Introduction to Object-Oriented Type Theory

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 '93 Tutorials
Washington DC, September 1993

Overview

- Motivation
- Classes and Types
- Abstract Data Types
- Subtyping and Inheritance
- Recursion and Polymorphism
- Classes as Type Spaces
- Implications for Language Design
- Reference Material

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 of viewpoints:
  "strong versus weak inheritance".
- Conflict of viewpoints:
  "specification versus implementation".

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

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

  $\forall$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:

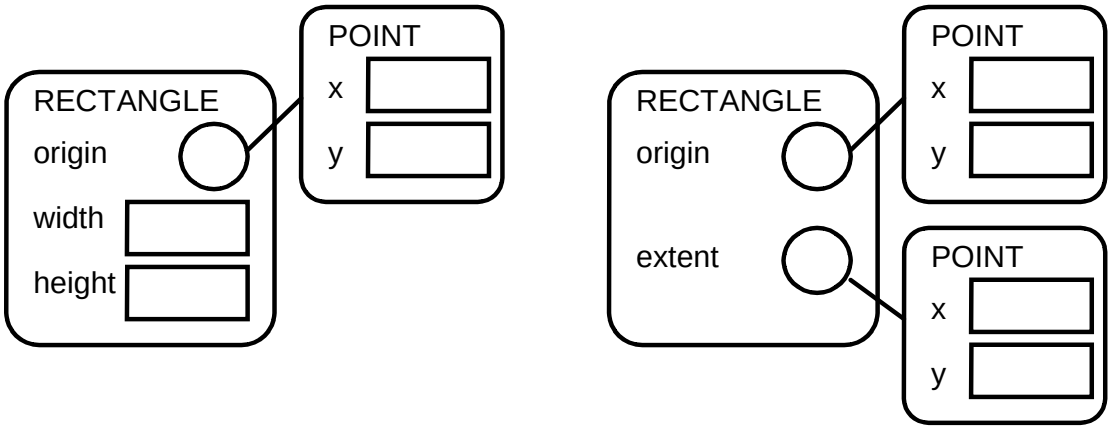
- adding new behaviours (adding method signatures);
- making behaviours more concrete (implementing/re-implementing methods);
- restricting set of objects (subclassing).

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

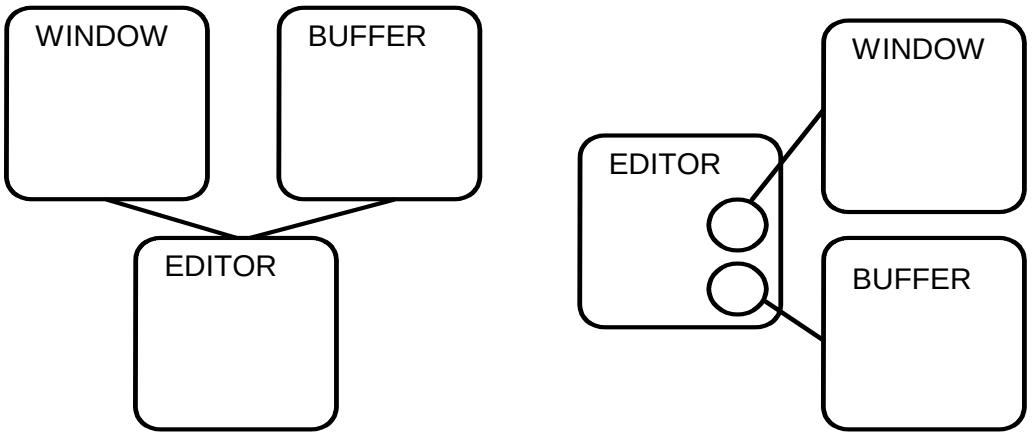
Class/Type Independence

Objects have class and type independently (Snyder, 1986); this demonstrated by:

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:

