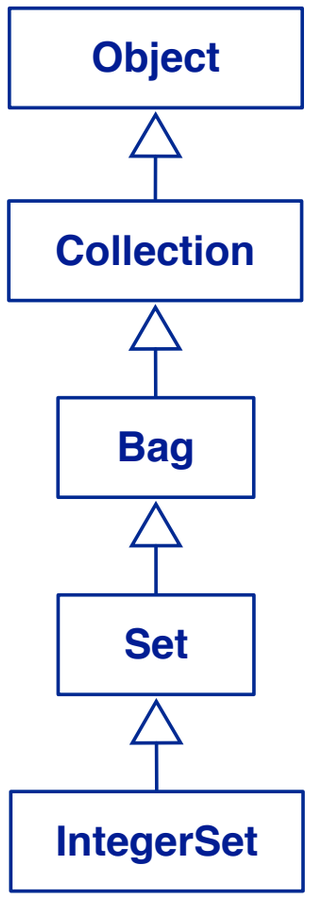
(a) Subclassing

incremental modification



(b) Subtyping

substitutability



(c) Is-A

specialization

This example comes from LaLonde and Pugh (JOOP, 1991).

Subclassing is an *incremental modification mechanism.*

Subtyping enforces *substitutability*.

Specialization says that one thing is a *special case* of another and does not (necessarily) imply substitutability.

(NB: IntegerSet <: Collection is perhaps debatable …)

This is the simple explanation of the difference between subclassing and subtyping. As we shall see, Simons presents a more refined notion of classes as families of types.

The Principle of Subtitutability

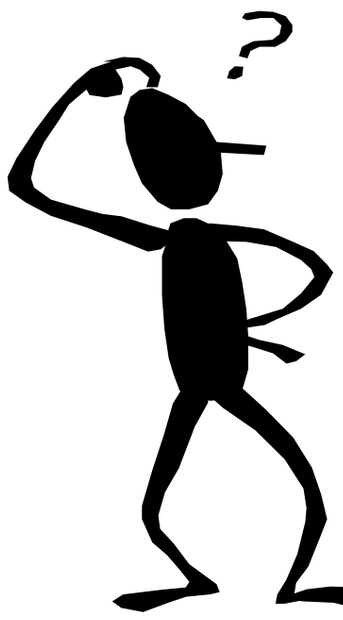
An instance of a subtype can always be used in any context in which an instance of a supertype was expected.

— Wegner & Zdonik

The quotation is from the classic ECOOP 1988 paper,
"Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like."

The statement can be interpreted in many ways. What does *"... can always be used ..."* mean? Does it just mean "won't cause any unexpected errors"? Or "will yield precisely the same result"? The choice leads to rather different forms of subtyping.

Components and Compatibility



A *client* has certain *expectations* of a component

A *compatible* component provides what the client requires

A *supplier* provides a component that should *satisfy* these expectations

Simons starts in Part 1 of his series of essays by considering what is a "*component*" and what it means for one component to be "*compatible*" with the *expectations* of a client. These expectations are what a type is meant to express.

Kinds of Compatibility

- ***Syntactic compatibility*** — the component provides all the expected operations (type names, function signatures, interfaces);

- ***Semantic compatibility*** — the component's operations all *behave* in the expected way (state semantics, logical axioms, proofs);

Most work published as “type theory” has concentrated on the first aspect, whereas the latter aspect usually comes under the heading of “semantics” or “model checking”.

Liskov substitutability principle

Let q(x) be a property provable about objects x of type T. Then q(y) should be true for objects y of type S, where S is a subtype of T.

— Liskov & Wing, 1994

http://en.wikipedia.org/wiki/Liskov_substitution_principle

This well-known principle (also referred to as “LSP”) is not so much a definition of substitutability as a general framework, depending on what kind of properties one is interested in.

Roadmap

1. *Objects and Type Compatibility*
   - The Principle of Substitutability
   - **Types and Polymorphism**
   - Type rules and type-checking
   - Object encodings and recursion

2. *Objects and Subtyping*
   - Subtypes, Covariance and Contravariance
   - Checking subtypes
   - Classes as families of types

Review: Static and Dynamic Typing

A language is statically typed if it is always possible to *determine the (static) type* of an expression *based on the program text alone.*

A language is dynamically typed if *only values have fixed type*. Variables and parameters may take on different types at run-time, and must be checked immediately before they are used.

A language is “strongly typed” if it is impossible to perform an operation on the wrong kind of object.

Type consistency may be assured by

- —compile-time type-checking,
- —type inference, or
- —dynamic type-checking.

See Cardelli & Wegner's classic 1985 Computing Surveys paper. All programming languages have some notion of type, so there is no such thing as an "untyped" languages. The difference is whether types are declared (and checked) statically, or only checked dynamically. The term "strongly typed" should be avoided.

Review: Polymorphism

- Monomorphism — variables hold values of exactly one type

- Polymorphism — variables may hold values of various types

All OO languages are polymorphic

All OO languages are polymorphic because we can always bind subclass instances to variables whose expected type is the superclass.

Recall that *subtype polymorphism* in OO languages differs from *parametric polymorphism* as seen in Haskell. In both cases variables can be bound to values of different types, but the underlying rules are rather different. We will explore both forms in detail in this lecture.

OO Polymorphism

"Beware object-oriented textbooks! Polymorphism does not refer to the dynamic behaviour of objects aliased by a common superclass variable, but to the fact that variables may hold values of more than one type in the first place. This fact is independent of static or dynamic binding."

— Simons, Theory of Classification, Part I

I.e., polymorphism has to do with whether single or multiple types are permitted in a given context. It has nothing to do with how messages are dispatched!

Review: Kinds of Polymorphism

- ***Universal:***
  - —Parametric: polymorphic map function in Haskell; nil/ void pointer type in Pascal/C
  - —Inclusion: subtyping — graphic objects
- ***Ad Hoc:***
  - —Overloading: + applies to both integers and reals
  - —Coercion: integer values can be used where reals are expected and v.v.

— *Cardelli and Wegner*

NB: Haskell supports parametric polymorphism and overloading, but not subtyping. It is an open problem whether ML style type inference can be extended to deal with subtyping.

In Java we find all four forms. We can bind variables to values belonging to a subtype (*inclusion*), we can define generic classes with type parameters (*parametric*), we can define multiple methods that take different types of arguments (*overloading*), and primitive values can be auto-boxed to objects, and unboxed again (*coercion*).

Kinds of Type Compatibility

Exact Correspondence	The component is *identical in type* and its behaviour exactly matches the expectations made of it when calls are made through the interface.
Subtyping	The component is a *more specific type, but behaves exactly like* the more general expectations when calls are made through the interface.
Subclassing	The component is a more specific type and *behaves in ways that exceed* the more general expectations when calls are made through the interface.

Exact correspondence refers to monomorphism.

Subtyping is inclusion polymorphism. As in LSP, we must be careful what we mean by “behaves exactly like”.

