

Copyright ©1998 Timothy Howard Merrett

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and full citation in a prominent place. Copyright for components of this work owned by others than T. H. Merrett must be honoured. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or fee. Request permission to republish from: T. H. