Object-oriented languages

Advanced Compiler Construction
Michel Schinz – 2024–05–23
(parts based on Yoav Zibin's PhD thesis)

Object-oriented languages

A **class-based**, **object-oriented** (OO) language is one in which:

- all (or most) values are objects,
- objects belong to a class,
- objects encapsulate **state** (fields) and **behavior** (methods).

Two of the most important features of OO languages are:

1. inheritance, and
2. polymorphism.

Inheritance

Inheritance enables a class to inherit:

- all fields, and
- all methods

of its superclass.

Important: *inheritance is nothing but code copying!*

(It usually is implemented in a smarter way to avoid code explosion.)

Subtyping & polymorphism

In typed OO languages:

- classes (and interfaces) define **types**,
- these types are related by a **sub-typing** relation.

If a type T_1 is a subtype of a type T_2 (written $T_1 \sqsubseteq T_2$):

- T_1 has at least the capabilities of T_2 (informally),
- can use a value of type T_1 everywhere a value of type T_2 is expected (**inclusion polymorphism**).

Inclusion polymorphism:

- prevents the exact type of values to be known statically,
- therefore makes implementation challenging.

Subtyping ≠ inheritance

Inheritance and subtyping are not the same thing!

However, many languages ties them together since:

- every class defines a type,
- the type of a class is a subtype of the type(s) of its superclass(es).

This is a design choice, not an obligation!

Some languages allow them to be separated, e.g.:

- C++ has private inheritance (inheritance w/o subtyping),
- Java has interfaces (subtyping w/o inheritance).

"Duck typing"

"Dynamically typed" OO languages (Smalltalk, Ruby, Python, etc.) make the distinction between subtyping and inheritance obvious:

- inheritance is used only to reuse code,
- no notion of type even exists, so no subtyping!

An object can be used in a given context iff it has the right set of methods. The position of its class in the inheritance hierarchy plays no role whatsoever.

Polymorphism challenges

Inclusion polymorphism makes the following problems challenging:

1. **object layout** – arranging object fields in memory,
2. **method dispatch** – finding which concrete implementation of a method to call,
3. **membership test** – testing whether an object is an instance of some type.

OO problem #1: Object layout

Object layout

The **object layout problem**:

How should the fields of an object be arranged in memory so that they can be accessed efficiently?

Inclusion polymorphism makes this difficult because:

- ideally, a field defined in a type T should appear at the same offset in all subtypes of T,
- (i.e. the layout of different object types should be compatible).

Object layout example

```
class A {
  int x;
}
class B extends A {
  int y;
}
void m(A a) { System.out.println(a.x); }
```

at which position in a does x appear?

Case 1: single inheritance

Single inheritance

In OO languages like Java which:

- have only single inheritance,
- tie inheritance and subtyping,

the object layout problem can be solved easily as follows:

The fields of a class are laid out sequentially, starting with those of the superclass – if any.

This ensures that all fields belonging to a type T_1 appear at the same location in all values of type $T_2 \sqsubseteq T_1$.

Example

```
class A {
  int x;
}
```

layout for A

offset	field
0	x

```
class B extends A {
  int y;
}
```

layout for B

offset	field
0	x
1	y

```
void m(A a) { System.out.println(a.x); }
```

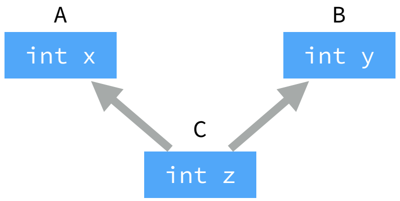
access position 0 of a

Case 2: multiple inheritance

Multiple inheritance

In a multiple inheritance setting, the object layout problem becomes much more difficult.

For example, in the following hierarchy, how should fields be laid out?



Unidirectional layout

If a standard, unidirectional layout is used, then some space is wasted! Example:

layout for A

offset	field
0	x

layout for C

offset	field
0	x
1	y
2	z

layout for B

offset	field
0	–
1	y

wasted

Bidirectional layout

For this particular hierarchy, it is however possible to use a **bidirectional layout** to avoid wasting space.

layout for `A`

offset	field
`0`	`x`

layout for `B`

offset	field
`-1`	`y`

layout for `C`

offset	field
`-1`	`y`
`0`	`x`
`1`	`z`

Bidirectional layouts

Bidirectional layouts are not ideal:

- there does not always exist a bidirectional layout that does not waste space,
- finding an optimal bidirectional layout – one minimizing the wasted space – is NP-complete,
- computing a good bidirectional layout requires the whole hierarchy to be known, and is not really compatible with Java-style run time linking.