Subclassing in Wegner and Zdonik's view replies on inheritance as an “*incremental modification mechanism*”. As we shall see, “ways that exceed” has to do with how objects are recursively defined. We focus first on subtyping, with a view to later understanding subclassing.

Kinds of Types

Axioms and algebraic types

☞ *Types as sets of signatures and axioms*

Syntactic and existential abstract types

☞ *Types as sets of function signatures*

Model-based and constructive types

☞ *Types as sets ($x: X \Leftrightarrow x \in X$)*

Bit-interpretation schemas

☞ *Based on memory storage requirements*

Bit schemas are too low-level for understanding objects.

Types as sets are too restrictive, as we shall see.

Types as function signatures is the most practical approach.

The axiomatic approach is more ambitious, and typically goes beyond what is really needed in practice.

Roadmap

1. *Objects and Type Compatibility*
   - The Principle of Substitutability
   - Types and Polymorphism
   - **Type rules and type-checking**
   - Object encodings and recursion

2. *Objects and Subtyping*
   - Subtypes, Covariance and Contravariance
   - Checking subtypes
   - Classes as families of types

Dimensions of Type-checking

	Bit Schemas	***Interfaces***	***Algebras***
Exact	*C*	*Pascal, C, Ada, Modula2*	
Subtyping	*C++*	*C++, Java, Trellis*	*OBJ, Eiffel*
Subclassing		*Smalltalk, [Eiffel]*	

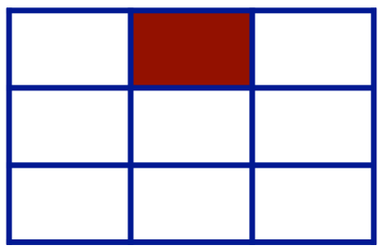
Simons, Part 2.

The areas we are more interested in are in darker grey. Note that we cannot really classify languages, since each language must typically deal with multiple issues. Rather we give some examples of languages where these issues come out more strongly.

Type-checking exact interfaces

Start with:

- *sets* A, B, ..., corresponding to the given primitive types in the universe; and
- *elements* a, b, ..., of these sets, corresponding to the values in the universe; and
- *set operations* such as membership $\in$, inclusion $\subseteq$, and union $\cup$; and
- *logical operations* such as implication $\Rightarrow$, equivalence $\Leftrightarrow$ and entailment $\vdash$



To warm up, we explore exact correspondence and types as interfaces.

We start with sets of primitive values, such as numbers and strings, and then we build up more complex values using various set operations.

Type Rules

$$\frac{n : N, m : M}{\langle n, m \rangle : N \times M}$$

Product Introduction

Says: "if n is of type N and m is of type M, then we may conclude that a pair ⟨n, m⟩ has the product type N × M"

We will be seeing lots of type rules expressed in the form. Read the bar as implication. Whatever is above the bar expresses our assumptions, and what follows below expresses our conclusions. To prove that an expression has a particualr type, we will use a number of these type rules in combination.

This particular rule expresses how we can build up simple tuple types from primitive types. Using this rule, if we know that `name` is a `string` and `salary` is an `integer`, then the pair `<name,salary>` is of type `string x integer`.

$$\frac{e : N \times M}{\pi_1(e) : N, \pi_2(e) : M}$$

Product Elimination

$$\frac{x : D \vdash e : C}{\lambda x.e : D \to C}$$

Function Introduction

$$\frac{f : D \to C, v : D}{f(v) : C}$$

Function Elimination

If e has type N×M, then $\pi_1(e)$ has type N, π_2(e) has type M.

If x has type D and e has type C, then a function $\lambda x.e$ has type D→C.

If f has type D→C and v has type D, then $f(v)$ has type C.

$$\frac{\alpha_i : A, e_i : T_i}{\{\alpha_1 \mapsto e_1, \cdots, \alpha_n \mapsto e_n\} : \{\alpha_1 : T_1, \cdots, \alpha_n : T_n\}} \; for\; i = 1..n$$

Record Introduction

$$\frac{e : \{\alpha_1 : T_1, \cdots, \alpha_n : T_n\}}{e.\alpha_i : T_i} \; for\; i = 1..n$$

Record Elimination

If α_i are labels and each e_i has type T_i, then the record $\{\alpha_i \rightarrow e_i\}$ has type $\{\alpha_i : T_i\}$ …

Applying the Rules

Consider:

$$\begin{aligned}\text{make-point} = \quad & \lambda(e : \text{Integer} \times \text{Integer}).\{ \\ & x \mapsto \pi_1(e), \\ & y \mapsto \pi_2(e), \\ & \text{equal} \mapsto \lambda(p : \text{Point}).(\pi_1(e) = p.x \wedge \pi_2(e) = p.y)\}\end{aligned}$$

$$\dfrac{\dfrac{e : \text{Integer} \times \text{Integer}}{\pi_1(e) : \text{Integer}, \pi_2(e) : \text{Integer}} \quad \dfrac{p : \text{Point}}{p.x : \text{Integer}, p.y : \text{Integer}}}{\text{make-point} : \text{Integer} \times \text{Integer} \to \text{Point}}$$

But what is the Point *type?!*

We define a JS-like make-point function that takes an pair containing x and y coordinates, and returns a record containing *x*, *y* and an *equals* "method".

We would like to use our type rules to conclude that:

make-point : Integer × Integer → Point

So far so good, but we never said what the Point type is … We must therefore take a closer look at what we mean by *objects*.

Roadmap

1. *Objects and Type Compatibility*
   - The Principle of Substitutability
   - Types and Polymorphism
   - Type rules and type-checking
   - **Object encodings and recursion**

2. *Objects and Subtyping*
   - Subtypes, Covariance and Contravariance
   - Checking subtypes
   - Classes as families of types

Object Encodings and Recursion

What is an object?

- ***Three answers:***
  - Existential object encoding
    - *Objects as data abstractions*
  - Functional object encoding
    - *Objects as functional closures*
  - Object calculus
    - *Objects as self-bound records*

Each theory has its pros and cons. We will focus on the functional view for the type theory.

See Simons, Part 3.

Existential Object Encoding

$$\begin{aligned}\text{Point} = \exists\text{rep}.(\text{rep}\times \quad \{ \quad & x : \text{rep} \to \text{Integer}; \\ & y : \text{rep} \to \text{Integer}; \\ & \text{equal} : \text{rep} \times \text{rep} \to \text{Boolean} \\ \})\end{aligned}$$

Says: There is some (hidden) representation that makes the Point interface work as it should.

The existential view is consistent with the principle of encapsulation. It says that there is some hidden representation, *rep*, which objects of type Point have access to. Since clients do not know what *rep* is, they can only access the public part, namely the record containing *x*, *y*, and the *equals* method. Each of these requires the *rep* part to work, but the client cannot do anything with it aside from pass it to the three operations.

The key point is that we do not say what *rep* is — any *rep* will do.

