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

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

  $\forall i,j,k : \text{INTEGER}$
  $\quad plus(i,j) = plus(j,i)$
  $\quad plus(plus(i,j),k) = plus(i,plus(j,k))$
  $\quad plus(i,0) = i$
  $\quad ...$

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).

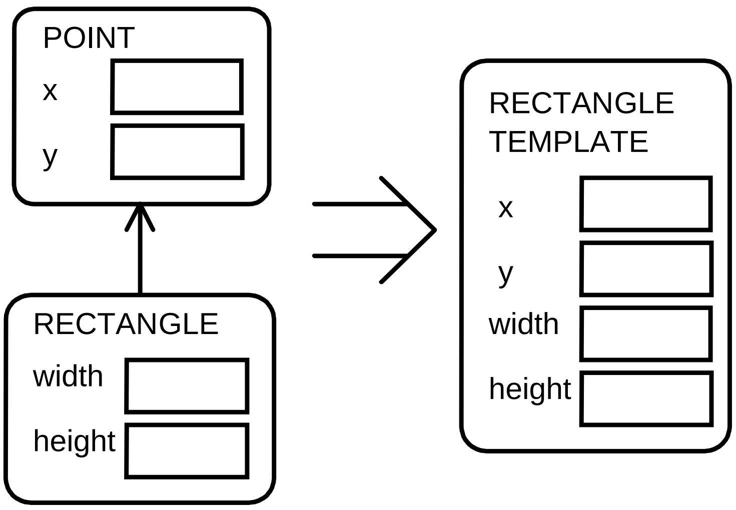
Class seen as a *unit of specification* (ambitious viewpoint).

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:

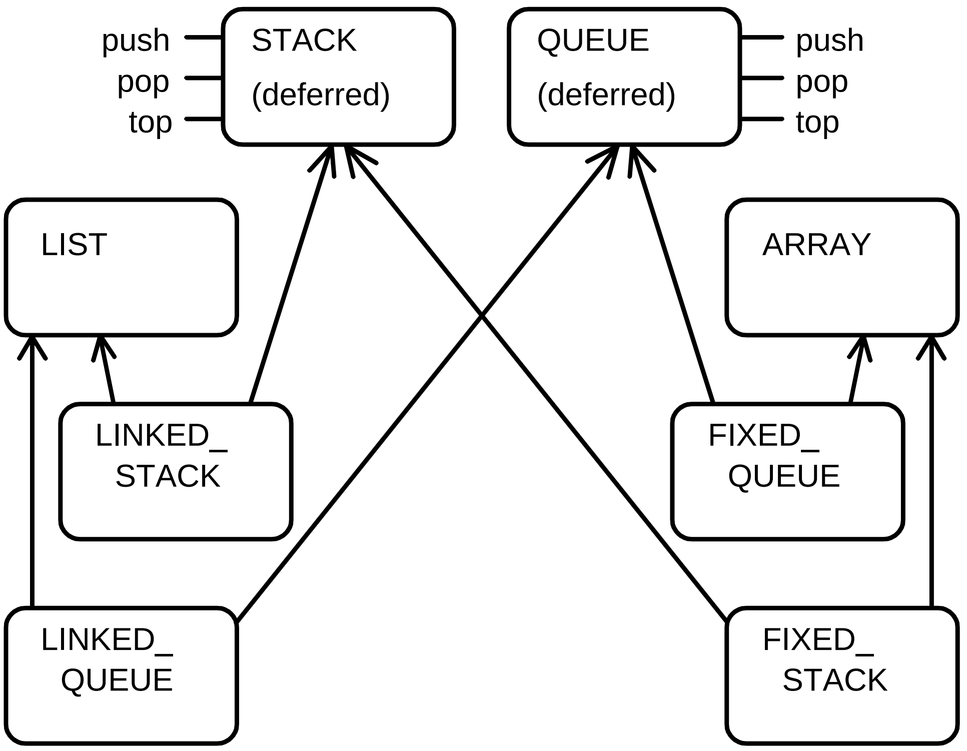


but is a RECTANGLE really a kind of POINT? Odd taxonomy.

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

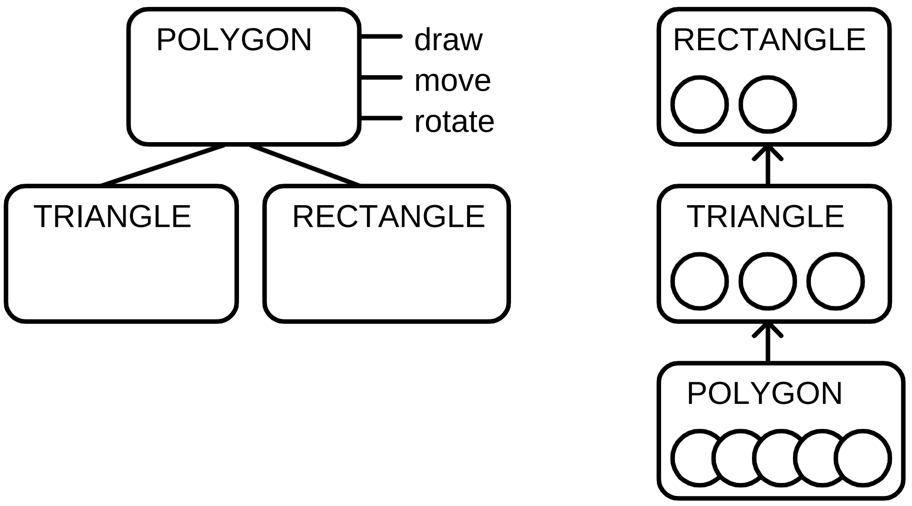
but is this formally correct? ...

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.



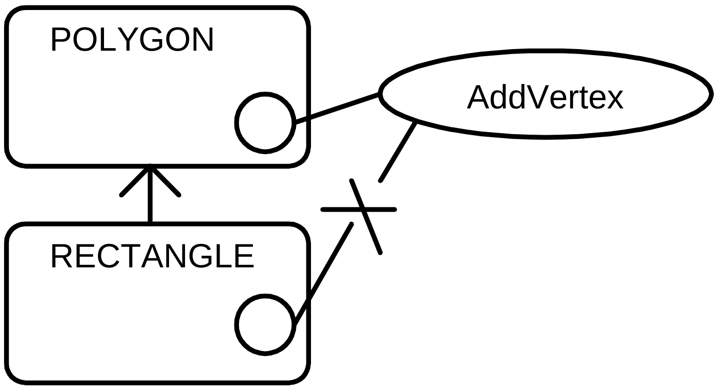
eg in Smalltalk (Goldberg and Robson, 1983):

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.