Accessor methods

With multiple inheritance, the object layout problem can be solved by:

- laying out fields freely,
- defining accessor methods for them (getters/setters),
- always using accessors to get/set fields,
- overriding accessors in subclasses whenever a field changes position.

This reduces the object layout problem to the method dispatch problem, described later.

Other techniques

To summarize:

- bidirectional layout is fast but often waste space,
- accessor methods are slow, but do not waste space.

Two-dimensional, bidirectional layout:

- is slower than bidirectional layout, but faster than accessor methods,
- never wastes space.

Unfortunately, it also requires the full hierarchy to be known. We won't cover it here.

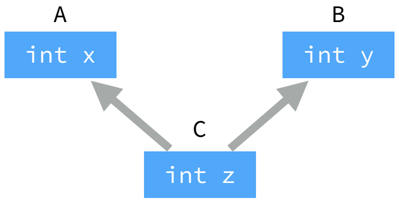
Object layout summary

Object layout summary:

- trivially solved by laying out fields sequentially, starting with those of the superclass, in Java-like languages that:
  1. offer only single inheritance,
  2. tie inheritance and subtyping,
- more difficult in a multiple-inheritance setting, where one must either trade space for speed, or speed for space.

Exercise

In C++ implementations, the position of a field is not the same in all subtypes of the type that introduced it. For our example, instances of `C` would typically be laid out as follows (gray fields contain information for method dispatch):



offset	field
0	▨
1	x
2	▨
3	y
4	z

With such a layout, what should happen when a method inherited from `B` is invoked on an instance of `C`?

OO problem #2: method dispatch

Method dispatch

The **method dispatch problem**:

When a method is invoked, how can the actual piece of code to execute be found efficiently?

Inclusion polymorphism makes this difficult since it prevents the problem to be solved statically – i.e. at compilation time. Efficient dynamic dispatching methods therefore have to be devised.

Method dispatch example

```
class A {
  int x;
  void m() { println("m in A"); }
  void n() { println("n in A"); }
}
class B extends A {
  int y;
  void m() { println("m in B"); }
  void o() { println("o in B"); }
}
void f(A a) { a.m(); }
```

which implementation of `m` should be invoked?

Case 1: single subtyping

Single subtyping

In OO languages like Java which:

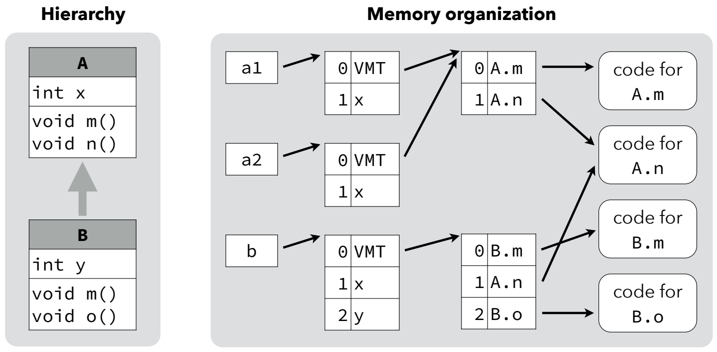
- have only single inheritance,
- tie inheritance and subtyping,

the method dispatch problem can be solved as follows:

Method pointers are stored sequentially, starting with those of the superclass, in a **virtual methods table** (**VMT**) shared by all instances of the class.

This ensures that the implementation for a given method is always at the same position in the VMT, and can be extracted quickly.

Virtual methods table



Program

```
A a1 = new A();
A a2 = new A();
B b = new B();
```

Dispatching with VMTs

Using a VMT, dispatching is accomplished by:

1. extracting the VMT of the selector,
2. extracting the code pointer for the invoked method from the VMT,
3. invoking the method implementation.

On a modern CPU, one step = one instruction.

VMTs pros and cons

VMT pros:

- very efficient dispatching,
- low memory usage,
- also work when new classes can be added at run time at the bottom of the hierarchy (as in Java).