An Existential Point

$$\text{aPoint} = \langle\langle 2, 3\rangle, \quad \{ \quad x \mapsto \lambda(s : \text{rep}).\pi_1(s),$$
$$y \mapsto \lambda(s : \text{rep}).\pi_2(s),$$
$$\text{equal} \mapsto \lambda(p : \text{rep} \times \text{rep}).$$
$$(\pi_1(\pi_1(p)) = (\pi_1(\pi_2(p))$$
$$\wedge \pi_2(\pi_1(p)) = \pi_2(\pi_2(p)))$$
$$\}\rangle$$

Says: aPoint consists of some concrete representation and a record of methods for accessing the state.

Here is a concrete instance of type Point with its internal representation consisting of an ⟨*x*,*y*⟩ pair. The implementations of *x*, *y*, and *equals* know how to access this representation.

Note that clients do not know what the representation is. They simply pass it in to the methods without touching or examining it.

Pros and Cons

- ***Pros:***
  - Models private state.
  - Don't need recursive types.

- ***Cons:***
  - Clumsy invocation: $\pi_2(p).m(\pi_1(p))$ to invoke method m of p (!)
  - Mutator methods not directly modeled.

Simons points out: One way would be to define a special method invocation operator "•" to hide the ungainly syntax, such that the expression:

obj • msg(arg)

would expand to:

π2(obj).msg(π1(obj), π1(arg))

However, this has several drawbacks. Firstly, separate versions of "•" would be needed for methods accepting zero, or more arguments. Secondly, "•" would have to accept objects, messages and arguments of all types, requiring a much more complicated higher-order type system to express well-typed messages.

Functional Object Encoding

$$\text{Point} = \mu\ \text{pnt}.\ \{\ \ x : \text{Integer};\ \ y : \text{Integer};\ \ \text{equal} : \text{pnt} \to \text{Boolean}\ \})$$

Says: a Point is a record of methods, some of which manipulate Point objects

Recall: μ is a special operator used for recursive definitions. AKA "fix", "rec", "letrec" or "def".

$$(\mu\ \text{f.E})\ \text{e} \rightarrow \text{E}\ [(\mu\ \text{f.E})\ /\ \text{f}]\ \text{e}$$

This definition has the sense of “let *pnt* be a placeholder standing for the eventual definition of the Point type, which is defined as a record type whose methods may recursively manipulate values of this *pnt* type.”

In this style, “μ pnt” (sometimes notated as “rec pnt”) indicates that the following definition is recursive.

A Functional Point

$$
\begin{array}{l}
\textbf{let } xv = 2, yv = 3 \textbf{ in} \\
\qquad\qquad \text{aPoint} = \{ \quad x \mapsto xv, \\
\qquad\qquad\qquad\qquad\quad y \mapsto yv, \\
\qquad\qquad\qquad\qquad\quad \text{equal} \mapsto \lambda(p : \text{Point}). \\
\qquad\qquad\qquad\qquad\qquad\qquad (xv = p.x \wedge yv = p.y) \\
\qquad \}
\end{array}
$$

Says: aPoint is a *closure* consisting of a record with access to hidden state xv and yv.

Compare this with objects as closures in JavaScript.

Note we do not yet deal with *self*!

Objects and Recursion

$$
\begin{array}{l}
\textbf{let } xv = 2, yv = 3 \textbf{ in} \\
\qquad \text{aPoint} = \mu\ \text{self}.\{ \quad x \mapsto xv, \\
\qquad\qquad\qquad\qquad\quad\ \ y \mapsto yv, \\
\qquad\qquad\qquad\qquad\quad\ \ \text{equal} \mapsto \lambda(p : \text{Point}). \\
\qquad\qquad\qquad\qquad\qquad\quad (\text{self}.x = p.x \wedge \text{self}.y = p.y) \\
\qquad \}
\end{array}
$$

Says: aPoint is an object, with (recursive) access to itself

This recursive definition of a point is a functional closure that also recursively binds *self*. Note that *self* is the fixpoint.

Pros and Cons

- ***Pros:***
  - Models private state.
  - Direct interpretation of methods.

- ***Cons:***
  - Requires fixpoints.
  - Mutator methods not directly modeled.

Unfortunately fixpoints considerably complicate the type system.

So far all our “objects” are immutable values, so we do not handle mutator methods.

Abadi and Cardelli's Object Calculus

$$
\begin{array}{lll}
\text{aPoint} = [& x = \varsigma(\text{self})2, & \\
& y = \varsigma(\text{self})3, & \\
& \text{equal} = \varsigma(\text{self})\lambda(p) & (\ \text{self}.x = p.x \\
& & \wedge\ \text{self}.y = p.y) \\
]
\end{array}
$$

Says: aPoint is an object with self bound *by definition of the calculus*

Abadi & Cardelli define a kind of lambda calculus of objects as records, with a special operator, ς, which binds self references similar to the way that μ binds fixpoints.

See Abadi & Cardelli's "*A Theory of Objects*".

Roadmap

1. *Objects and Type Compatibility*
   - The Principle of Substitutability
   - Types and Polymorphism
   - Type rules and type-checking
   - Object encodings and recursion

2. ***Objects and Subtyping***
   - Subtypes, Covariance and Contravariance
   - Checking subtypes
   - Classes as families of types

Literature (II)

- Luca Cardelli and Peter Wegner, *"On Understanding Types, Data Abstraction, and Polymorphism,"* ACM Computing Surveys , vol. 17, no. 4, December 1985, pp. 471-522.

- William Cook, Walter Hill and Peter Canning, *"Inheritance is not Subtyping,"* Proceedings POPL '90 , San Francisco, Jan 17-19 1990.

- **Anthony J. H. Simons, *"The Theory of Classification"*, Parts 4-8, Journal of Object Technology, 2002-2003, www.jot.fm.**

Roadmap

1. *Objects and Type Compatibility*
   - The Principle of Substitutability
   - Types and Polymorphism
   - Type rules and type-checking
   - Object encodings and recursion

2. *Objects and Subtyping*
   - **Subtypes, Covariance and Contravariance**
   - Checking subtypes
   - Classes as families of types

Generators and fixpoints

$$\begin{aligned}\text{aPoint} = \mu\ \text{self}.\{\quad & x \mapsto 2,\\ & y \mapsto 3,\\ & \text{equal} \mapsto \lambda(p : \text{Point}).\\ & \qquad (\text{self}.x = p.x \wedge \text{self}.y = p.y)\\ \}&\end{aligned}$$

Can be seen as a shortcut for …

$$\begin{aligned}\text{genAPoint} \quad = \quad & \lambda\ \text{self}.\{\quad x \mapsto 2,\\ & \qquad\qquad y \mapsto 3,\\ & \qquad\qquad \text{equal} \mapsto \lambda(p : \text{Point}).\\ & \qquad\qquad\qquad (\text{self}.x = p.x \wedge \text{self}.y = p.y)\\ & \qquad\qquad \}\\ \text{aPoint} \quad = \quad & \mathbf{Y}\ \text{genAPoint}\\ \Rightarrow \quad & \text{genAPoint}(\text{genAPoint}(\text{genAPoint}(\cdots)))\end{aligned}$$

We can view μ as a kind of built-in fixpoint operator.