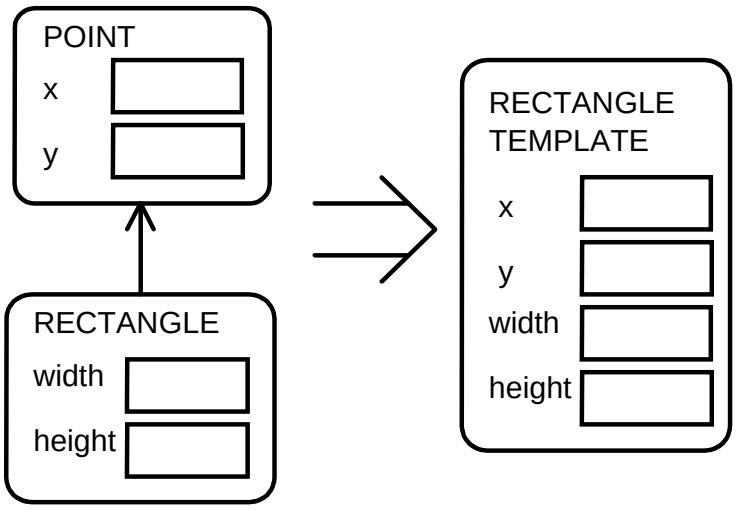


Convenience Viewpoint

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

- decoupling of class from type (can reason separately), as in Emerald;
- 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:



Ambitious Viewpoint

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!

$\Rightarrow$ Class and type are directly related notions.

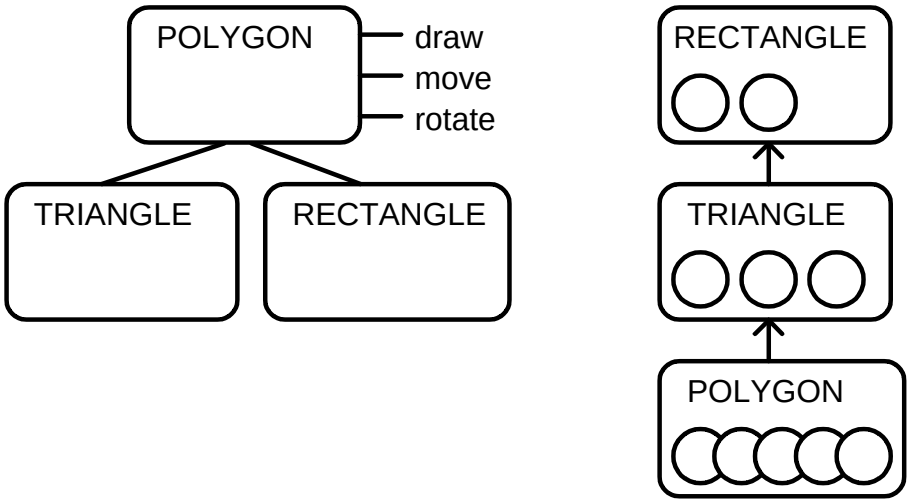
- Is it correct to treat classes as types?
- What usage of classes is type-consistent?

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.



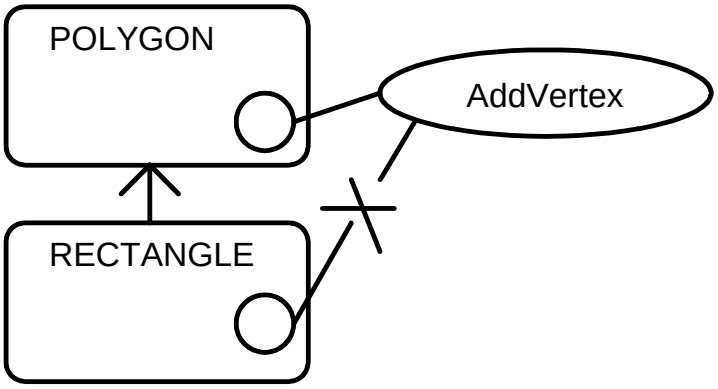
Maximising reuse of storage for corners of figures {origin, extent, ... nth vertex} leads to crazy type hierarchies.

Type-consistent inheritance - allow only certain kinds of implementation sharing.

Creeping Implementation

Clash of ambitious/convenience views:

Selective inheritance: introduced through orthogonal export rules; undefinition rules (eg in Eiffel)



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

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!!

Inheritance: Exercises

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

  ALBATROSS (which soars, mainly)

  PENGUIN (which swims, mainly)

  OSTRICH (which runs, mainly)

  What is it that unites the class of all birds?

- 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.

Abstract Data Types

Now, a look at the foundations of type theory.

- Types as sorts and carrier sets.
- Types defined with function signatures.
- Types defined with logic axioms.
- Recursion: fixed point analysis.
- Recursion: ideals and Scott domains.