VMT cons:

- not usable for dynamic languages, or in the presence of any kind of multiple subtyping (e.g. Java interfaces).

Case 2: multiple subtyping

Java interfaces

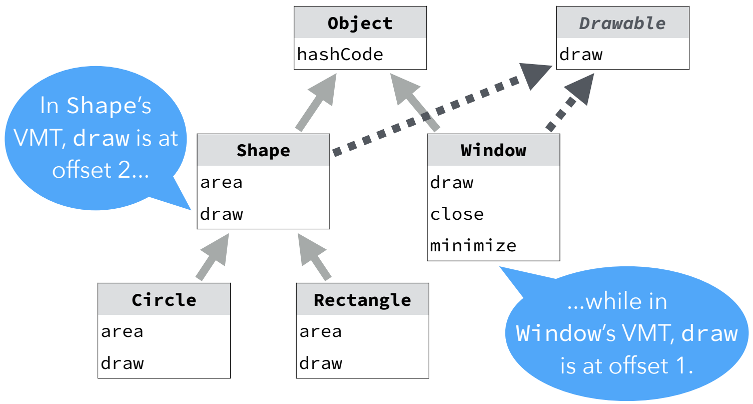
To understand why VMTs cannot be used with multiple subtyping, consider Java interfaces:

```
interface Drawable { void draw(); }
void drawAll(List<Drawable> ds) {
  for (Drawable d: ds)
    d.draw();
}
```

When the `draw` method is invoked:

- we know `d` has a `draw` method, but
- we don't know where that method is in the VMT, since the class of `d` can be anywhere in the hierarchy.

Java interfaces



Dispatching matrix

A trivial way to solve the problem is to use a global **dispatching matrix**, containing code pointers and indexed by classes and methods.

	hashCode	draw	close	minimize	area
Object	hashCode$_O$				
Shape	hashCode$_O$				
Circle	hashCode$_O$	draw$_C$			area$_C$
Rectangle	hashCode$_O$	draw$_R$			area$_R$
Window	hashCode$_O$	draw$_W$	close$_W$	minimize$_W$	

Dispatching matrix

Dispatching matrix pros:

- dispatching is very fast.

Dispatching matrix cons:

- too big to be usable as-is in practice.

Solution: compress the matrix, taking advantage of:

1. its sparsity (a class implements only a limited subset of all methods),
2. its redundancy (many methods are inherited).

Null elimination

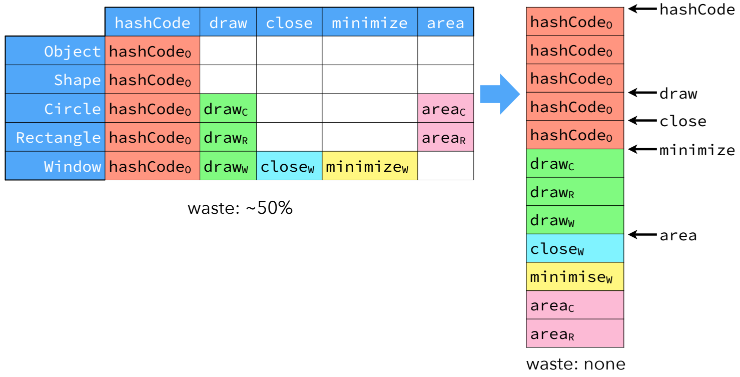
The dispatching matrix is very sparse (~50% full in our example).
Null elimination takes advantage of that sparsity, by "eliminating" the `nulls` that take a lot of space.
Column (or row) displacement is such a technique.

Column displacement

Column displacement:
- transform matrix to linear array, by shifting its columns,
- shift in a smart way to "fill" the holes in the process.

(Row displacement is another option, but column displacement works better in practice.)

Column displacement



	hashCode	draw	close	minimize	area
Object	hashCode$_O$				
Shape	hashCode$_O$				
Circle	hashCode$_O$	draw$_C$			area$_C$
Rectangle	hashCode$_O$	draw$_R$			area$_R$
Window	hashCode$_O$	draw$_W$	close$_W$	minimize$_W$	

waste: ~50%

hashCode$_O$
hashCode$_O$
hashCode$_O$
hashCode$_O$
hashCode$_O$
draw$_C$
draw$_R$
draw$_W$
close$_W$
minimise$_W$
area$_C$
area$_R$

waste: none

Dispatching with CD

Dispatching with column displacement consists in:

1. extract the code pointer by adding:
   - the offset of the method being invoked (known at compilation time) and
   - the offset of the class of the receiver (known only at run time),
2. invoking the method referenced by that pointer.

As fast as dispatching with an uncompressed matrix!

Duplicates elimination

The dispatching matrix is also very redundant. Null elimination does not exploit this characteristic.

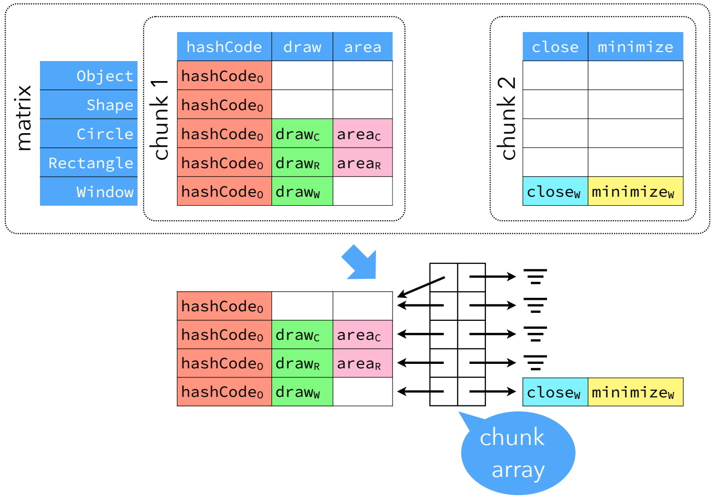
Duplicates elimination techniques try to share as much information as possible instead of duplicating it.

Compact dispatch table is such a technique.

Compact dispatch tables

Compact dispatch table:
- split the dispatch matrix into sub-matrices (**chunks**),
- share the duplicated chunk rows via a chunk array.

Compact dispatch tables



	hashCode	draw	area
Object	hashCode$_O$		
Shape	hashCode$_O$		
Circle	hashCode$_O$	draw$_C$	area$_C$
Rectangle	hashCode$_O$	draw$_R$	area$_R$
Window	hashCode$_O$	draw$_W$	

close	minimize
close$_W$	minimize$_W$

hashCode$_O$		
hashCode$_O$	draw$_C$	area$_C$
hashCode$_O$	draw$_R$	area$_R$
hashCode$_O$	draw$_W$	

close$_W$	minimize$_W$

Dispatching with CDTs

Dispatching with a compact dispatch table consists in:
1. extracting the code pointer by using:
   - the offset and chunk of the method being invoked (known at compilation time) and
   - the offset of the class of the receiver (known only at run time),
2. invoking the method referenced by that pointer.

Compared to column displacement:
- slightly slower due to additional indirection,
- better compression rates in practice.

Hybrid techniques

Hybrid dispatching techniques can be used, for example in Java:
- use VMTs when the type of the receiver is a class type,
- use other techniques when it is an interface type.

The JVM even has different instructions:
- `invokevirtual` for class dispatch (based on VMTs),
- `invokeinterface` for interface dispatch.

Method dispatch optimization

Inline caching

Even when dispatch is efficient, doing it on every method call is expensive.
Observation:

In practice, many call sites are **monomorphic** (i.e. target a *single* implementation).

Inline caching takes advantage of this by:

- recording, at every call site, the target of the latest dispatch, and
- assuming that the next one will be the same.

Implementing inline caching

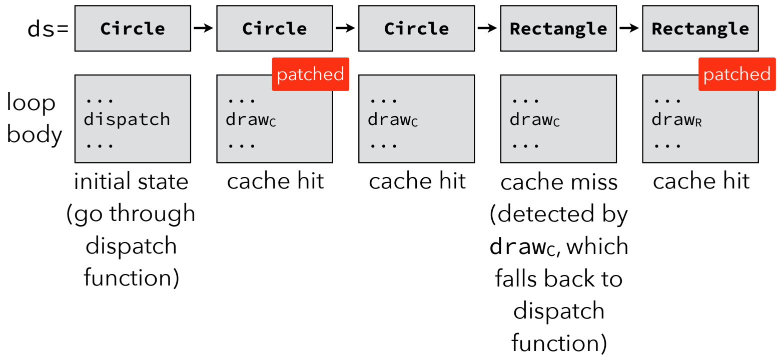
Inline caching works by patching code. At first, all method calls go through a standard dispatching function that:

1. computes the target of the call,
2. patches the call site to refer to that target.