Recursive types

The Point *type* must also be recursive:

$$\text{Point} \quad = \quad \mu\sigma.\{x : \text{Integer}; y : \text{Integer}; \text{equal} : \sigma \to \text{Boolean}\}$$

Which can be seen as a shortcut for …

$$\begin{aligned} \text{GenPoint} &= \lambda\sigma.\{x : \text{Integer}; y : \text{Integer}; \text{equal} : \sigma \to \text{Boolean}\} \\ \text{Point} &= \mathbf{Y}\ \text{GenPoint} \\ &\Rightarrow \text{GenPoint}[\text{GenPoint}[\text{GenPoint}[\cdots]]] \end{aligned}$$

… What does subtyping mean for such types?

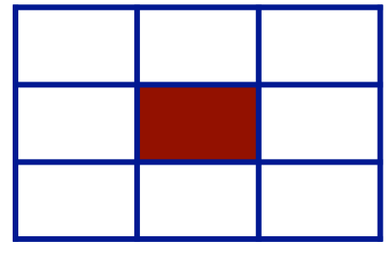
We use μ not only for objects, but also for types. As with objects, the μ represents the “unfolding” of a recursively-defined type.

Although this we can easily make sense of this notion of a type as a specification of a client’s expectations, it is not at all clear what a “subtype” would be. We need to start over and examine what we mean by a “type”.

Types as Sets

Sets	***Types***
$x \in X$	$x : X$
x is a member of X	*x is of type X*
$Y \subseteq X$	$Y <: X$
Y is a subset of X	*Y is a subtype of X*

E.g., john : Student ⇒ john : Person



Caveat: *Although this view of types as sets is intuitive and appealing, it leads to conceptual problems … as we shall see.*

We start with a simple view of types as sets. If `Student` is a *subclass* of `Person`, then it is also a *subtype*, and every *student* is then also a member of the *set of persons*.

Function subtyping

Suppose:

$$f_Y : D_Y \to C_Y$$

$$f_X : D_X \to C_X$$

✎ When can f_Y be safely substituted for f_X?

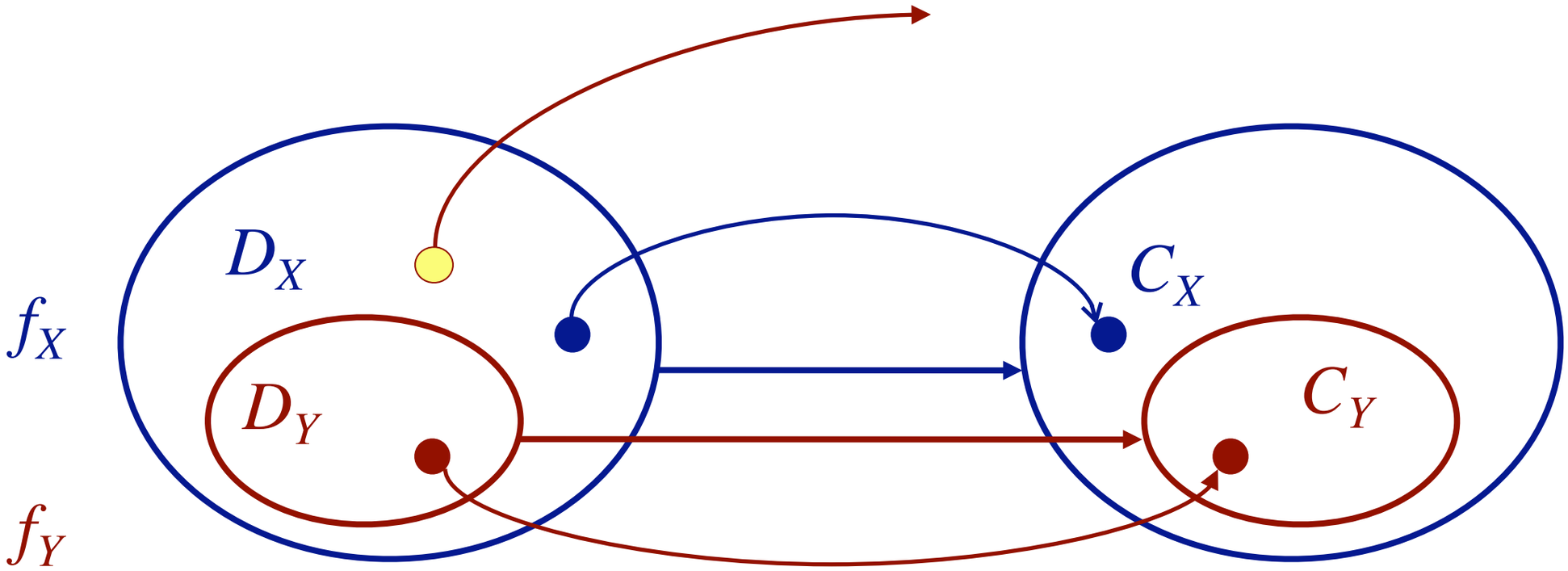
Ask: What relationships must hold between the *sets* D_X , D_Y , C_X and C_Y?

If f_Y can be *substituted* for f_X, then it must accept as *argument* everything that f_X accepts.

Similarly, the *result* of f_Y must be substitutable for the result of f_X.

Try to draw Venn diagrams for these sets. (Map to Student, Person, Dog, Pet.)

Covariant types



A client who expects the behaviour of f_X, and applies f_Y to a value in D_X might get a run-time type error.

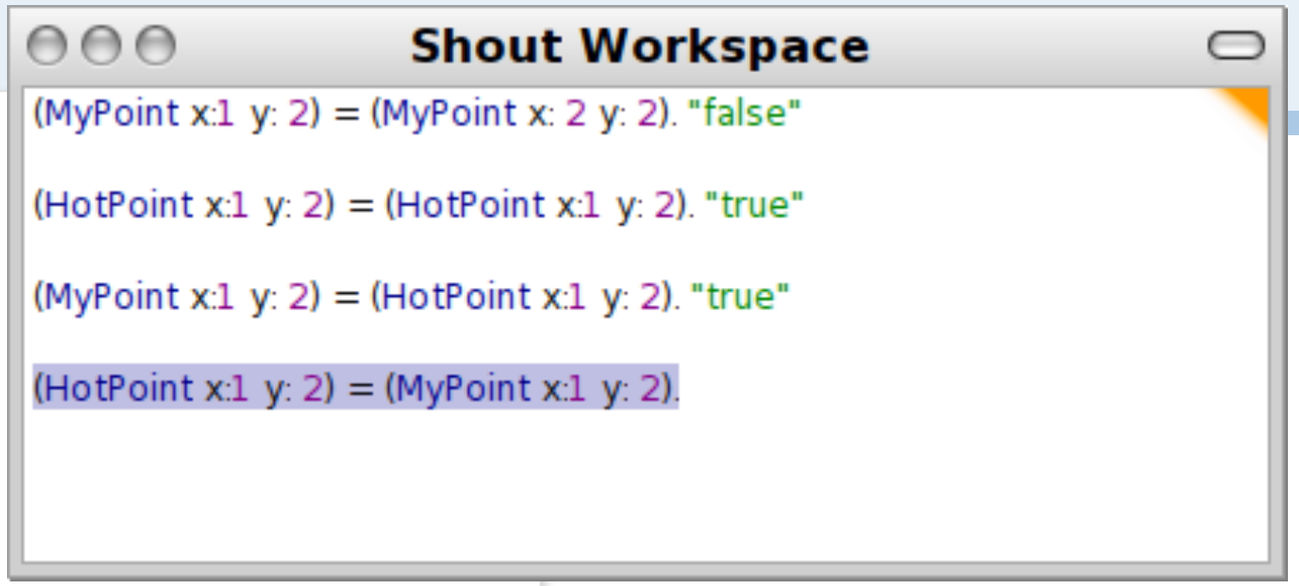
Covariant subtyping means that to subtype a function, we *subtype both domain and codomain*. This is often what we want to do in practice, since when we specialize a class we often specialize all the operations that manipulate instances.