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

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.

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

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

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

$BOOLEAN \equiv \langle BOOL, \{\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow\}\rangle$

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

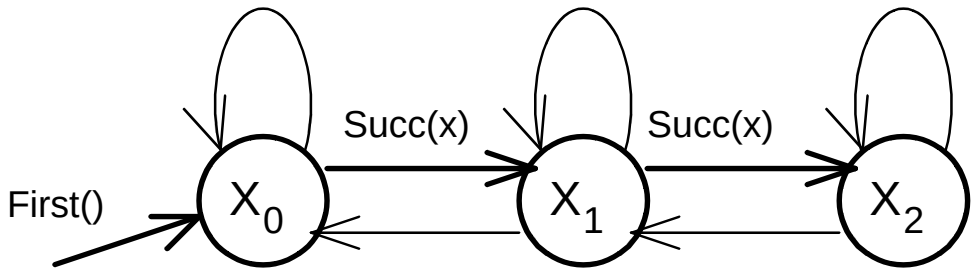
But this is still not enough - consider the possibility that:

Succ(1) $\rightarrow$ 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 is exactly enough to ensure that the type behaves like a SIMPLE_ORDINAL:



Abstract types defined in terms of operations with axioms are both more general and more precise than the types-as-sets view.

Types: Recursion

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

SIMPLE_ORDINAL $\equiv \exists$ ord . {
  First : $\rightarrow$ ord;
  Succ : ord $\rightarrow$ ord }

Analogy: Consider the recursive function:

add $\equiv \lambda a.\lambda b.$ 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 (Scott, 1976).

Recursion: Fixed Point Analysis

Approach to solving recursive equations:

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

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

  ADD ≡ 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 ≡ (ADD <some fn>)

- It so happens that what we really need is:

  add ≡ (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:

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

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

  $V \equiv \text{BOOLEAN} + \text{NATURAL} + [V \times V] + [V \rightarrow V]$.

- 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. What do these properties mean?

Types: Ideals

Example: the powerset of natural numbers is an ideal. Can *approximate* NAT and P(NAT) with finite subsets:

NAT = {0, 1, 2}

P(NAT) = {{}, {0}, {1}, {2}, {0,1}, {1,2}, {0,2}, {0,1,2}}

- Downward closed: if {1,2} is in the type (set), then so are its approximations {1}, {2} and {} which are all 'less than' {1,2} under the cpo $\subseteq$.
- Consistently closed: if an approximation to the type (subset) is {{0}, {1,2}} then its least upper bound {0,1,2} is also in the type.

  Here, LUB(S) $\equiv \forall x,y \in S\ \exists z \in S\ (x \subseteq z \wedge y \subseteq z)$

Two important results from using *ideals* (MacQueen et al., 1984):

- the set of all types (ideals) becomes a complete lattice under $\subseteq$;
- recursive type equations have solutions.

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: Is the set of NATURAL numbers an ideal? Explain why, or why not.

- Q3: Provide a functional and axiomatic specification for the abstract type STACK.

Inheritance and Subtyping

At first glance, inheritance looks very similar to subtyping; both are kinds of partial order relationship.

- Subtyping Rule for Subranges
- Subtyping Rule for Functions
- Subtyping Rule for Axioms
- Subtyping Rule for Records

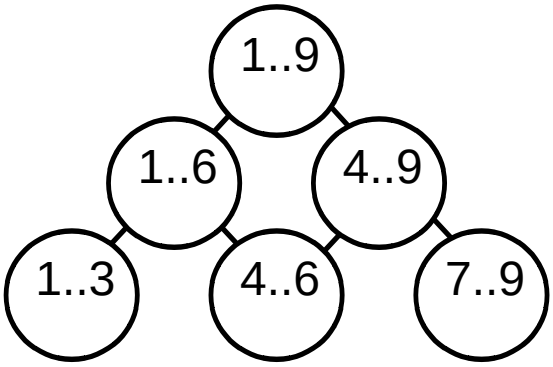
> "A type A is included in (is a subtype of) another type B when all the values of type A are also values of B" (Cardelli and Wegner, 1985).