To handle mispredictions, all methods start with a check that invokes the standard dispatching function in that case.

Inline caching example

```
for (Drawable d: ds) d.draw();
```



Inline caching pros & cons

Inline caching pros:

- greatly speeds up method calls at monomorphic call sites.

Inline caching cons:

- slows down method calls at polymorphic call sites (e.g. alternating circles and rectangles in our example).

Polymorphic inline caching addresses this issue.

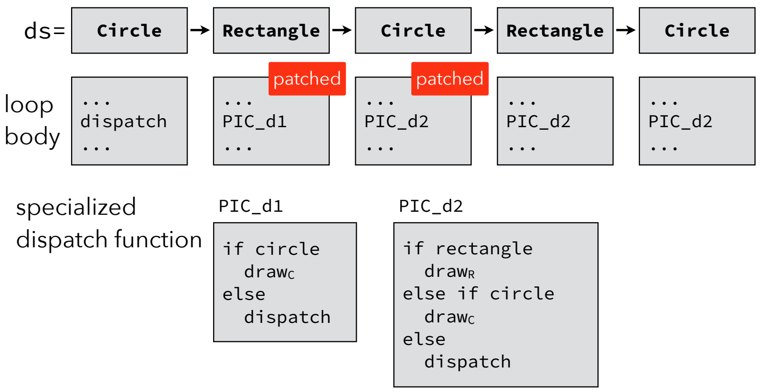
Polymorphic inline caching

Inline caching replaces the call to the dispatch function by a call to the latest method that was dispatched to.

Polymorphic inline caching (**PIC**) replaces it instead by a call to a specialized dispatch routine, generated on the fly. That routine handles only a subset of the possible receiver types – namely those that were encountered previously at that call site.

PIC example

```
for (Drawable d: ds) d.draw();
```



PIC receiver type test

The specialized dispatch function must check very quickly whether an object is of a given type:

- can be done by storing a class id in every object,
- this checks type *equality*, not sub-typing,
- therefore, an inherited method can appear several times in the dispatch function.

PIC optimizations

The methods called from the specialized dispatch function can be inlined into it. For example, `PIC_d2` could become:

```
if rectangle
  // inlined code of drawR
else if circle
  // inlined code of drawC
else
  dispatch
```

Also, the tests can be rearranged so that the ones corresponding to the most frequent receiver types appear first.

Method dispatch summary

Method dispatch summary:

- trivially solved by VMTs in Java-like languages that:
  1. offer only single inheritance,
  2. tie inheritance and subtyping,
- less trivially solved in other languages, usually using some compressed variant of the dispatching matrix.

In both cases, (polymorphic) inline caching can dramatically reduce the cost of dispatching.

Exercise

As we have seen, inline caching is useful to optimize method dispatch in an object-oriented (OO) language.
Could it also be useful in a functional (and not OO) language? Explain.

OO problem #3: membership test

Membership test

The **membership test problem:**

How to check efficiently at run time that an object has a given type?

This problem must be solved often, e.g. in Java:

- when the `instanceof` operator is used,
- when a type cast is performed,
- when a value is stored in an array,
- when an exception is thrown (to find the matching handler).

Membership test example

```
class A { }
class B extends A { }
boolean f(Object o) {
  return (o instanceof A);
}
```

is o an instance of A or any of its subtypes?

Case 1: single subtyping

Membership test

As usual, membership test is relatively easy to do in a single subtyping setting.

We will examine two techniques that work in that context:

1. relative numbering, and
2. Cohen’s encoding.

Relative numbering

Relative numbering numbers the types in the hierarchy during a preorder (or postorder) traversal.