Unfortunately, a value in D_X but not in D_Y may be rejected by f_Y. As a consequence, *covariant subtyping is not safe*.

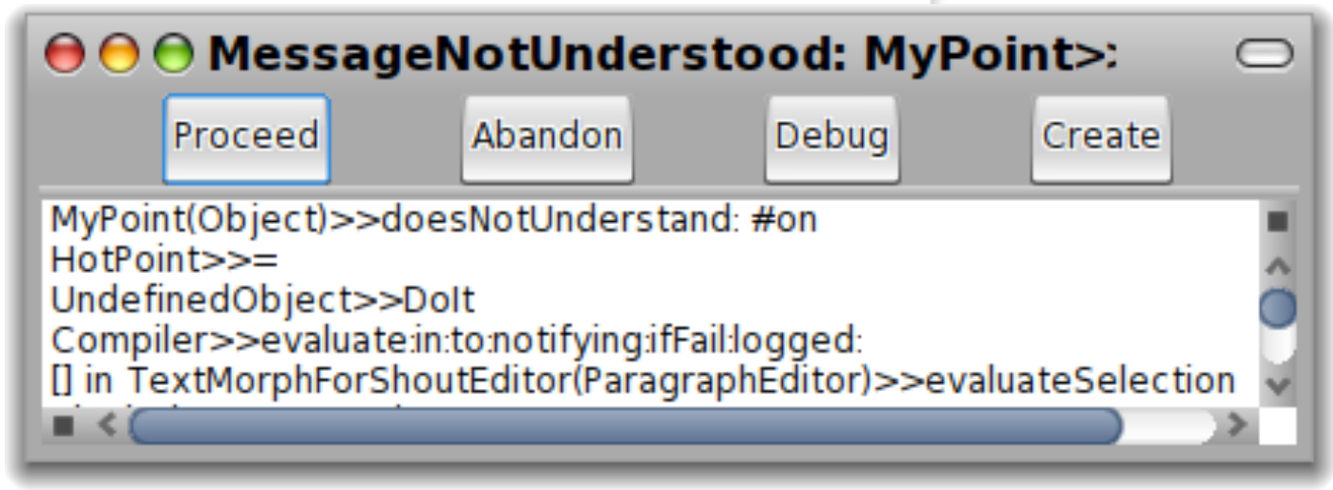
This is the Eiffel approach. Actually, the Eiffel compiler flags these, and performs a run-time type check.

Covariant type errors

```
Object subclass: #MyPoint
 instanceVariableNames: 'x y '
x
 ^x
y
 ^y
= aPoint
 ^ self x = aPoint x and: [self y = aPoint y]
```



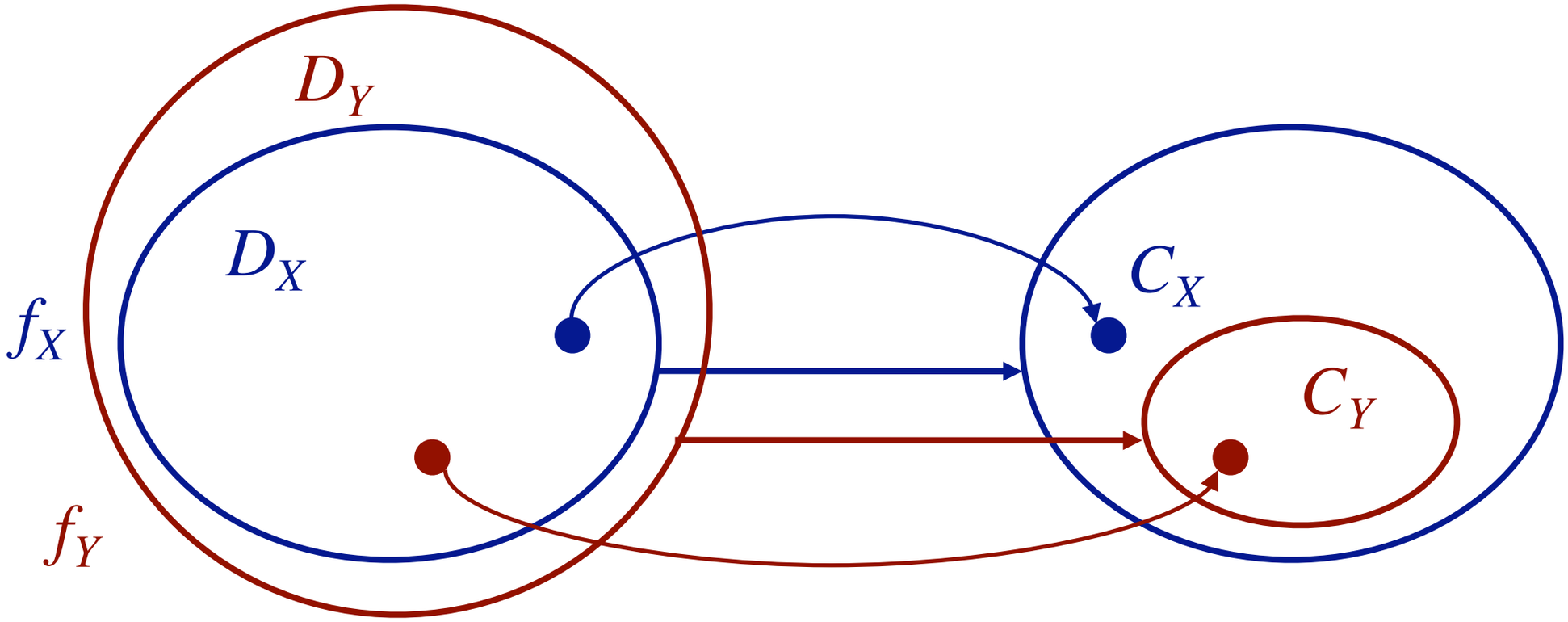

```
MyPoint subclass: #HotPoint
 instanceVariableNames: 'on '
on
 ^ on
toggle
 on := on not
on: boolean
 on := boolean
= aHotPoint
 ^ super = aHotPoint and: [ self on = aHotPoint on ]
```



We skip the constructors and the initializers for conciseness. `HotPoint` specializes `Point`, and `HotPoint`'s `=` also tries to specialize `Point`'s `=`. This is fine, as long as we don't mix Points and `HotPoint`.

Note that a proper `=` method should accept *any object* as its argument, and return `false` if the argument is not an instance of the same class.

Contravariant types



A contravariant result type guarantees that the client will receive no unexpected results.

Contravariant subtyping works in the *opposite* direction for arguments — to subtype a function, we need a *supertype* of the domain, but a subtype of the co-domain.

f_Y should *accept as argument at least* everything that f_X accepts, and *return at most* everything that f_X returns.

A client supplying an argument in domain D_X is guaranteed to get a result in codomain C_X, since C_Y is contained in C_X.

Covariance and contravariance

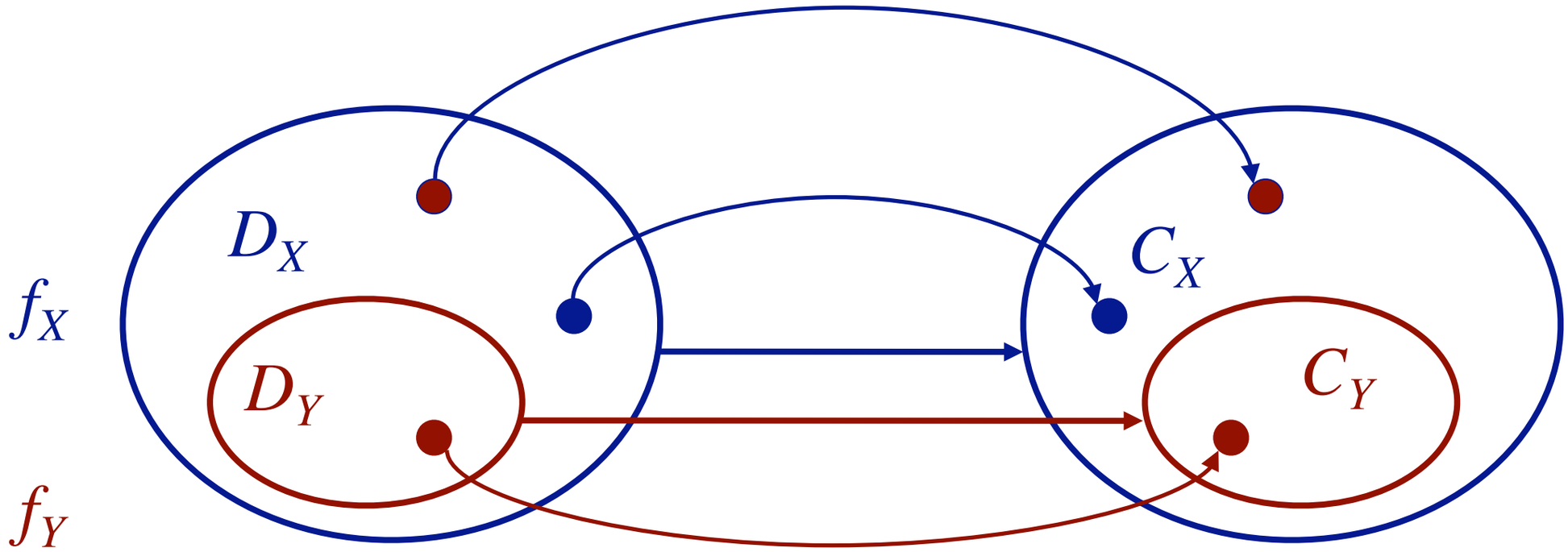
$$\frac{D_X <: D_Y, C_Y <: C_X}{D_Y \to C_Y <: D_X \to C_X}$$

For a function type $D_Y \to C_Y$ to be a subtype of (I.e., substitutable for) $D_X \to C_X$, we must be covariant in the result type, but *contravariant* in the argument type!

Use the intuition that D_X is a subset of D_Y etc. Note that Eiffel does this wrong (but with good justification).

This is a nice, clean rule, but violates our modeling principles — usually we really do want to specialize both the domain and co-domain.

Overloading



With overloading, the old and the new methods co-exist. No type errors arise, but the domains of f_X and f_X will overlap, leading to subtle problems …

In theory this looks very nice, and it is a very practical solution adopted by both Java and C++. The problem, as we shall see in the example, is that f_X and f_Y are *two independent methods whose domains overlap*. Which one will be called *depends on the static type of the target*. Furthermore, the view of types as sets is not really valid, as the following example illustrates.

Example

```
class Point {
  private int x, y;
  Point(int x, int y) {
    this.x = x; this.y = y; }
  int getX() { return x; }
  int getY() { return y; }
  boolean equals(Point other) {
    return (this.getX() == other.getX())
    && (this.getY() == other.getY());
  }
}
```

```
class HotPoint extends Point {
  private boolean on = false;
  HotPoint(int x, int y) { super(x, y); }
  void toggle() { on = !on; }
  boolean getOn() { return on; }
  boolean equals(HotPoint other) {
    return super.equals(other)
    && (this.getOn() == other.getOn());
  }
}
```

```
class Main {
  public static void main(String args[]) {
    HotPoint hotpt1, hotpt2;
    hotpt1 = new HotPoint(3, 5);
    hotpt2 = new HotPoint(3, 5);
    hotpt2.toggle();
    compare(hotpt1, hotpt2);
  }
  private static void compare(Point pt1, Point pt2) {
    System.out.println(pt1.toString() + " is " +
    (pt1.equals(pt2) ? "" : "not ") + "the same as " + pt2);
  }
}
```

```
HotPoint@7e0df503 is the
same as HotPoint@4650d89c
```

What went wrong?!

The `Point` class has an `equals()` method that only accepts a `Point` object as its argument. `HotPoint` extends `Point` with a boolean flag that can be toggled, and defines its own `equals()` method that accepts a `HotPoint` instance.

We create two equal `HotPoint` instances and toggle the flag of one of them so they should no longer be equal. However the main program reports that they are still equal! What went wrong?

Note that equals is covariant in its argument. What we have here is overloading, not subtyping! If we had strict subtyping, the example would not even compile!

Overloading

- The *static type* of `pt1` is used to select the message to send, namely `equals(Point)`
- This is *different* from `equals(HotPoint)`, which is only visible in the subclass
  - There are two different equals methods, and the wrong thing happens
- If `equals(HotPoint)` could *override* instead of *overload* `equals(Point)`, then we could instead have run-time type errors (cf. Eiffel “catcalls”)

In Java, a subclass method may not change the signature of a method it overrides. If the argument type changes, it is interpreted as an overloaded method. If just the return type changes, it is an error.

Roadmap

1. *Objects and Type Compatibility*
   - The Principle of Substitutability
   - Types and Polymorphism
   - Type rules and type-checking
   - Object encodings and recursion

2. *Objects and Subtyping*
   - Subtypes, Covariance and Contravariance
   - **Checking subtypes**
   - Classes as families of types

Record extension

$$\frac{a_1, \cdots a_k, \cdots a_n : A}{\{a_1 : T_1, \cdots a_n : T_n\} <: \{a_1 : T_1, \cdots a_: T_k\}} \; for\ 1 <= k <= n$$

Says: A record (object) with *more fields* can be safely substituted for one with fewer fields

Cf. Java

Java, Eiffel and C++ apply this rule. Note that we do not change the type of the common fields.

Record overriding

$$\frac{a_i : A, S_i <: T_i}{\{a_i : S_i\} <: \{a_i : T_i\}} \; for\; i = 1..n$$

Says: the type of a record is a subtype of another record, if each of its fields is a subtype of the same field of the second.

Is this really useful in practice?

In practice, few languages support this. Java and C++ do not, and Eiffel applies a covariant rule. The main reason is that it is not really useful for practical programming problems. Typically we want specialization, not subtyping.

Record subtyping

$$\frac{a_i : A, S_1 <: T_1, \cdots S_k <: T_k}{\{a_1 : S_1, \cdots a_n : S_n\} <: \{a_1 : T_1, \cdots a_k : T_k\}} \; for\ 1 <= k <= n$$

Generalizes the two previous rules

This version simply combines the rules for extension and overriding to yield subtyping.

Recursive subtypes

$$\frac{\sigma <: \tau \vdash S <: T}{\mu\sigma.S <: \mu\tau.T} \; \sigma \textit{ free only in } S, \; \tau \textit{ free only in } T$$

Says: if knowing that σ is a subtype of τ lets you conclude that S is a subtype of T, then the recursive type $\mu\sigma$.S is a subtype of $\mu\tau$.T.

Can we use this to conclude that $\mu\ \sigma.\mathrm{Int} < \mu\ \tau.\mathrm{Num}$?
(Where σ and τ are the self types of Int and τ.Num.)

Looks ok, but …

Rule by Cardelli. We need this to deal with the fact that self gets specialized down the class hierarchy.

Note that σ and τ represent the self-types. This looks reasonable, but actually poses nasty problems, as we will see shortly …

Types in practice

- Smalltalk:
  - —overriding (no static type-checking)
  - —subclass may disable or restrict inherited methods
- C++ & Java:
  - —overriding + overloading
  - —exact type-matching for overridden methods
  - —cast vs. downcast (`dynamic_cast` in C++)
- Eiffel:
  - —covariant type rule with CAT-call detection
  - —contravariant rule for assertions

In Smalltalk, `FixedSizeCollection` is a subclass of `Collection`; the subclass may raise an overflow exception in the `add:` method.

CAT = Change in Availability or Type ⇒ run-time check. In Eiffel the HotPoint example will yield a run-time type error.

Roadmap

1. *Objects and Type Compatibility*
   - The Principle of Substitutability
   - Types and Polymorphism
   - Type rules and type-checking
   - Object encodings and recursion

2. *Objects and Subtyping*
   - Subtypes, Covariance and Contravariance
   - Checking subtypes
   - **Classes as families of types**

The Problem with Recursive Closure

Consider:

$$\text{Animal} = \mu\,\sigma.\,\{\ldots, \text{mate} : \sigma \to \sigma, \ldots\}$$

$$\text{Animal.mate} : \text{Animal} \to \text{Animal}$$

What about Dogs and Cats?

$$\text{Dog.mate} : \text{Dog} \to \text{Dog}$$

$$\text{Cat.mate} : \text{Cat} \to \text{Cat}$$

Covariance breaks subtyping!

Let's enforce subtyping …

$$\text{Dog} = \mu\,\sigma.\,\{\ldots, \text{mate} : \text{Animal} \to \sigma, \ldots\}$$

$$\text{Dog.mate} : \text{Animal} \to \text{Dog}\ ?!$$

Preserves subtyping but breaks nature!

In the first case we try to specialize in a covariant way. This models the world but breaks subtyping.

In the second case we have subtyping, but this does not reflect our domain.

What is really going on is therefore not subtyping, but something else, namely … *subclassing*.

The Problem of Type-loss

Consider:

$$\begin{aligned}\text{Number} &= \mu\,\sigma.\,\{\ \text{plus} : \sigma \to \sigma\ \} \\ &= \{\ \text{plus} : \text{Number} \to \text{Number}\ \}\end{aligned}$$

$$\begin{aligned}\text{Integer} &= \mu\,\sigma.\,\text{Number} \cup \{\ \text{minus} : \sigma \to \sigma, \ldots\} \\ &= \{\ \text{plus} : \text{Number} \to \text{Number}, \\ &\quad\ \text{minus} : \text{Integer} \to \text{Integer}, \ldots\}\end{aligned}$$

Now:

```
i, j, k : Integer
i.plus(j).minus(k)
```

fails to typecheck!

Here we see how the view of types as sets really breaks down. Inheritance as set union does not really express well what is going on.

In Java and C++ we need to perform a run-time type-check to recover the lost type information.

We are trying to hard to make `Integer <: Number`. This is not really what we want or need.

Classes as Type Generators

Now consider:

GenNumber = λ σ. { plus : σ→σ }
GenNumber[Number] = { plus : Number → Number }
GenNumber[Integer] = { plus : Integer → Integer }
GenNumber[Complex] = { plus : Complex → Complex }

What about subtyping?

Integer <: Number ***(no!)***

Integer <: GenNumber[Integer] ***(yes!)***

Instead of thinking of a class as having a fixed type, we should think of it as being a generator of types.

`GenNumber` is the (generic) type of `Number`, which is instantiated when we instantiate `Number` or one of its subclasses.

`Integer` is not a subtype of `Number` (since contravariance is violated) but of `GenNumber[Integer]`. The `Integer` type extends `GenNumber[Integer]` by ordinary record extension.

F-bounded quantification

What are the types of the Number class?

$$\forall(\tau <: \text{GenNumber}[\tau])$$

Says: the class of Numbers represents the *family of types* which are subtypes of GenNumber applied to themselves.

Note that τ appears on both sides of the subtype equation.

See Canning et al, "F-Bounded Polymorphism for Object-Oriented Programming", OOPSLA 1989.

The take-home message is that we should not expect subtyping and subclassing to always coincide.

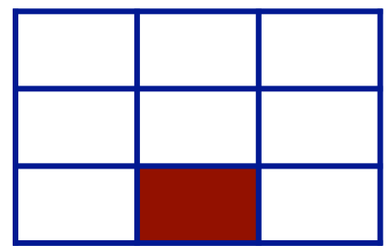
Classes as *families* of types

> So what about Animals and Dogs?

$$\text{GenAnimal} = \lambda\,\sigma.\,\{\ \text{mate} : \sigma \to \sigma\ \}$$
$$\forall(\tau <: \text{GenAnimal}\ [\tau]).\ \tau.\text{mate} : \tau \to \tau$$

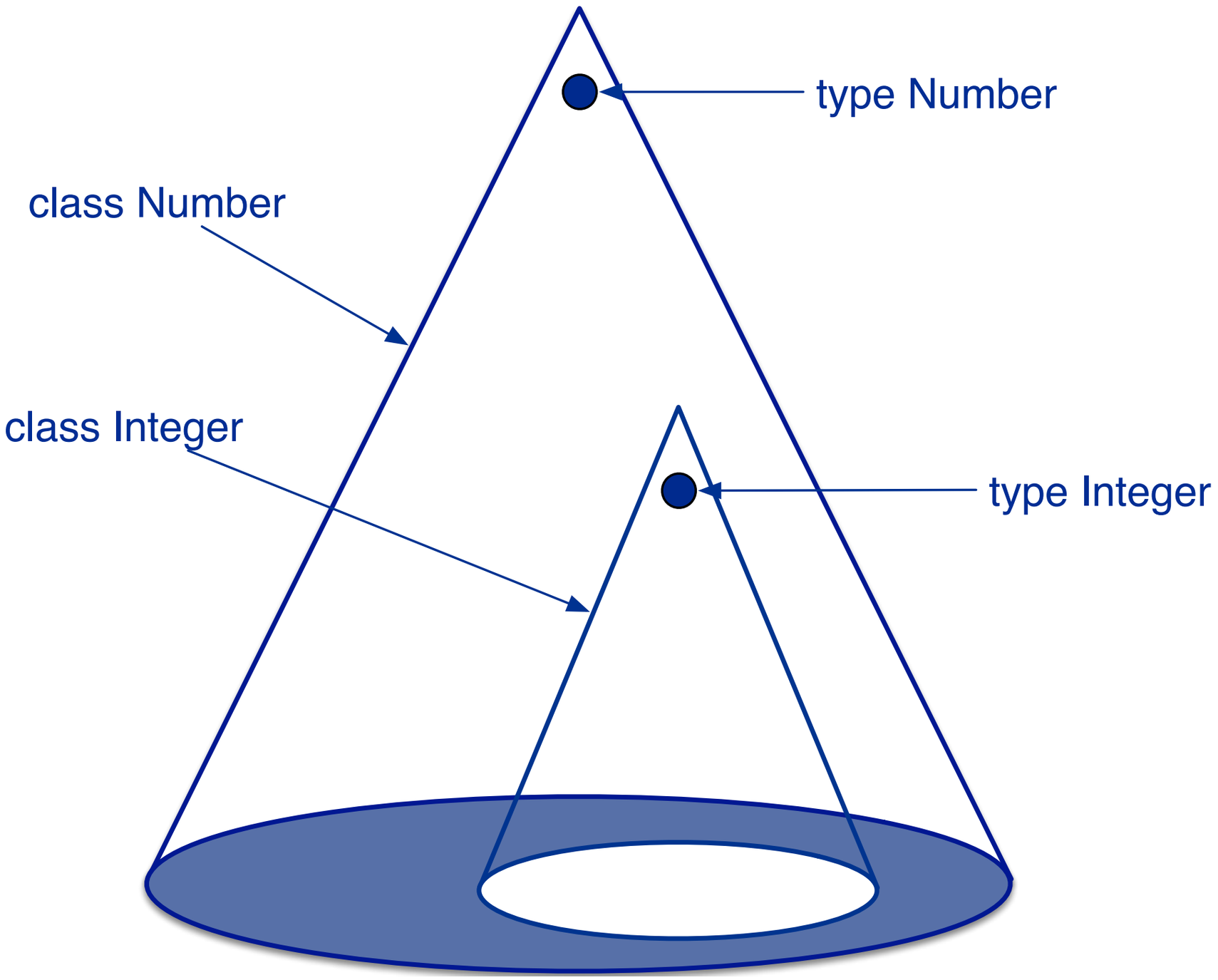
> So: $\text{Dog} :< \text{GenAnimal}[\text{Dog}]$

> But not $\text{Dog} <: \text{Animal}$



Animal represents a family of types, each with the same structure. But we must not necessarily expect subtype relationships between the subclasses.

Types and classes



“*Classes* are nested volumes in the space of types. *Types* are points at the apex of each bounded volume.”

Inheritance — subclassing without subtyping

$$\text{GenNumber} = \lambda\,\tau.\ \{\ \text{plus} : \tau \to \tau\ \}$$

$$\begin{aligned}
\text{GenInteger} \;=\; & \lambda\,\sigma.\ (\text{GenNumber}\ [\sigma] \\
& \cup\ \{\quad \text{minus} : \sigma\to\sigma, \\
& \qquad\ \ \text{times} : \sigma\to\sigma,\ \text{divide} : \sigma\to\sigma\ \}) \\
= \;\; & \lambda\,\sigma.\ \{\ \text{plus} : \sigma\to\sigma,\ \text{minus} : \sigma\to\sigma, \\
& \qquad\ \ \text{times} : \sigma\to\sigma,\ \text{divide} : \sigma\to\sigma\ \}
\end{aligned}$$

$$\text{Integer} = \text{fix}\ \text{GenInteger}$$

There is no subtype relation. Inheritance gives you a subclass relation. `Integer` is subtype of `GenInteger[Integer]`, but not of `Number`!

(Partial) Answers

- What is an object?
  - *An object is a (functional?) closure.*
- What is a type?
  - *A type is a specification of component compatibility.*
- What is a subtype?
  - *A subtype instance can be safely substituted where its supertype is expected.*
- What is the difference between a class and a type?
  - *A class is a family of types.*
- What is the difference between inheritance and subtyping?
  - *Inheritance extends a class to yield a new family of types.*
- When is a subclass also a subtype?
  - *In general only when record extension and the contravariant subtyping rules are obeyed.*

Part 1 — What you should know!

- What is the difference between subtyping and specialization?
- What is the Principle of Substitutability?
- What does it mean for a component to “satisfy” a clients expectations?
- What is the difference between syntactic and semantic compatibility?
- In what way are object-oriented languages “polymorphic”?
- Which forms of polymorphism does Java support?
- Why is subtyping a “universal” form of polymorphism?
- Why are there so many different interpretations of types?
- What does the “turnstile” (entailment) mean in a type rule?
- How does the existential object encoding avoid recursion?
- What is the μ operator for?
- What is a closure?
- How is an object like a closure?

Part 2 — What you should know!

- Why does a generator need to take a fixpoint? What does the fixpoint represent?
- How are types like sets? How are types not like sets?
- What is meant by covariance? By contravariance?
- How can covariance lead to type errors?
- How does contravariance support subtyping?
- Why is contravariance undesirable in practice?
- How does (Java-style) overloading sidestep the covariant/ contravariant dilemma?
- How does Eiffel sidestep the dilemma?
- How does the contravariant type rule apply to assertions?
- What problems do recursive types pose for subtyping?
- What is the difference between a type and a type generator?
- How is a class like a family of types?