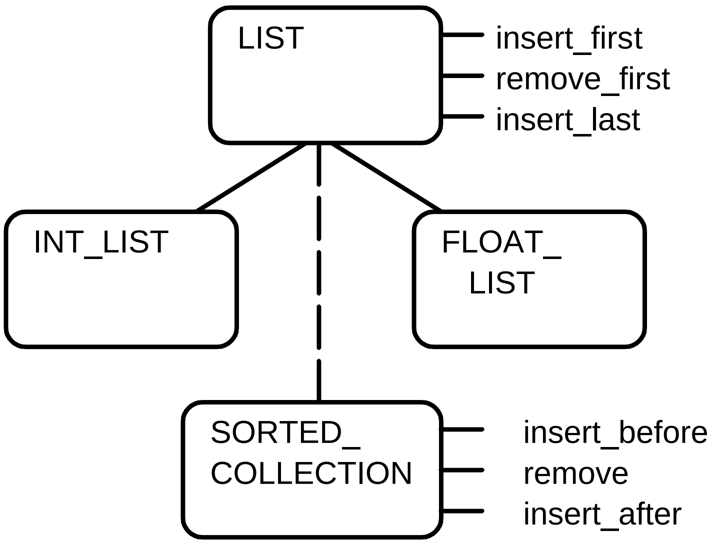
...but a RECTANGLE can't add to its vertices!!

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:

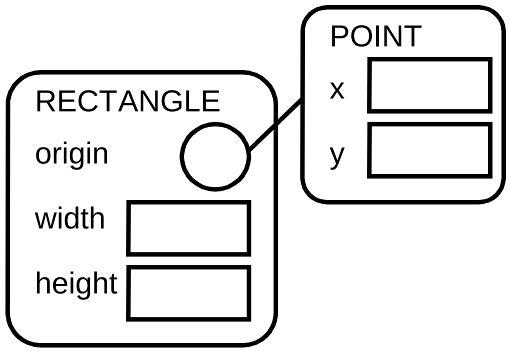
- *private* inheritance - subclass only inherits implementation of its parent;
- *public* inheritance - subclass also inherits specification of its parent.

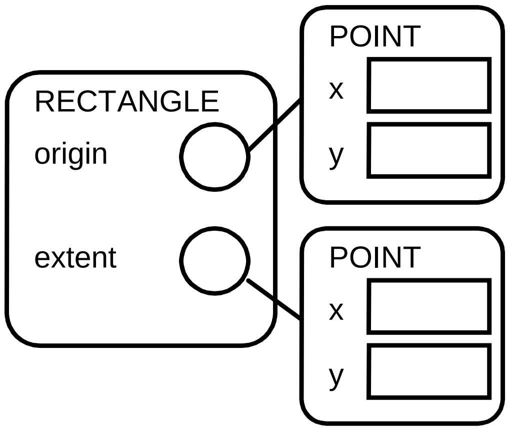
An INT_LIST is type-compatible with LIST. A SORTED_COLLECTION is not.

Class/Type Independence

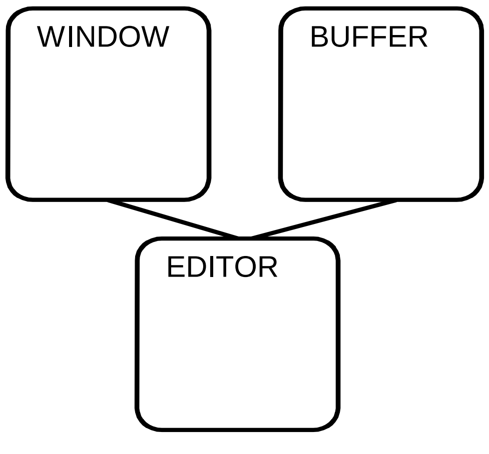
Objects *seem* to have class and type independently (Snyder, 1987):

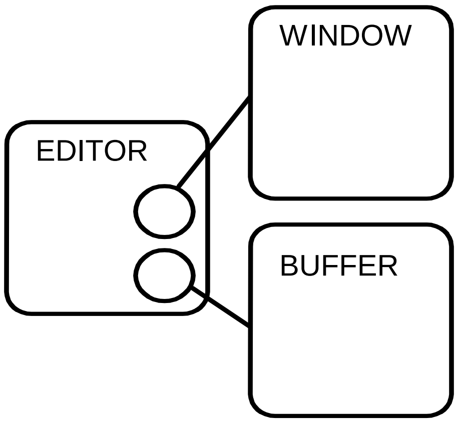
M:1 mappings from class hierarchies into type hierarchies, due to multiple concrete representations:





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

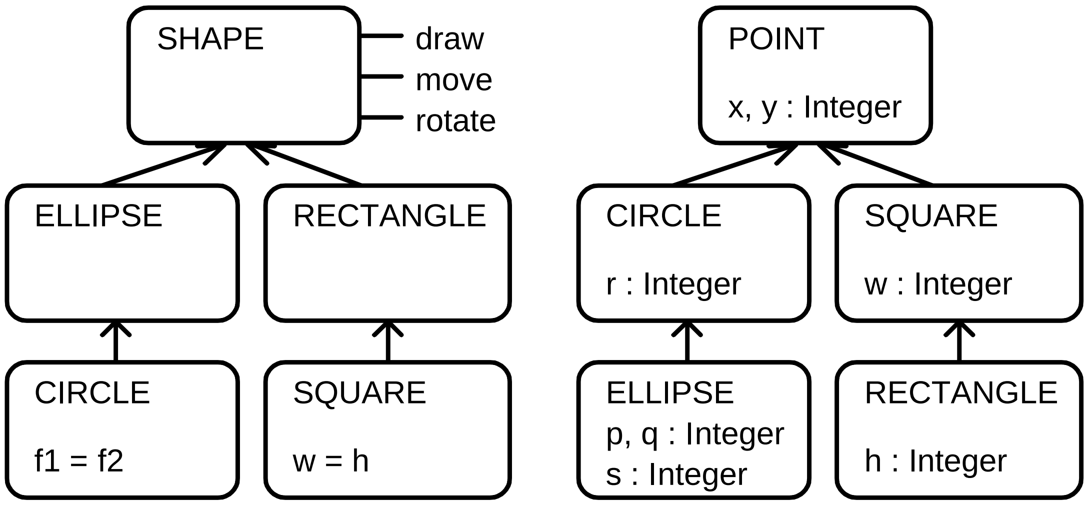




Separate Sharing of Class and Type

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 $\triangleq$ {0, 1, 2, ... }

SIMPLE_ORDINAL $\triangleq$ {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

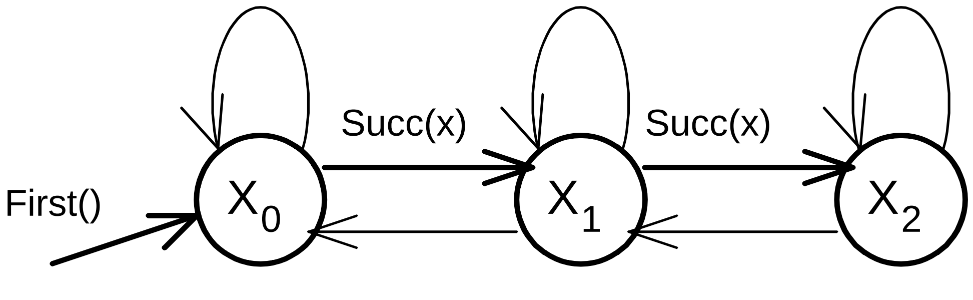
But this is still not enough - consider the possibility that:

