Exploring Object-Oriented Type Systems

Tony Simons

A.Simons@dcs.shef.ac.uk

A J H Simons,
Department of Computer Science,
Regent Court, University of Sheffield,
211 Portobello Street,
SHEFFIELD, S1 4DP, United Kingdom.

OOPSLA '94 Tutorials
Portland Oregon, October 1994

Overview

- Motivation
- Classes and Types
- Abstract Types and Subtyping
- Type Recursion and Polymorphism
- Implications for Language Design
- Reference Material and Appendix

Motivation: Practical

Object-oriented languages have developed ahead of underlying formal theory:

- Notions of "class" and "inheritance" may be ill-defined.
- Programmers may confuse classes and types, inheritance and subtyping.
- Type rules of OOLs may be compromised - formally incorrect.
- Type security of programs may be compromised - unreliable.

There is an immediate need...

- to uncover the relationship between classes (in the object-oriented sense) and types (in the abstract data type sense).
- to construct a secure type model for the next generation of object-oriented languages.

Motivation: Theoretical

OOLs introduce a powerful combination of language features for which theory is immature.

Challenge to mathematicians:

- To extend the popular treatments of types in strongly-typed languages to allow for systematic sets of relationships between types.
- To present a convincing model of type recursion under polymorphism.
- Plausible link between object-oriented type systems and order-sorted algebras (Category Theory).

Classes and Types

First, a look at some of the issues surrounding classes and types.

- What are types?
- What are classes?
- Convenience viewpoint:
  "classes are not like types at all".
- Ambitious viewpoint:
  "classes are quite like types".
- Conflict between viewpoints.
- Separation of viewpoints.
- The future of classification?

What is a Type?

- Concrete: a schema for interpreting bit-strings in memory
- Eg the bit string

  01000001

  is 'A' if interpreted as a CHARACTER;

  is 65 if interpreted as an INTEGER;

- Abstract: a mathematical description of objects with an invariant set of properties:
- Eg the type INTEGER

  INTEGER $\triangleq$ Rec i . { plus : $i \times i \rightarrow i$;
  minus : $i \times i \rightarrow i$; times : $i \times i \rightarrow i$;
  div : $i \times i \rightarrow i$; mod : $i \times i \rightarrow i$ }

  $\forall$i,j,k : INTEGER
  plus(i,j) = plus(j,i)
  plus(plus(i,j),k) = plus(i,plus(j,k))
  plus(i,0) = i
  ...

What is a Class?

Not obvious what the formal status of the object-oriented *class* is:

- type - provides interface (method signatures) describing abstract behaviour of some set of objects;
- template - provides implementation template (instance variables) for some set of objects;
- table - provides a table (class variables) for data shared among some set of objects.

In addition, each of these views is open-ended, through inheritance:

- incomplete type;
- incomplete template;
- incomplete table...

Two Viewpoints

A class can be viewed as a kind of extensible record:

- storage for data;
- storage for methods;

Class seen as a *unit of implementation* (convenience viewpoint).

A class can be viewed as a kind of evolving specification:

- adding new behaviours (adding method signatures);
- making behaviours more concrete (implementing/re-implementing methods);
- restricting set of objects (subclassing).

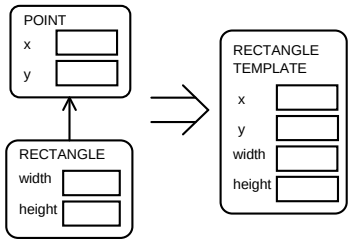
Class seen as a *unit of specification* (ambitious viewpoint).

Convenience Viewpoint

Class as a unit of implementation: formally lax; but with some advantages...