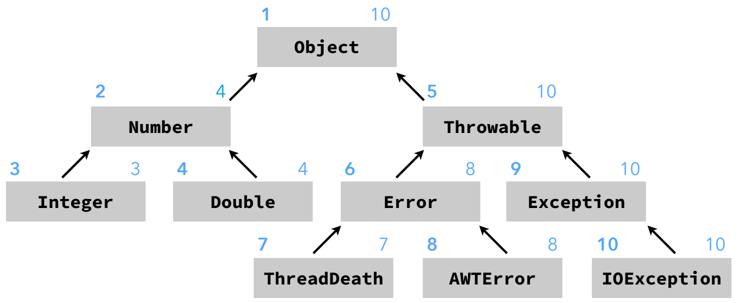
Property:

All descendants of a type are numbered consecutively.

Therefore:

Membership can be tested by checking whether the type of the object lies within a given interval.

Relative numbering example



x **instanceof** Throwable ⇔ 5 ≤ x.tid ≤ 10

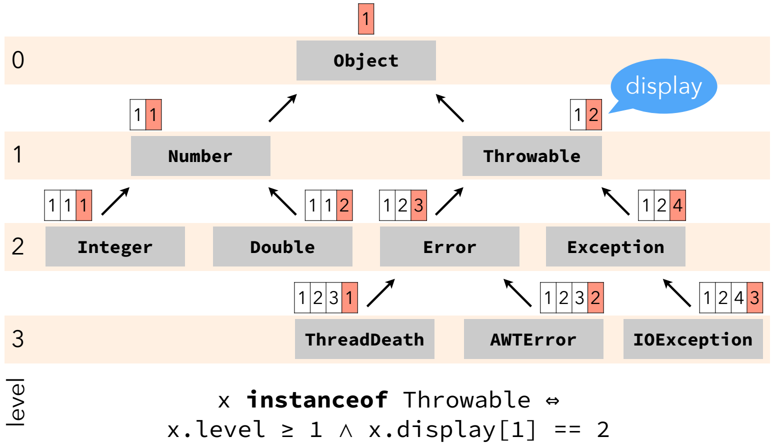
Cohen's encoding

Cohen's encoding:

1. partition types according to their **level** (distance from root) in the hierarchy,
2. number types so that no two types at a given level have the same number,
3. attach a **display** to all types, mapping all smaller levels to the number of the ancestor at that level.

Membership can be tested by checking the display at the appropriate level.

Cohen's encoding example



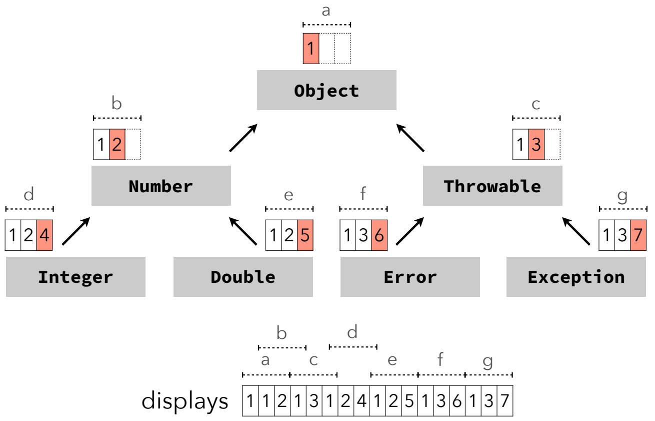
x **instanceof** Throwable ⇔
x.level ≥ 1 ∧ x.display[1] == 2

Global identifiers

The display bound-check can be removed by:

- using globally-unique identifiers,
- storing displays consecutively in memory, longest one at the end.

Cohen's encoding (global)



```
x instanceof Throwable ⇔ x.display[1] == 3
```

Comparison

Cohen's encoding:

- is more complicated, and
- uses more memory than relative numbering.

However, Cohen's encoding is **incremental**, i.e. new types can be added to the bottom of the hierarchy without needing a global recomputation.

This is important for systems where new types can be added at run time, e.g. Java.

Case 2: multiple subtyping

Membership test

In a multiple subtyping setting, neither relative numbering nor Cohen's encoding can be used directly.

Techniques that work with multiple subtyping can however be derived from them. We'll look at three of them:

1. range compression,
2. packed encoding, and
3. PQ encoding.

Range compression

Range compression generalizes relative numbering to a multiple subtyping setting.

It consists in numbering types during a preorder (or postorder) traversal of a spanning forest of the hierarchy.