Succ(1) $\rightarrow$ 1
Succ(b) $\rightarrow$ a

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

$\forall$x : SIMPLE_ORDINAL
Succ(x) $\neq$ x
Succ(x) $\neq$ First()
Succ(x) = Succ(y) $\Leftrightarrow$ x = y

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

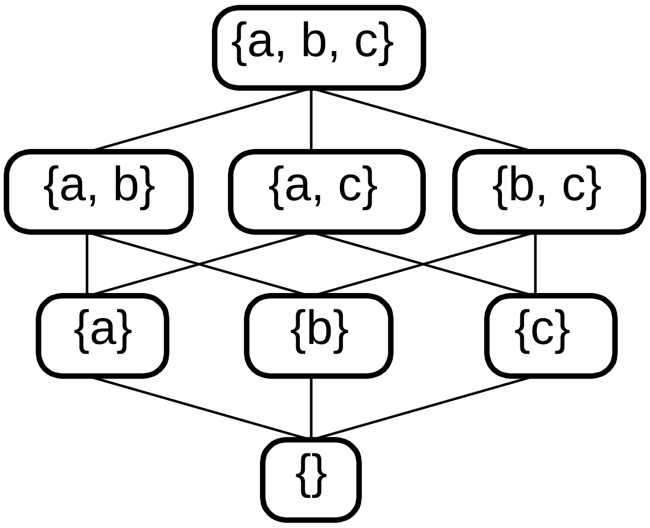
We may use *partial orders* (POs) to model types and subtypes.

- Example: any powerset forms a PO:

  S $\triangleq$ {a, b, c}

  $\mathcal{P}$(S) $\triangleq$ {{}, {a}, {b}, {c}, {a, b}, {b, c}, {a, c}, {a, b, c}}

- Ordering relationship: $\subseteq$ exists between (some) elements of the PO.



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

Abstract Types: Exercises

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

  S is the sort ($\approx$ 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?

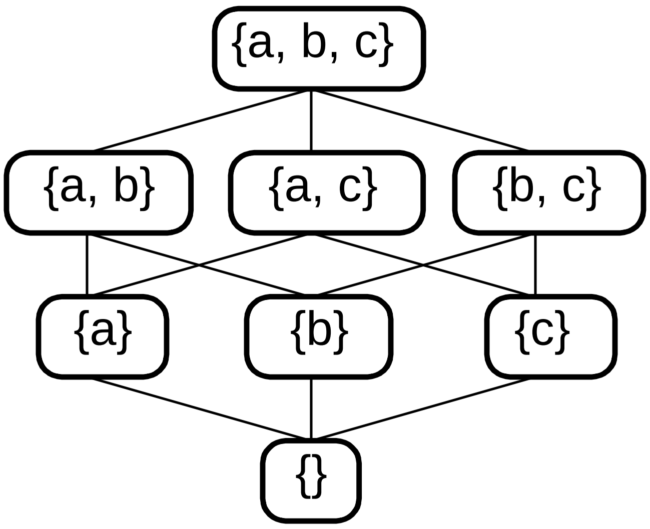
- Q2: Provide a functional and axiomatic specification for the abstract types STACK and QUEUE. How do they differ?

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*:

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

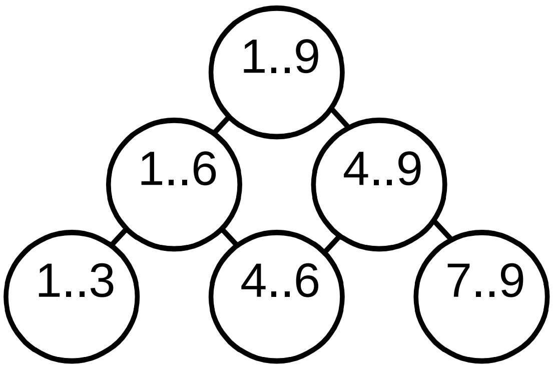
ie all the values (elements) of σ are also values (elements) of τ.

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)

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

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:

$$\begin{aligned} SQ \triangleq \exists\, sq\, .\, \{&push : ELEMENT \times sq \rightarrow sq;\\ &pop : sq \nrightarrow sq;\\ &top : sq \nrightarrow ELEMENT\} \end{aligned}$$

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:

$$\begin{aligned} &\forall e : ELEMENT,\ \forall c : COLLECTION\\ &\quad e \in add(e,c) \end{aligned}$$

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

$$\begin{aligned} &\forall e : ELEMENT,\ \forall s : STACK\\ &\quad e \in add(e,s);\\ &\quad top(add(e,s)) = e \end{aligned}$$

which is a *more stringent* constraint.

Subtyping Rule for Axioms

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

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

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:

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

<u>Subtyping for Axioms (Rule 3)</u>

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

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 $\triangleq$ { x : $\rightarrow$ INTEGER;
y : $\rightarrow$ INTEGER; color : $\rightarrow$ 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 $\triangleq$ {
x : $\rightarrow$ NATURAL; y : $\rightarrow$ NATURAL }

are a subset of all INT_POINTs defined by:

$\{ \forall p \in \text{INT\_POINT} \mid p.x \geq 0, p.y \geq 0 \}$

<u>Subtyping for Records (Rule 4)</u>

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

This rule says that for two records A $\subseteq$ 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 $\subseteq$ 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 $\subseteq$ 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

Conforming to Subtyping

How do existing inheritance schemes measure up to the subtyping model?

Smalltalk:

- variable argument lists ☒ and types ☒
- derailment (≈ selective inheritance) ☒

C++:

- fixed argument lists ☑ and types ☑
- linked export through public inheritance ☑

Eiffel:

- covariant result ☑ and arguments ☒
- undefinition, orthogonal export (≈ selective inheritance) ☒
- weakened preconditions ☑ strengthened postconditions ☑

Trellis:

- covariant result ☑ contravariant arguments ☑

Failure to Conform

What happens in languages that do not conform to strict subtyping?

- Starting with two basic classes in Eiffel:

```
class SAMPLE
feature
        data : INTEGER;
        magnitude : INTEGER is
        do              -- absolute value of sample
                if data < 0
                        then Result := - data
                        else Result := data
        end
end; -- SAMPLE


class POWER_SAMPLE inherit SAMPLE
feature
        power : INTEGER is
        do              -- square of sample
                Result := data * data
        end
end; -- POWER_SAMPLE
```

Breaking a Rule

- Eiffel allows *covariant* argument redefinition - strictly breaking a subtyping rule;
- at first, this looks reasonable enough...

```
class SIGNAL
feature
    rectify(arg : SAMPLE) : INTEGER is
    do          -- simple strategy
        Result := arg.magnitude
    end
end; -- SIGNAL


class POWER_SIGNAL inherit SIGNAL
feature
        -- redefining with new argument
    rectify(arg : POWER_SAMPLE)
                        : INTEGER is
    do          -- more sophisticated strategy
        Result := arg.power
    end
end; -- POWER_SIGNAL
```

Type Failure

- ...until you have a routine which expects a SIGNAL and is given a POWER_SIGNAL:

```
local
        sam : SAMPLE;
        sig : SIGNAL;
        pow : POWER_SIGNAL;
do
        sam.Create;
        pow.Create;
                -- statically correct if pow ⊆ sig
        sig := pow;
                -- statically correct for SIGNAL
        sig.rectify(sam);
                -- but invokes POWER_SIGNAL's
                -- rectify(sam) which in turn invokes
                -- sam.power!!!  Runtime failure!!!
end; -- some routine
```

Failure to conform to subtyping can result in programs being passed as type-correct, but hiding run-time type failure (Cook,1989).