- maximum reuse of implementations (but some odd abstractions);
- economy in levels of indirection (in structures) and levels of nesting (in call-graphs).

eg RECTANGLE as a subclass of POINT:

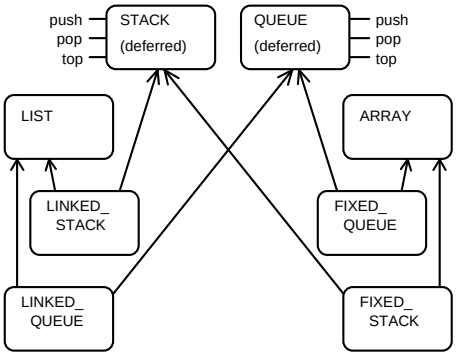


but is a RECTANGLE really a kind of POINT? Odd taxonomy.

Ambitious Viewpoint

Class as a unit of specification: formally strict;

eg providing abstract specifications with multiple alternative implementations in Eiffel (Meyer, 1988 and 1992):



In Eiffel and Trellis (Schaffert *et al*, 1986):

- classes *are* types
- subclasses *are* subtypes

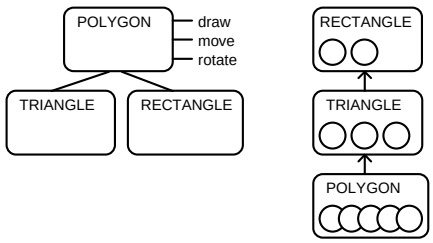
but is this formally correct? ...

Strong and Weak Inheritance

Clash of ambitious/convenience views:

Strong inheritance: sharing specification - functional interface and type axioms by which all descendants should be bound.

Weak inheritance: sharing implementation - opportunistic reuse of functions and declarations for storage allocation.



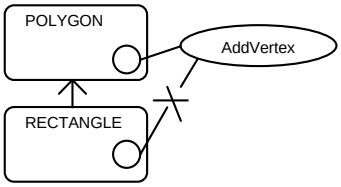
eg in Smalltalk (Goldberg and Robson, 1983):

Maximising reuse of storage for corners of figures {origin, extent, ... nth vertex} leads to strange type taxonomies.

Creeping Implementation

Clash of ambitious/convenience views:

Selective inheritance: introduced through orthogonal export rules; undefinition rules, eg in Eiffel (Meyer, 1988 and 1992):



Implementation concerns creep into abstract specification of POLYGON:

- intended as abstract type for all closed figures;
- actually used to model concrete N-vertex polygons.

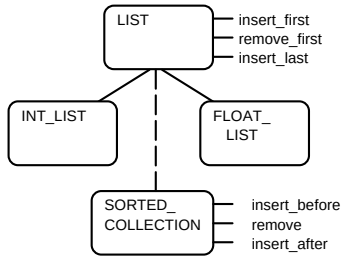
...but a RECTANGLE can' t add to its vertices!!

Leads to type violation - RECTANGLE does not respond to all the functions of POLYGON, therefore cannot be a POLYGON.

Separation of Concerns

Separation of ambitious/convenience views:

In C++ (Stroustrup, 1991) classes are also types, but sometimes inheritance is not subtyping:



Two kinds:

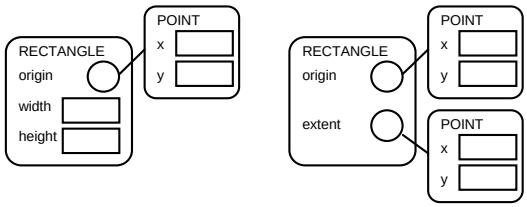
- *private* inheritance - subclass only inherits implementation of its parent;
- *public* inheritance - subclass also inherits specification of its parent.

An INT_LIST is type-compatible with LIST. A SORTED_COLLECTION is not.