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 is an ideal; useful partial order $\subseteq$ among elements allowing the construction of a *subtype graph:*



Subtyping for Subranges (Rule 1)

$s..t \subseteq \sigma..\tau$ **iff** $s \geq \sigma$ **and** $t \leq \tau$

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:

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

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:

$$\{ \forall x \in \text{STACK} \} \subseteq \{ \forall y \in \text{COLLECTION} \}$$

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:

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

Subtyping for Axioms (Rule 3)

$$\frac{\alpha_1 \Rightarrow \beta_1, \ldots \alpha_k \Rightarrow \beta_k}{\{ \alpha_1, \ldots \alpha_k, \ldots \alpha_n \} \subseteq \{ \beta_1, \ldots \beta_k \}}$$

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 (could be identical).

Objects as Records

Simple objects may be modelled as records whose components are a set of labelled functions representing methods:

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

Non-recursive records:

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

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

CART_POINT $\equiv$ 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:

$$\text{COL\_POINT} \equiv \{\ x : \rightarrow \text{INTEGER};\ \ y : \rightarrow \text{INTEGER};\ \ \text{color} : \rightarrow \text{INTEGER}\ \}$$

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

Consider also that objects of type:

$$\text{NAT\_POINT} \equiv \{\ x : \rightarrow \text{NATURAL};\ \ y : \rightarrow \text{NATURAL}\ \}$$

are a subset of all INT_POINTs defined by:

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

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 $A \subseteq B$ 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).

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
attributes
   x : INTEGER;
   y : INTEGER;
methods
   foo : B → C;
end.
```

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

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

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

Polymorphism and Type Recursion

We can now describe inheritance in terms of subtyping; but soon will see how this is not enough.

- Inheritance considered as subtyping
- Polymorphism introduces type recursion
- Polarity in type expressions
- Subtyping breaks down: positive recursion
- Subtyping breaks down: negative recursion

The type model we have introduced cannot yet handle the kind of type recursion introduced by inheritance with polymorphism.

Inheritance as Subtyping

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).

Many current OOLs violate these constraints. However, even this is not sufficient...

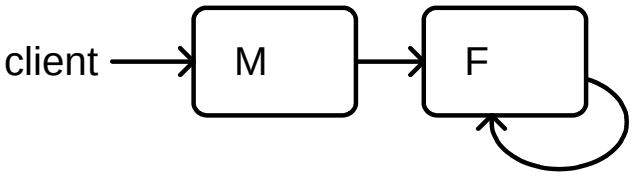
Polymorphism and Type Recursion

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...

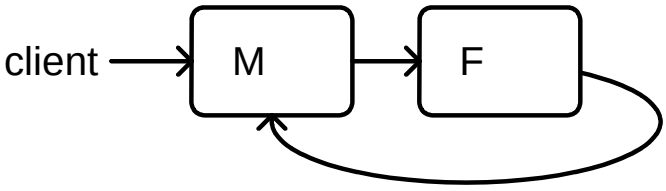
Direct derivation - encapsulation is preserved:-

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:

$$\text{MOVEABLE} \equiv \text{Rec mv} \,.\, \{ \text{move} : \text{INTEGER} \times \text{INTEGER} \rightarrow \text{mv} \}$$

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

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

$$\text{move} : \forall t \subseteq \text{MOVEABLE} \,.\, t \rightarrow (\text{INTEGER} \times \text{INTEGER} \rightarrow t)$$

but rather the type:

$$\text{move} : \forall t \subseteq \text{MOVEABLE} \,.\, t \rightarrow (\text{INTEGER} \times \text{INTEGER} \rightarrow \text{MOVEABLE})$$

- 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

$$\text{COMPARABLE} \equiv \text{Rec cp . } \{ < : \text{cp} \rightarrow \text{BOOLEAN} \}$$

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:

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

but rather the type:

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

- 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:

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

By explicit redefinition, < now has the type

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

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

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

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.
- A simple subtyping model breaks down in the presence of polymorphism.

Type Recursion: Exercises

- Q1: Given the following classes, explain how the expression

  x.park;

  involves mutual type recursion when x is of type CAR.