Inheritance with Polymorphism

Even a subtype-conformant language may fail to express what we want. Consider the type:

$\text{OBJECT} \triangleq \text{Rec obj . \{}$
$\quad \text{identity} : \rightarrow \text{obj};$
$\quad \text{equal} : \text{obj} \rightarrow \text{BOOLEAN \}}$

So, *identity* and *equal* are functions with the types:

$\text{identity} : \text{OBJECT} \rightarrow (\rightarrow \text{OBJECT})$

$\text{equal} : \text{OBJECT} \rightarrow (\text{OBJECT} \rightarrow \text{BOOLEAN})$

But what should happen when we apply these functions polymorphically to some inheriting class, such as POINT? We would like:

$\text{identity} : \text{POINT} \rightarrow (\rightarrow \text{POINT})$

$\text{equal} : \text{POINT} \rightarrow (\text{POINT} \rightarrow \text{BOOLEAN})$

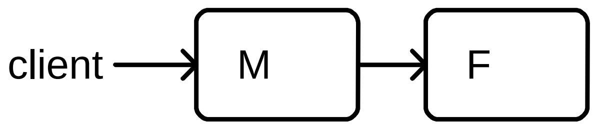
- the types of polymorphic functions need to change under inheritance;
- the desired type modifications *seem* to violate subtyping rules.

Mutual Type Recursion

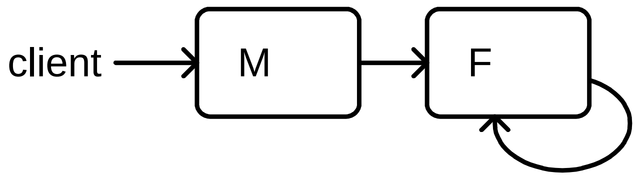
Inheritance with polymorphism is analogous to mutual type-recursion (Cook & Palsberg,1989):

Consider a function F and a derived (modified) version M which depends on F...

Direct derivation - encapsulation is preserved:-

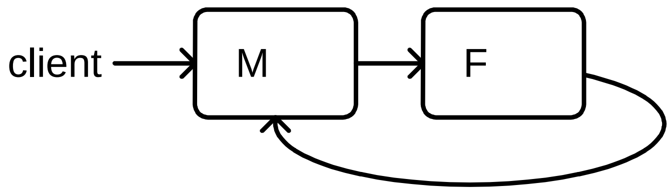


Naive derivation from recursive structure:-



- In the naive case, the modification only affects external clients, not recursive calls.

Derivation analogous to inheritance:-



- In the case of polymorphic inheritance, self-reference in the original class must be changed to refer to the modification.

Polarity in Type Expressions

When we examine our inheritance-as-subtyping model in the context of polymorphism, different things go wrong depending on the location of recursive type variables.

Analogy with *polarity* in logic (Canning, Cook, Hill, Olthoff & Mitchell 1989):

Definition: Positive and Negative Polarity

In the type expression:

$$\sigma \rightarrow \tau$$

σ appears negatively and τ positively.

Positive Type Recursion:

- occurs when the recursive type variable appears on the RHS of the $\rightarrow$ constructor.

Negative Type Recursion:

- occurs when the recursive type variable appears on the LHS of the $\rightarrow$ constructor.

Positive Type Recursion

Consider classes in a simple screen graphics package. We would like a *move* function:

$$\begin{aligned}&\text{MOVEABLE} \triangleq \text{Rec mv . \{}\\ &\quad \text{move : INTEGER} \times \text{INTEGER} \rightarrow \text{mv \}}\end{aligned}$$

to apply polymorphically to all descendants of MOVEABLE, such as SQUARE and CIRCLE.

However *move* does **not** have the type:

$$\begin{aligned}&\text{move : } \forall t \subseteq \text{MOVEABLE . } t \rightarrow\\ &\quad (\text{INTEGER} \times \text{INTEGER} \rightarrow t)\end{aligned}$$

but rather the type:

$$\begin{aligned}&\text{move : } \forall t \subseteq \text{MOVEABLE . } t \rightarrow\\ &\quad (\text{INTEGER} \times \text{INTEGER} \rightarrow \text{MOVEABLE})\end{aligned}$$

- Whenever we *move* SQUAREs or CIRCLEs we always obtain an object of exactly the type MOVEABLE (we lose type information).
- The algebra does not force the function's result type to mirror its polymorphic target.
- Cannot cope with positive type recursion.

Negative Type Recursion

Consider now that we would like a < function

$$\text{COMPARABLE} \triangleq \text{Rec } cp \,.\, \{ \\ \quad < : cp \rightarrow \text{BOOLEAN} \}$$

to apply polymorphically to all descendants of COMPARABLE such as INTEGER and CHARACTER, which inherit the < operation.

Now, the function < does **not** have the type:

$$< : \forall t \subseteq \text{COMPARABLE} \,.\, t \rightarrow \\ \quad (t \rightarrow \text{BOOLEAN})$$

but rather the type:

$$< : \forall t \subseteq \text{COMPARABLE} \,.\, t \rightarrow \\ \quad (\text{COMPARABLE} \rightarrow \text{BOOLEAN})$$

- Whenever we compare INTEGERs, the < function always expects an argument of exactly the type COMPARABLE.
- The algebra does not force < to compare operands of the same type.
- Cannot cope with negative type recursion.

Subtyping Breaks Down

If we unroll the inherited type definitions for CHARACTER or INTEGER, we can force the function < to accept the types we desire:

CHARACTER $\triangleq$ Rec ch . { ...; print : ch $\rightarrow$;
  < : ch $\rightarrow$ BOOLEAN; ... }

By explicit redefinition, < now has the type

< : $\forall$t $\subseteq$ CHARACTER . t $\rightarrow$
  (CHARACTER $\rightarrow$ BOOLEAN)

To ensure CHARACTER $\subseteq$ COMPARABLE, we need to obtain subtyping among functions in the pair of records:

{...; < : CHARACTER $\rightarrow$ BOOLEAN; ... }
  $\subseteq$ { < : COMPARABLE $\rightarrow$ BOOLEAN }

requiring COMPARABLE $\subseteq$ CHARACTER in turn by contravariance!

- CHARACTER $\subseteq$ COMPARABLE cannot be derived using the rules of subtyping unless in fact CHARACTER $\equiv$ COMPARABLE.
- The subtyping model breaks down. polymorphism.