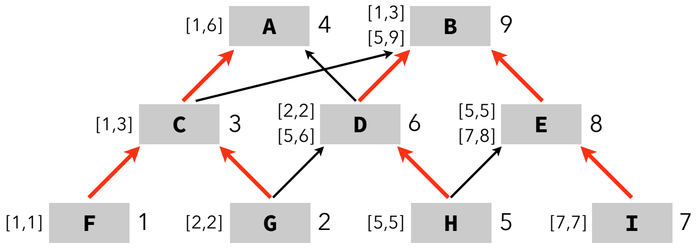
Property:

All descendants of a type should be numbered mostly consecutively.

Therefore:

Membership can be tested by checking whether the type of the object lies within a (small) set of intervals.

Range compression example



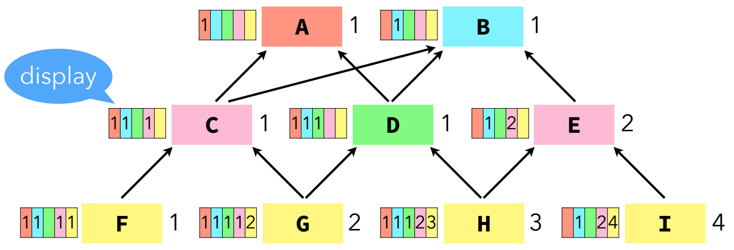
```
x instanceof B ⇔ x.tid ∈ [1,3] ∨ x.tid ∈ [5,9]
```

Packed encoding

Packed encoding generalizes Cohen's encoding to a multiple inheritance setting:

1. partition types into slices (as few as possible), such that all ancestors of a type are in different slices,
2. number types so that no two types in a given slice have the same number,
3. attach a display to all types, mapping all slices to the number of the ancestor in that slice.

Packed encoding example



```
x instanceof B ⇔ x.display[1] == 1
```

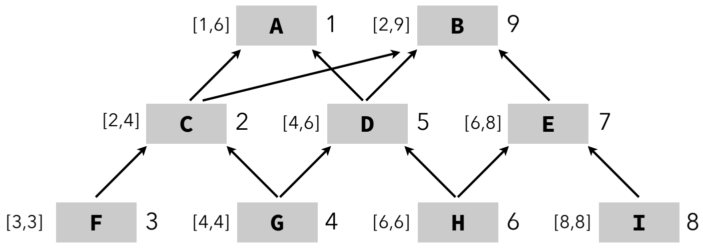
Cohen's/packed encoding

Cohen's encoding is a special case of packed encoding where slices are levels.
This is legal with single inheritance, as no two ancestors can be at the same level (i.e. in the same slice).

PQ encoding

PQ encoding combines ideas from packed encoding and relative numbering:
- partition types into slices (as few as possible),
- uniquely number types in each slices so that:
  for all types T in a slice S, all descendants of T – independently of their slice – are numbered consecutively in slice S.

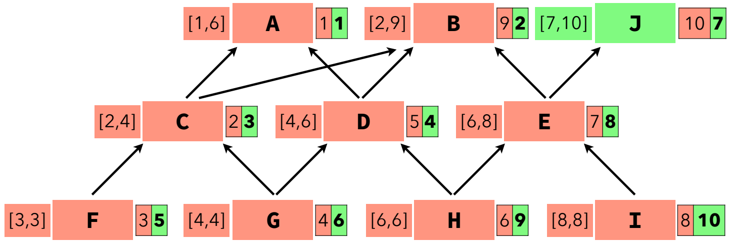
PQ encoding example 1



```
x instanceof B ⇔ x.tid ∈ [2,9]
```

Note: a single slice is sufficient for this hierarchy.

PQ encoding example 2



```
x instanceof B ⇔ x.tid[0] ∈ [2,9]
x instanceof J ⇔ x.tid[1] ∈ [7,10]
```

Hybrid techniques

Like for the dispatch problem, it is perfectly possible to combine several solutions to the membership test problem.

For example, a Java implementation could use:

- Cohen’s encoding to handle membership tests for classes, and
- PQ encoding for interfaces.

Membership test summary

In a single subtyping context, two simple solutions to the membership test exist:

1. relative numbering, and
2. Cohen’s encoding.

The first isn't incremental, the second is.

These techniques can be generalized to a multiple subtyping context to get:

1. range compression,
2. packed encoding,
3. PQ encoding.

Unfortunately, none of them are incremental.