```
class VEHICLE
attributes
   home : GARAGE;
methods
   park is
     home.take(self)
   end;
end.

class PORT inherit
   GARAGE
attributes
   keep : CAR;
end.

class CAR inherit
      VEHICLE
attributes
      home : PORT;
   end.

   class GARAGE
   attributes
      keep : VEHICLE:
methods
      take(v : VEHICLE) is
         keep := v
      end;
   end.
```

Polymorphic Type Space

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

- Ambiguous semantics of class
- Type space and fixed points
- F-bounded quantification
- Polymorphic subtyping
- Relationship with Category Theory

Classes: Semantic Ambiguity

Problem stems from semantic ambiguity which we have not captured:

A class may denote either:

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

Implicitly we adopt the open semantics interpretation:

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

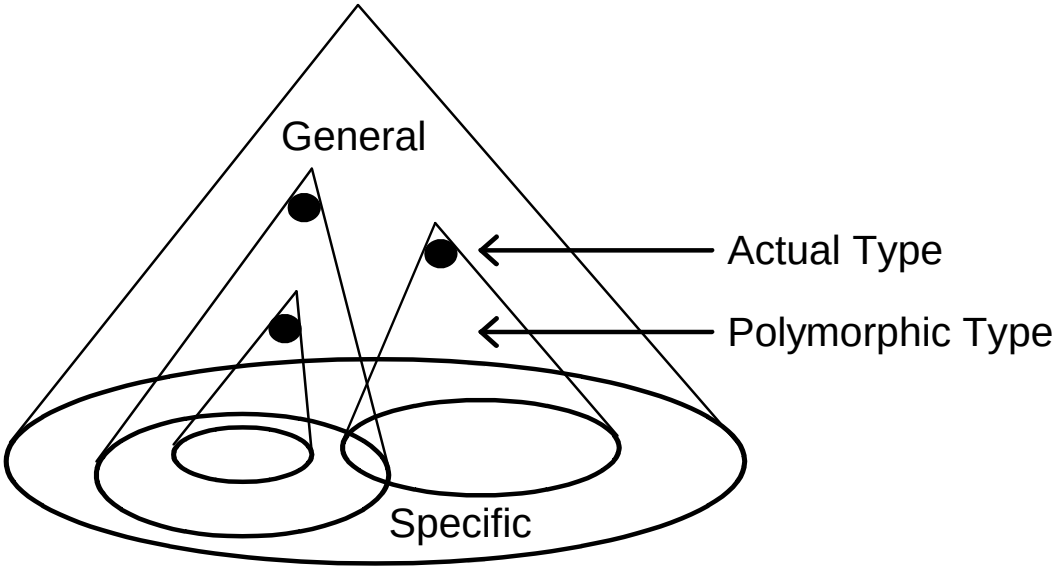
but may switch to the closed semantics interpretation:

- when creating new objects.

Type Space and Fixed Points

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

- is quantal in single functions (and axioms);
- 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

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

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

Classes are also 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.

Mathematically, we have been modelling polymorphism inadequately using *bounded universal quantification:*

$$\forall t \subseteq (\text{Rec } r \,.\, F(r)) \,.\, \sigma(t)$$

whereas we need a construct which permits type recursion in the constraining typing function itself:

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

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

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

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:*

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

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

Polymorphic Types

Using such typing functions which preserve recursion in type parameters, we can give an intuitively pleasing type to polymorphic spaces in the type domain:

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

F-Comparable[t] $\equiv$ { < : 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] $\equiv$ { move :
  INTEGER $\times$ INTEGER $\rightarrow$ SQUARE }

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

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

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

Polymorphic Subtyping

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

SQUARE $\equiv$ Rec sqr . { ...;
move : INTEGER $\times$ INTEGER $\rightarrow$ sqr; ... }

CIRCLE $\equiv$ Rec cir . { ...;
move : INTEGER $\times$ INTEGER $\rightarrow$ cir; ... }

thereby demonstrating that:

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

which is precisely what we want.

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

MOVEABLE $\subseteq$ F-Moveable[MOVEABLE]

but we do not have any other simple subtyping relationships:

SQUARE $\not\subset$ MOVEABLE
CIRCLE $\not\subset$ MOVEABLE

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 : $x \rightarrow y$, 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 : $y \rightarrow x$, 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*, ψ : C $\rightarrow$ 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 : (ψ t) $\rightarrow$ 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 : (ψ t) $\rightarrow$ 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 (ψ t) $\cong$ t, a fixed point of ψ,

  cf the most abstract denotation of a recursive type.

- Since (ψ t) $\cong$ t, the inverse g : t $\rightarrow$ (ψ 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 $\rightarrow$ (ψ t).

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

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

- which implies a map g : t $\rightarrow$ (ψ 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].

Type Spaces: Exercises

- Q1: What kind of semantics (open or closed) do we typically give to the following kinds of expression?