Inheritance not Subtyping

Mathematically, subtyping involves *bounded universal quantification* (Cardelli & Wegner, 1985):

$\forall t \subseteq T . \sigma(t)$ where $T \equiv \text{Rec } r . F(r)$

For inheritance to conform to subtyping, we would either need:

- no recursive types; or
- no polymorphic functions.

Instead, we desire a typing construct which permits full type recursion:

$\forall t \subseteq F[t] . \sigma(t)$

and this is called *function-bounded quantification* (Canning, Cook, Hill, Olthoff & Mitchell 1989).

In this model (Cook, Hill & Canning, 1990):

- classes are not types;
- inheritance is not subtyping.

Deriving Typing Functions

(Canning, Cook, Hill, Olthoff & Mitchell 1989) obtain typing functions which have the recursive properties we desire.

Consider the polymorphic *move* function.

Working backwards, we seek the condition on a type *t* so that for any variable *x : t* we can derive "$\triangleright$" that *x.move(1, 1)* is also of type *t*.

$x : t \triangleright x.move(1, 1) : t$ { by assumption }

Using a type rule for function application:

$$\text{APP} \quad \frac{f : \sigma \to \tau ,\ \ v : \sigma}{(f\ v) : \tau}$$

$x : t \triangleright x.move : (INTEGER \times INTEGER \to t)$

Using a type rule for record selection:

$$\text{SEL} \quad \frac{r : \{ \alpha_1 : \tau_1, ..., \alpha_n : \tau_n \}}{r.\alpha_i : \tau_i} \qquad i \in 1..n$$

$x : t \triangleright x : \{ move : INTEGER \times INTEGER \to t \}$

F-Bounded Quantification

$x : t \triangleright x : \{ \text{move} : \text{INTEGER x INTEGER} \rightarrow t \}$

is the minimal constraint on the record type of *x*. Using the subtyping rule, we can introduce more specific record types τ such that:

$$\frac{\tau \subseteq \{ \text{move} : \text{INTEGER} \times \text{INTEGER} \rightarrow t \}}{x : t \ \triangleright \ x : \tau}$$

Since the type τ does not occur in any other assumption, we may simplify using $\{ t / \tau \}$ to the requirement

$$t \subseteq \{ \text{move} : \text{INTEGER} \times \text{INTEGER} \rightarrow t \}$$

which cannot be proved without additional assumptions.

Expressing this condition as $t \subseteq$ F-Moveable[t], where F-Moveable[t] is a *typing function:*

$$\begin{aligned} &\text{F-Moveable}[t] \triangleq \{ \\ &\quad \text{move} : \text{INTEGER} \times \text{INTEGER} \rightarrow t \} \end{aligned}$$

it is clear that this condition fits the format for the kind of quantification we desire.

Polymorphic Types

Classes are now modelled as *typing functions* containing bound parameters. These describe spaces of possible types, hence they are polymorphic type descriptions:

F-Moveable[t] $\triangleq$ {
move : INTEGER $\times$ INTEGER $\rightarrow$ t }

F-Comparable[t] $\triangleq$ { < : t $\rightarrow$ BOOLEAN }

Actual types are obtained by the application of these typing functions to specific types, whereby the parameter is replaced:

F-Moveable[SQUARE] = { move :
INTEGER $\times$ INTEGER $\rightarrow$ SQUARE }

F-Moveable[CIRCLE] = { move :
INTEGER $\times$ INTEGER $\rightarrow$ CIRCLE }

F-Comparable[CHARACTER] = {
< : CHARACTER $\rightarrow$ BOOLEAN }

F-Comparable[INTEGER] = {
< : INTEGER $\rightarrow$ BOOLEAN }

Polymorphic Subtyping

If we unroll the type definitions for SQUARE or CIRCLE we get:

$\text{SQUARE} \triangleq \text{Rec sqr . \{ ...;}$
$\quad \text{move : INTEGER} \times \text{INTEGER} \rightarrow \text{sqr; ... \}}$

$\text{CIRCLE} \triangleq \text{Rec cir . \{ ...;}$
$\quad \text{move : INTEGER} \times \text{INTEGER} \rightarrow \text{cir; ... \}}$

thereby demonstrating that:

$\text{SQUARE} \subseteq \text{F-Moveable[SQUARE]}$
$\text{CIRCLE} \subseteq \text{F-Moveable[CIRCLE]}$ ...etc

which is precisely what we want.

Note that the most general type satisfying the condition is the type over whose body we abstracted:

$\text{MOVEABLE} \subseteq \text{F-Moveable[MOVEABLE]}$

but we do not have any other simple subtyping relationships:

$\text{SQUARE} \not\subset \text{MOVEABLE}$
$\text{CIRCLE} \not\subset \text{MOVEABLE}$

Type Recursion: Exercises

- Q1: Given the following classes, explain the type of

  x.fetch;

  when x is of type CAR. What is the type of a CAR's home?

```
class VEHICLE              class GARAGE
attributes             attributes
   home : GARAGE;   holds : VEHICLE;
methods                    methods
   park is                    put(v : VEHICLE) is
     home.put(self)              holds := v
   end;                       end;
   fetch : VEHICLE is         get : VEHICLE is
     return home.get()           return holds
   end;                       end;
end.                       end.


              class CAR inherit
                    VEHICLE
              end.
```

Implications for Language Design

The problem is that we have been treating classes as though they were actual types, when in fact they are type constructors.

- Distinguishing *class* and *type* semantics
- Classes as type constructors
- Type recursion, substitution, rebinding
- Bound and free type parameters
- Homogenous and heterogenous types
- Full recursive capture of type
- Solving the type failure problem

Classes: Semantic Ambiguity

So, what is a class? In most languages, class identifiers are used ambiguously.

A class may denote either:

- polymorphic: a space of possible types (the bounded, but possibly infinite set of descendent classes derived from it); or
- monomorphic: a specific type (the type of new objects created from the class template);

Implicitly we adopt the polymorphic *class* interpretation:

- when designing open-ended class libraries using inheritance;
- when assigning types to polymorphic variables;

but may switch to the monomorphic *type* interpretation:

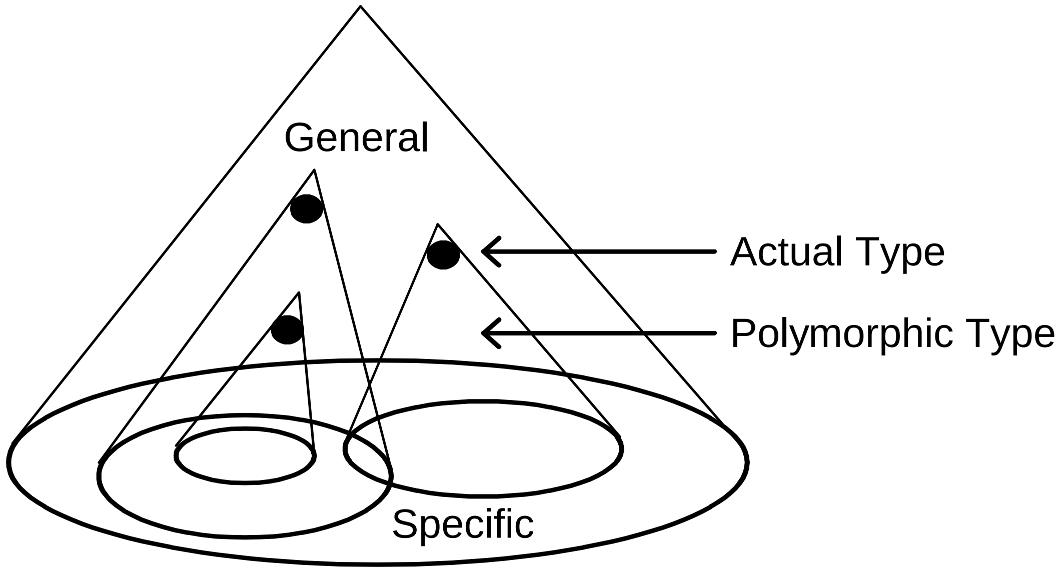
- when creating new objects.