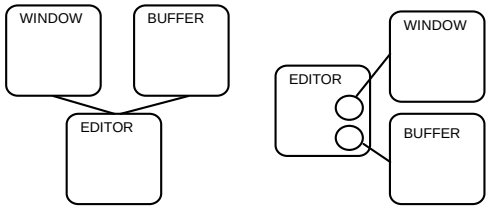
Class/Type Independence

Objects *seem* to have class and type independently (Snyder, 1987):

M:1 mappings from class hierarchies into type hierarchies, due to multiple concrete representations:



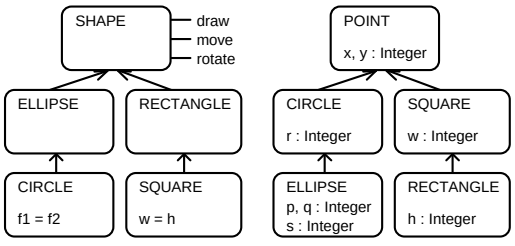
M:1 mappings from class hierarchies into type hierarchies due to free choice between inheritance and composition:



Separate Sharing of Class and Type

Separation of notions of class and type, eg CommonObjects (Snyder, 1987) and POOL-I (America, 1990):

- can reason about implementation and type independently;
- orthogonal class and type hierarchies:



Separation of specification and implementation concerns, eg Emerald (Raj and Levy, 1989):

- hierarchy used to express type-sharing;
- implementation-sharing only through composition.

A Failure of Nerve?

Is this impoverished view of class a failure of nerve?

Class *fulfils the same role* as type for OOP:

- classification a natural activity in Psychology, undergirds types and abstraction;
- concept differentiation in AI can be compared with coerceable typing systems;
- strong desire to capture abstraction even in the type-free OOP languages;
- traditional languages have not addressed the possibility of systematic sets of relationships between types;

The fact that something systematic is possible in OOP means that there probably is an underlying type model which has not yet been discovered!

Class and Type: Exercises

- Q1: Design a type-consistent inheritance hierarchy (without deletions) for modelling the abstract behaviour of different kinds of COLLECTION, to include:

  STACK , QUEUE, DEQUEUE and SET

  What is it that unites the class of all COLLECTIONs?

- Q2: Some OO methods advocate the discovery of inheritance structures by identifying entities, listing their attributes and factoring out common attributes in local superclasses.

  Explain why this approach fails to guarantee type-consistent inheritance.

Types and Subtyping

Now, a look at the foundations of type theory.

- Types as sorts and carrier sets.
- Function signatures and axioms.
- Recursive types and subtypes.
- Subtyping for sets and subranges.
- Subtyping for functions and axioms.
- Subtyping for record types.

Algebraic approach to type modelling (cf Goguen). Advantage: you define *abstract types*, rather than *concrete* ones.

Type-Consistency

Is it possible to produce a type-consistent model of object-oriented classes?

- Can classes be made to conform to types?
- Can inheritance be made to conform to subtyping?

  "A type A is included in (is a subtype of) another type B when all the values of type A are also values of B" (Cardelli and Wegner, 1985).

Intuitively, a subtype must:

- bear structural similarity with its parent;
- respect *all* of its parent' s functions;
- behave in a similar way to its parent.

Need to define what an abstract type is, much more closely...

Types: Sorts and Carrier Sets

Initial idea is that all types are sets:

$x : T \Leftrightarrow x \in T$

This concept used to ' bootstrap' the first few abstract type definitions; $\Rightarrow$ Notion of *sorts* and *carrier sets*.

A *sort* (eg NATURAL or BOOLEAN) is:

"an uninterpreted identifier that has a corresponding carrier in the standard (initial) algebra" (Danforth and Tomlinson, 1988).

A *carrier set* is some concrete set of objects which you can use to model sorts.

$BOOL \triangleq \{true, false\}$ - finite set

$NAT \triangleq \{0, 1, 2, ... \}$ - infinite set

An *algebra* is a pair of a sort ($\approx$ carrier set) and a set of operations over elements of the sort (carrier):

$BOOLEAN \triangleq <BOOL, \{\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow\}>$

Types: Functions

However, it is too restrictive to model abstract types as concrete sets - consider:

SIMPLE_ORDINAL ≙ {0, 1, 2, ... }

SIMPLE_ORDINAL ≙ {a, b, c ... }

The type SIMPLE_ORDINAL can be modelled by a variety of carriers which have an ordering defined over them.

"Types are not sets" (Morris, 1973).

SIMPLE_ORDINAL is more precisely defined as the abstract type over which the functions *First()* and *Succ()* are meaningfully applied:

SIMPLE_ORDINAL ≙ ∃ ord . {
  First : → ord;
  Succ : ord → ord }

NB: ord is an existentially quantified variable awaiting the full definition of the type - to allow for recursion in the type definition.

Types: Axioms

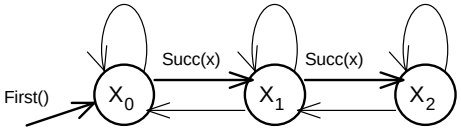
But this is still not enough - consider the possibility that:

Succ(1) → 1
Succ(b) → a

We need to constrain the semantics of operations using logic axioms:

∀x : SIMPLE_ORDINAL
  Succ(x) ≠ x
  Succ(x) ≠ First()

This, plus the *principle of induction*, is exactly enough to ensure that the type behaves like a SIMPLE_ORDINAL:



Types described with functions and axioms are *more general* and *more precise* than sets.

Types: Recursion

Do existential types exist? Problems with recursion in type definitions:

SIMPLE_ORDINAL ≙ ∃ ord . {
  First : → ord;
  Succ : ord → ord }

Analogy: Consider the recursive function:

add ≙ λa.λb. if b = zero then a
  else (**add** (succ a)(pred b))

This is merely an equation that *add* must satisfy:

- there is no guarantee that *add* exists;
- there may not be a unique solution.

cf $x^2 = 4 \Rightarrow x = 2 \mid x = -2$

Standard technique for dealing with recursion is to ' solve' the equation above using *fixed point analysis* (Scott, 1976).

- see example in Technical Appendix.

Subtypes: Partial Orders

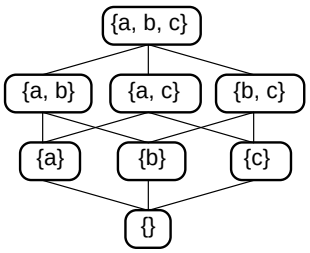
We may use *partial orders* (POs) to model types and subtypes.

- Example: any powerset forms a PO:

  S ≙ {a, b, c}

  P(S) ≙ {{}, {a}, {b}, {c}, {a, b}, {b, c}, {a, c}, {a, b, c}}

- Ordering relationship: ⊆ exists between (some) elements of the PO.



- Certain *complete* POs (CPOs) are called *ideals* and form a complete lattice under ⊆ .

Abstract Types: Exercises

- Q1: In mathematics, a *monoid* is an algebra <S, op, id> with certain properties, where

  S is the sort (≈ set) of elements;

  op : $S \times S \rightarrow S$ is an *associative* function taking a pair of elements back into the sort;

  id $\in$ S is the identity element for which

  (op id any) $\Rightarrow$ any.

  How many examples of monoids can you find in the standard data types provided in programming languages?

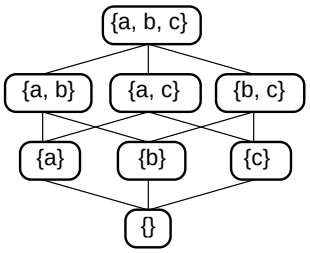
- Q2: Provide a functional and axiomatic specification for the abstract types STACK and QUEUE. How do they differ?