```
p : POINT;     ...
p.Create(2,4);

a : ARRAY[GRAPHIC];      ...
a.put(sqare1, 1);
a.put(circle3, 2);

x : like Current;  ...
x.Create(...);
```

- Q2: What morphisms exist, in the category of monoids, between <LIST, append, nil>, and <STRING, concat, ""> ?

- Q3: Describe the category theoretic relationship between the algebras <SEQUENCE, {append}> and <SEQUENCE, {append, length}> .

Implications for Language Design

If classes are considered properly as type constructors, then various considerations arise in relation to resolving the number and scope of implicit parameters (Simons and Cowling, 1992).

- Type recursion with self-reference.
- Single and multiple parameters.
- Homogenous and heterogenous collections.
- Combining polymorphism and genericity.

Type Recursion: Self-Reference

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

m : MOVEABLE;

from F-bounded polymorphic types,

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

which we might represent syntactically using an explicit parameter:

m : MOVEABLE(M);

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

move : M $\rightarrow$ (INTEGER $\times$ INTEGER $\rightarrow$ M);

The parameterised construct correctly handles the flexible typing of expressions such as:

x : like <anchor>;	in Eiffel
x class new;	in Smalltalk

in the same manner as (Milner, 1978) for the functional languages.

Single and Multiple Parameters

Object-oriented languages need to be able to distinguish polymorphic functions accepting *homogenous* arguments:

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

from polymorphic functions accepting *heterogenous* arguments:

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

where this might be represented syntactically using *two* type parameters:

```
class COMPARABLE(S|T)     ...

    < : S → (T → BOOLEAN)
```

permitting *different instantiations* of each parameter. Such a function would accept:

```
3 < 'a';        'b' < 70;
```

as legal expressions (possibly implemented over byte-size representations).

Homogenous and Heterogenous Collections

Object-oriented languages need to be able to distinguish properly the types of homogenous collections from heterogenous collections. For example,

d : LIST [GRAPHIC];

should denote homogenous lists of objects precisely of the (albeit general) type GRAPHIC; whereas:

d : LIST [GRAPHIC(T)]

should denote polymorphic, but homogenous lists of any type constrained by

$\forall t \subseteq$ F-Graphic[t]

and finally

d : LIST [GRAPHIC(S|T)]

should denote polymorphic, heterogenous lists of any types constrained by

$\forall s, t \subseteq$ F-Graphic[s, t] .

Combining Polymorphism and Genericity

The introduction of a bound parameter for polymorphic types brings these closer to generic constructs (eg in Ada and Eiffel).

Instead of a special generic syntax:

```
x : LIST [ANY(S|T)]
```

we could treat all recursively and iteratively defined collections as recursive record types:

```
class LIST(L)          ...
   item : ANY(S|T);
   next : L;           ...
```

and apply generic type constructors using the same syntax as for inheritance:

```
class ORDERED_LIST(O)
inherit LIST(O)
redefine
   item : COMPARABLE(C);        ...
```

here, obtaining a homogenous list of ordered items from a heterogenous list of any item.

Reference Material

The basic bibliography 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!

Basic Bibliography

- 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.
- S Danforth and C Tomlinson (1988), 'Type theories and OOP', ACM Computing Surveys, 20 (1), 29-72.
- 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.
- W Cook and J Palsberg (1989), 'A denotational semantics of inheritance and its correctness', Proc. OOPSLA-89, 433-443.
- P Canning, W Cook, W Hill and W Olthoff (1989), 'Interfaces for strongly-typed OOP', Proc. OOPSLA-89, 457-467.

Other References

- D Scott (1976), 'Data types as lattices', SIAM J. Computing, 5 (3), 523-587.
- 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.
- D MacQueen, G Plotkin and R Sethi (1984), 'An ideal model for recursive polymorphic types', Proc. POPL-84, 165-174.
- A Snyder (1986), 'Encapsulation and inheritance in OOP languages', Proc. OOPSLA-86, 38-45.
- W Cook, W Hill and P Canning (1990), 'Inheritance is not subtyping', Proc. POPL-89, 125-135.
- 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, UK.