Type Space and Fixed Points

Let us try to distinguish *class* and *type*:

Imagine the space of all possible recursive abstract types (RATs) - this type domain:

- contains RATs corresponding to the powerset of all functions;
- forms a complete lattice under the cpo $\subseteq$.



Classes define *bounded, closed volumes* in the type domain - these are true polymorphic types - with a *least fixed point* at the apex - this is the most general actual type satisfying the bound.

Classes as Type Constructors

Compare with other familiar mechanisms for handling polymorphic types:

eg: the type constructor for ARRAYs contains *explicit type parameters* denoting 'unknown' or 'incomplete' parts of the type:

ARRAY [$\forall$s $\subseteq$ SUBRANGE] OF [$\forall$t $\subseteq$ TOP]

Classes are also type constructors, containing an *implicit type parameter* which:

- abstracts over the entire class body;
- corresponds to the 'known' parts of the type;
- must permit full recursive instantiation with any suitable type satisfying the class bound.

F-OBJECT[t] $\triangleq$ { identity : $\rightarrow$ t;
  equal : t $\rightarrow$ BOOLEAN }

So, a class is a higher-order construct in type theory, cf parameterised types in ML (Milner, 1978).

Distinguishing Class and Type

Object-oriented languages need to distinguish actual (albeit general) *types*, eg:

```
m : MOVEABLE;
```

from F-bounded polymorphic *classes*,

m : $\forall t \subseteq$ F-Moveable[t];

which we might represent syntactically by introducing an *explicit* parameter M in:

```
m : MOVEABLE[M];
```

such that it is clear when expressions have a recursively instantiated polymorphic type:

```
class MOVEABLE[M]
{  position : POINT;
   move (x, y : INTEGER) : MOVEABLE[M];
}
```

- *position* returns a *type* POINT;
- *move* returns a *class* M of types constrained by the F-bound MOVEABLE;
- cf *T* and *T'Class* in Ada9X.

Type Recursion: Inheritance

Parameterising the self-type gives us a simple mechanism - unification - for expressing type recursion under inheritance (Simons *et al*, 1994):

```
class MOVEABLE[M]
{  position : POINT;
   move (x, y : INTEGER) : MOVEABLE[M];
}
```

Now, CIRCLE inherits from MOVEABLE:

```
class CIRCLE[C] : MOVEABLE[M]
{  radius : INTEGER;
}
```

- By unification, we have C' = C $\cap$ M, and the resulting constraint CIRCLE[C'] since $\forall$t CIRCLE[t] $\subseteq$ MOVEABLE[t].
- inherited *move* now returns a *class* C' of types constrained by the F-bound CIRCLE.

This captures the self-type recursion in:

- x : like Current;   -- in Eiffel
- self class new.   "in Smalltalk"

Type Substitution: Genericity

Records may contain other parameterised components, eg the generalised coordinate:

```
class NUM_POINT[P]
{  x, y : NUMBER[N];
   set (p, q : NUMBER[N]) : NUM_POINT[P];
}
```

where N is a parameter to be replaced by any type satisfying NUMBER[ ].

A simple scheme for type substitution permits the generation of many useful types:

```
intPoint : NUM_POINT { INTEGER/N };

realPoint : NUM_POINT { REAL/N };
```

This captures:

- parametric polymorphism in ML;
- generic packages in Ada;
- *constrained* genericity in Eiffel.

Note that: NUM_POINT ≡ NUM_POINT[P]
{ NUM_POINT/P }

Type Restriction: Rebinding

Type parameters can be *rebound* through unification to derive classes with extra semantics:

```
class LIST[L]
{  head : OBJECT[O];          ...any object
   tail : LIST[L];
   add (x : OBJECT[O]) : LIST[L];
}
```

Sorted lists are like lists except that their elements must be comparable:

```
class SORTED_LIST[S] :
         LIST[L] { COMPARABLE[C/O] }
{  add (x : COMPARABLE[C]) :
         SORTED_LIST[S];     ...redefined add
}
```

- By unification, we have C' = C $\cap$ O, and the resulting constraint COMPARABLE[C'] since $\forall$t COMPARABLE[t] $\subseteq$ OBJECT[t].
- inherited *head* now returns a *class* C' of types constrained by the F-bound COMPARABLE.

Tied Parameters

By linking the instantiation of type parameters, we can force polymorphic functions to accept *homogenous* types of argument:

```
class COMPARABLE[C]
{  lessThan (x : COMPARABLE[C]) :
                               BOOLEAN }

3.lessThan(4)                ...true
4.5.lessThan(3.2)            ...false
3.lessThan(4.5)        ...static type error
```

Separate, non-linked parameters permit polymorphic functions to accept *heterogenous* argument types:

```
class COMPARABLE[C]
{  lessThan (x : COMPARABLE[D]) :
                               BOOLEAN }

3.lessThan(4.5)        ...true
```

This captures aspects of:

- parametric polymorphism in ML;
- template functions in C++.

Static Type Resolution

Parameter unification and type substitution allow polymorphic functions to acquire static types at compile time:

```
class MOVEABLE[M]
{  position : POINT;
   move (x, y : INTEGER) : MOVEABLE[M];
}

class SQUARE[S] : MOVEABLE[M]
{  side : INTEGER;
}
```

- move : SQUARE[S] $\rightarrow$
  (INTEGER $\times$ INTEGER $\rightarrow$ SQUARE[S])

because of tying through unification S $\cap$ M;

```
s : SQUARE;
s.move(3, 4);
```

- move : SQUARE $\rightarrow$
  (INTEGER $\times$ INTEGER $\rightarrow$ SQUARE)

because of implicit substitution due to SQUARE $\equiv$ SQUARE[S] { SQUARE/S }.

Free Parameters

Parameters have the effect of linking the type-instantiations of parts of polymorphic structures:

```
m : MOVEABLE[X];
s : SQUARE;  c : CIRCLE;
m := s;                  ...{ SQUARE/X }
m.move(3, 4);
m := c;                  ...static type error
m.move(6, 8);
```

- Within the same scope, a tied parameter cannot be instantiated twice.