Subtyping Rule for Sets

Going back to our original intuition of types-as-sets:

$x : \tau \Leftrightarrow x \in \tau$

Since we can construct a CPO relating all our types (ie sets) in a lattice:



then we can assert that a *subtype* means the same thing as a *subset*:

$\sigma \subseteq \tau \Leftrightarrow \forall x\ (x \in \sigma \Rightarrow x \in \tau)$

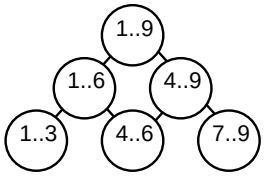
ie all the values (elements) of σ are also values (elements) of τ.

Subtyping Rule for Subranges

Type constructor for subranges: s..t

where s $\in$ NATURAL;
t $\in$ NATURAL;
s $\leq$ t;

The set of all subranges has useful partial order $\subseteq$ among its elements:



Subtyping for Subranges (Rule 1)

$s..t \subseteq \sigma..\tau \Leftrightarrow s \geq \sigma \wedge t \leq \tau$

henceforward, we shall use the (weaker) implication and denote this using:

$$\frac{s \geq \sigma,\ t \leq \tau}{s..t \subseteq \sigma..\tau}$$

Functions: Generalisation

Type constructor for functions:

name : domain $\rightarrow$ codomain

Use subranges to model types in the domain and codomain of λ-expressions:

f : 2..5 $\rightarrow$ 3..6
$\equiv \lambda x\ .\ x + 1$ (f 3) $\Rightarrow$ 4

Consider how simple types generalise: 3 has type 3..3 and also the type of any supertype:

$3 : (3..3) \subseteq (3..4) \subseteq (2..4) \subseteq (2..5)$

Now consider how function types generalise:

$g : (2..5 \rightarrow 4..5) \subseteq (2..5 \rightarrow 3..6)$

because it maps its domain to naturals between 4 and 5 (and hence between 3 and 6); however

$h : (3..4 \rightarrow 3..6) \not\subset (2..5 \rightarrow 3..6)$

because it only maps naturals between 3 and 4 (and hence not between 2 and 5) to its codomain.

Subtyping Rule for Functions

The inclusion (ie generalisation) rule for function types therefore demands that

- the domain shrinks; but
- the codomain expands:

$f : (2..5 \rightarrow 3..6) \subseteq (3..4 \rightarrow 2..7)$

Subtyping for Functions (Rule 2)

$$\frac{s \supseteq \sigma,\ t \subseteq \tau}{s \rightarrow t \subseteq \sigma \rightarrow \tau}$$

This means that for two functions $A \subseteq B$ if

- A is *covariant* with B in its result type; ie (result A) $\subseteq$ (result B)
- A is *contravariant* with B in its argument type; ie (argument A) $\supseteq$ (argument B)

This is an important result for OOP.

Axioms: Specialisation

Consider that STACK and QUEUE have indistinguishable functional specifications:

$SQ \triangleq \exists\ sq\ .\ \{push : ELEMENT \times sq \rightarrow sq;$
$pop : sq \nrightarrow sq;$
$top : sq \nrightarrow ELEMENT\}$

without the appropriate constraints to ensure

- LIFO property of STACKs
- FIFO property of QUEUEs.

Imagine an unordered collection receiving an element - we may assert the constraint:

$\forall e : ELEMENT, \forall c : COLLECTION$
$e \in add(e,c)$

Now, if we want to consider a STACK as a kind of COLLECTION, we may assert an additional axiom to enforce ordering:

$\forall e : ELEMENT, \forall s : STACK$
$e \in add(e,s);$
$top(add(e,s)) = e$

which is a *more stringent* constraint.

Subtyping Rule for Axioms

A constraint is *more stringent*, if it rules out more objects from a set:

$\{ \forall x \mid \alpha_{STACK} \} \subseteq \{ \forall y \mid \beta_{COLLECTION} \}$

and this is the subtyping condition.