The symbol ? is a *free type parameter*, not linked to any other part of a structure:

```
n : MOVEABLE[?];
n := s;             ...{ SQUARE/?#1 }
n.move(3, 4);
n := c;             ...{ CIRCLE/?#2 }
n.move(6, 8);
```

- Within the same scope, a free parameter may be instantiated many times. Dynamic type checks may be needed.

Dynamic Type Resolution

Free parameters lead to situations where dynamic type checks may be needed:

```
class COL_SQUARE[C] : SQUARE[S]
{  shade : COLOR;
}

m : MOVEABLE[X]; n : SQUARE[?];
c : COL_SQUARE;  s : SQUARE;

m := c;                      ...{ COL_SQUARE/X }
n := s;                  ...{ SQUARE/? }
m := n.move(6, 8);   ...static type error?
```

In this case, *move* has the type:

- move : SQUARE[?] $\rightarrow$
  (INTEGER $\times$ INTEGER $\rightarrow$ SQUARE[?])

but since *m* has the type:

- m : MOVEABLE[X] { COL_SQUARE/X }
  $\equiv$ m : COL_SQUARE

the *move* expression is only type-correct if it returns a COL_SQUARE. Other languages cannot detect this.

Homogenous and Heterogenous Collections

Tied and free parameters distinguish the types of homogenous and heterogenous collections:

```
class LIST[L]
{  head : OBJECT[O];          ...any object
   tail : LIST[L];            ...like self
   add (x : OBJECT[O]) : LIST[L];
}
```

- The tail of the list must be in the same type L as the current list;
- So, the head of the tail of the list must be in the same type O as the head of the current list, etc...

```
class LIST[H]
{  head : OBJECT[?];          ...any object
   tail : LIST[H];            ...like self
   add (x : OBJECT[?]) : LIST[H];
}
```

- Here, even the head and the item added need not be in the same types ?#1, ?#2.

Mutual Type Recursion

There is a mutual recursive relationship between type parameters in recursive data types:

```
class LIST[L]
{  head : OBJECT[O];          ...any object
   tail : LIST[L];            ...like self
   add (x : OBJECT[O]) : LIST[L];
}

intList : LIST { INTEGER/O }:
```

The head of *intList* has the type:

- head : OBJECT[O] { INTEGER/O }
  $\equiv$ head : INTEGER

and the tail of *intList* has the type:

- tail : LIST[L] { LIST{INTEGER/O} / L }
  $\equiv$ tail : LIST { INTEGER/O }

ie modifications to the type of the head O affect the type of the tail L;

- cf *recursive capture* in BOPL (Palsberg & Schwartzbach, 1994).

Type Failure Solved: Type Error

Back to the Eiffel type failure problem:

```
class SIGNAL[S]
{   rectify (x : SAMPLE[X]) : INTEGER;   }

class POWER_SIGNAL[P] :
    SIGNAL[S] { POWER_SAMPLE[Y/X] }
{   rectify (y : POWER_SAMPLE[Y]) :
            INTEGER;   }...redefined rectify
```

First approach, using a tied parameter - type information propagated from *sig*:

```
sam : SAMPLE;                   ...created
sig : SIGNAL[T];
pow : POWER_SIGNAL;             ...created

sig := pow;         ...{ POWER_SIGNAL/T }
sig.rectify(sam);       ...static type error
```

because the type of *rectify* is now:

- rectify : POWER_SIGNAL $\rightarrow$ (POWER_SAMPLE[Y] $\rightarrow$ INTEGER)
- and { SAMPLE/Y } is a type error.

Type Failure Solved: Type Check

Second approach, using a free parameter - type information propagated from *sam*:

```
sam : SAMPLE;                ...created
sig : SIGNAL[?];
pow : POWER_SIGNAL;          ...created

sig := pow;        ...{ POWER_SIGNAL/? }
sig.rectify(sam);      ...{ SAMPLE/X }
```

Here, we need a dynamic type check, because the available types of *rectify* are:

- rectify : SIGNAL[S] $\rightarrow$
  (SAMPLE[X] $\rightarrow$ INTEGER)
- rectify : POWER_SIGNAL[P] $\rightarrow$
  (POWER_SAMPLE[Y] $\rightarrow$ INTEGER)

and { SAMPLE/X } yields the only solution:

- rectify : SIGNAL $\rightarrow$ (SAMPLE $\rightarrow$ INTEGER)

which is only type-correct if the object stored in *sig* is in fact of type SIGNAL.

Reference Material

The introductory material is still difficult for beginners, but may prove rewarding after the exposition of this tutorial.

The more advanced material provides much of the mathematical foundation for the arguments presented here and should only be handled by properly-trained mathematicians!

Bibliography

- P America (1990), 'Designing an object-oriented language with behavioural subtyping', *Proc. Conf. Foundations of Object-Oriented Languages*, 60-90.
- L Cardelli (1984), 'A semantics of multiple inheritance', in: *Semantics of Data Types, LNCS 173*, Springer Verlag, 51-68.
- L Cardelli and P Wegner (1985), 'On understanding types, data abstraction and polymorphism', *ACM Computing Surveys 17 (4)*, 471-521.
- P Canning, W Cook, W Hill, W Olthoff and J Mitchell (1989), 'F-bounded polymorphism for OOP', *Proc. Func. Prog. Langs. and Comp. Arch. 4th Int. Conf*, 273-280.
- P Canning, W Cook, W Hill and W Olthoff (1989), 'Interfaces for strongly-typed OOP', *Proc. OOPSLA-89*, 457-467.
- W Cook (1989), 'A proposal for making Eiffel type-safe', *Proc. ECOOP-89*, 57-72.

Bibliography

- W Cook, W Hill and P Canning (1990), 'Inheritance is not subtyping', *Proc. POPL-90*, 125-135.
- W Cook and J Palsberg (1989), 'A denotational semantics of inheritance and its correctness', *Proc. OOPSLA-89*, 433-443.
- S Danforth and C Tomlinson (1988), 'Type theories and OOP', *ACM Computing Surveys, 20 (1)*, 29-72.
- A Goldberg and D Robson (1983), *Smalltalk-80: the Language and its Implementation*, Addison-Wesley.
- D MacQueen, G Plotkin and R Sethi (1984), 'An ideal model for recursive polymorphic types', *Proc. POPL-84*, 165-174.
- B Meyer (1988), *Object-Oriented Software Construction*, Prentice-Hall.
- B Meyer (1992), *Eiffel: the Language*, Prentice-Hall.

Bibliography

- R Milne and C Strachey (1976), *A Theory of Programming Language Semantics*, Chapman and Hall.
- R Milner (1978), 'A theory of type polymorphism in programming', *J. Comp. and Sys. Sci. 17*, 348-375.
- J H Morris (1973), 'Types are not sets', *Proc. POPL-73*, 120-124.
- J Palsberg and M Schwartzbach (1994), *Object-Oriented Type Systems*, John Wiley.
- R Raj and H Levy (1989), 'A compositional model for software reuse', *Proc. ECOOP-89*, 3-24.
- C Schaffert, T Cooper, B Bullis, M Kilian and C Wilpolt (1986), 'An introduction to Trellis/Owl', *Proc. OOPSLA-86*, 9-16.
- D Scott (1976), 'Data types as lattices', *SIAM J. Computing, 5 (3)*, 523-587.

Bibliography

- A Simons and A Cowling (1992), 'A proposal for harmonising types, inheritance and polymorphism for OOP', *Report CS-92-13, Dept. Comp. Sci, University of Sheffield*.
- A Simons (1993), *Introduction to Object-Oriented Type Theory*, *OOPSLA-93* and *ECOOP-93* Tutorials.
- A Simons, Low E-K and Ng Y-M (1994), 'An optimising delivery system for object-oriented software, *J. of Object-Oriented Systems, 1 (1)*.
- A Snyder (1986), 'Encapsulation and inheritance in OOP languages', *Proc. OOPSLA-86*, 38-45.
- A Snyder (1987), 'Inheritance and the development of encapsulated software components', in B Shriver and P Wegner, *Research Directions in OOP*, MIT Press, 165-188.
- B Stroustrup (1991), *The C++ Programming Language, 2nd Edn*., Addison-Wesley.

Technical Appendix

This appendix includes technical material on lambda calculus, domain theory and category theory ancilliary to the exposition of this tutorial.

- Fixed point analysis method
- Derivation of a recursive function
- Domain theory
- Category theory
- Semantics of recursive types
- Semantics of polymorphic classes

Recursion: Fixed Point Analysis

Approach to solving recursive equations:

- transform body into non-recursive form by replacing recursive call with λ abstraction:

  add $\triangleq$ λa.λb. if b = zero then a
  else (**add** (succ a)(pred b)) $\Rightarrow$

  ADD $\triangleq$ λ**f**.λa.λb. if b = zero then a
  else (**f** (succ a)(pred b))

- use this new function to generate the recursive version:

  add $\triangleq$ (ADD <some fn>)

- It so happens that what we really need is:

  add $\triangleq$ (ADD add)

- ie *add* is defined as a value which is unchanged by the application of *ADD*:
- such a value is called a *fixed point* of *ADD*.

Recursion: Fixed Point Finder

We have transformed the task of finding a recursive solution for *add* into finding *fixed points* for *ADD*.

- There might be many such fixed points;
- *Under certain conditions*, it is possible to define the *least fixed point* of any function using the fixed point finder, Y.
- Y has the property that:

  $$f = (\mathrm{Y}\ F) \Leftrightarrow (F\ f) = f$$

- Here is a definition of Y. Note how it also is not recursive, but does contain delayed self-application:

  $$\mathrm{Y} \triangleq \lambda f.(\lambda s.(f\ (s\ s))\ \lambda s.(f\ (s\ s)))$$

λ Calculus Reduction Rules:

$(\lambda x.x\ a) \Rightarrow a$

$((\lambda x.\lambda y.(x\ y)\ a)\ b) \Rightarrow (\lambda y.(a\ y)\ b) \Rightarrow (a\ b)$

$(f\ a\ b) \equiv ((f\ a)\ b)$

Recursion: Derivation of *add*

```
ADD ≙ λg.λa.λb. if b = zero then a
            else (g (succ a)(pred b))

Y ≙ λf.(λs.(f (s s)) λs.(f (s s)))
```

```
add ≙ (Y ADD)                                 {ADD / λf}

⇒   (λs.(ADD (s s))                           {(s s) / λg}
     λs.(ADD (s s)))

⇒   (λs.λa.λb. if b = zero then a             {... / λs}
            else ((s s)(succ a)(pred b))
     λs.(ADD (s s)))

⇒   λa.λb. if b = zero then a
            else ((λs.(ADD (s s)) λs.(ADD (s s)))
                       (succ a)(pred b))

⇒ ... ⇒       {... / λs; (succ a) / λa; (succ b) / λb}

⇒   λa.λb. if b = zero then a
         else (if (pred b) = zero then (succ a)
            else ((λs.(ADD (s s)) λs.(ADD (s s)))
                       (succ (succ a))(pred (pred b)))
... etc
```

Types: Domain Theory

Finding fixed point solutions to existential types requires *certain conditions* - denotational semantics of λ calculus (Scott, 1976) needs *domain theory*.

- V is the *domain* of all computable values, ie

  $V \triangleq$ BOOLEAN + NATURAL +
  $[V \times V] + [V \rightarrow V]$.

- A *complete partial order* (cpo) relationship is constructed among some sets of values in V.

- Certain sets of values are used as carriers for types - can solve recursive equations using set-theoretic interpretation.

'Useful' carrier sets known as *ideals*, which have the following properties:

- downward closed under cpo;

- consistently closed under cpo;

on the domain V (Danforth & Tomlinson, 1986).

Links with Category Theory

One semantic interpretation of *F-bounded quantification* relies on Category Theory.

- A *category* is a collection of abstract objects with similar structure and behaviour,

  cf all objects conforming to some type.

- Structure-preserving maps, called *morphisms*, $f : x \to y$, exist from object to object in a category,

  cf correspondences between different representations of objects within a type.

- A morphism f with an inverse, $g : y \to x$, results in two *isomorphic* objects $x \cong y$,

  cf two objects with identical type.

- The *initial object* in each category has a single morphism extending to every other object,

  cf the most abstract denotation of all objects conforming to some type.

Ψ-Algebra Semantics

- Morphism-preserving maps, called *functors*, $\psi : C \to D$, exist from category to category,

  cf polymorphic inheritance which maps behaviour for one type into behaviour for another type (with "more structure").

- An *endofunctor* is a ψ which maps from a category into itself, ie $f : (\psi\ t) \to t$,

  cf recursive construction of a type (with "more structure", in the same category).

- A *ψ-algebra* is the category of pairs <t, f>, where $f : (\psi\ t) \to t$ are morphisms among a recursively constructed type,

  cf category of all objects in a recursive type.

- An *initial ψ-algebra* is the solution to the equation $(\psi\ t) \cong t$, a fixed point of ψ,

  cf the most abstract denotation of a recursive type.

- Since $(\psi\ t) \cong t$, the inverse $g : t \to (\psi\ t)$ must also exist.

Quantification over Ψ-Coalgebras

The *dual* of a category theory construct is one in which "arrows are reversed":

- morphisms are replaced by their inverse;
- initial objects become terminal objects.

The *dual* of a ψ-algebra is a *ψ-coalgebra* or category of pairs <t, g>, with $g : t \rightarrow (\psi\ t)$.

- Both the *initial ψ-algebra* and *terminal ψ-coalgebra* satisfy $t \cong (\psi\ t)$.

When we use F-bounded quantification, we say $\forall t \subseteq F[t]$

- which implies a map $g : t \rightarrow (\psi\ t)$, ie
- quantification over pairs <t, g> or some family of ψ-coalgebras.

Since any recursive type Rec t . F[t] may be regarded as a *particular* ψ-coalgebra,

F-bounded quantification is over a category whose objects are "generalisations" of the recursive type Rec t . F[t].