Constraints can be made more stringent by:

- adding axioms
- modifying axioms

A *modified axiom* is one which necessarily entails the original one; here we can assert:

$(top(add(e,s)) = e) \Rightarrow (e \in add(e,s))$

Subtyping for Axioms (Rule 3)

$$\frac{\alpha_1, \ldots \alpha_k \Rightarrow \beta_1, \ldots \beta_k}{\{ \forall x \mid \alpha_1, \ldots \alpha_k, \ldots \alpha_n \} \subseteq \{ \forall y \mid \beta_1, \ldots \beta_k \}}$$

This means that for two constraints $A \subseteq B$ if

- A has *n-k* more axioms than B
- The first *k* axioms in A entail those in B

Objects as Records

Simple objects may be modelled as records whose components are labelled functions (Cardelli and Wegner, 1985):

- access to stored attributes represented using nullary functions;
- modification to stored attributes represented by constructing a new object.

Non-recursive records:

INT_POINT $\triangleq$ {
x : $\rightarrow$ INTEGER; y : $\rightarrow$ INTEGER }

Recursive records (assumes $\exists$ *pnt*):

CART_POINT $\triangleq$ Rec pnt . {
x : $\rightarrow$ INTEGER; y : $\rightarrow$ INTEGER;
moveBy : INTEGER x INTEGER $\rightarrow$ pnt;
equal : pnt $\rightarrow$ BOOLEAN }

- assumes objects are *applied to* labels to select functions: (obj label).

Subtyping Rule for Records

Consider that objects of type:

COL_POINT ≙ { x : → INTEGER;
y : → INTEGER; color : → INTEGER }

may also be considered of type INT_POINT, since they respect all INT_POINT' s functions;

Consider also that objects of type:

NAT_POINT ≙ {
x : → NATURAL; y : → NATURAL }

are a subset of all INT_POINTs defined by:

{ ∀p ∈ INT_POINT | p.x ≥ 0, p.y ≥ 0 }

Subtyping for Records (Rule 4)

$$\frac{\sigma_1 \subseteq \tau_1, \ldots \sigma_k \subseteq \tau_k}{\{ x_1{:}\sigma_1, \ldots x_k{:}\sigma_k, \ldots x_n{:}\sigma_n \} \subseteq \{ x_1{:}\tau_1, \ldots x_k{:}\tau_k \}}$$

This rule says that for two records A ⊆ B if

- A has *n-k* more fields than B;
- the first *k* fields of A are subtypes of those in B (could be the identical type).

Inheritance as Subtyping

Combining the above Rules 1 - 4, we get:

Any two related classes, modelled as records containing sets of functions, are in a subtype relation A ⊆ B if:

- extension: A adds monotonically to the functions inherited from B (Rule 4); and
- overriding: A replaces some of B' s functions with subtype functions (Rule 4); and
- restriction: A is more constrained than B (Rule 3) or a subrange/subset of B (Rule 1).

A function may only be replaced by another if:

- contravariance: arguments are more general supertypes (Rule 2); and therefore preconditions are weaker (Rule 3);
- covariance: the result is a more specific subtype (Rule 2); and therefore postconditions are stronger (Rule 3).

Subtyping: Exercises

- Q1: Is class B ⊆ class A? Explain why, or why not. (NB - here, model classes as records and attributes as nullary functions).

```
class A                    class B inherit A
attributes                 attributes
  x : INTEGER;               b : BOOLEAN;
  y : INTEGER;             methods
methods                      foo : A → D;
  foo : B → C;               bar : B → D;
end.                       end.
```

```
class C                    class D inherit C
attributes                 attributes
  o : A;                     o : B;
methods                    methods
  baz : A → C                baz : B → D
end.                       end.
```


Subtyping versus Type Recursion

We can now describe inheritance in terms of subtyping; but soon will see how this is not adequate to capture inheritance with polymorphism.

- Exploring the subtyping model of inheritance
- Polymorphism introduces type recursion
- Subtyping breaks down: positive recursion
- Subtyping breaks down: negative recursion
- F-bounded quantification
- Polymorphic subtyping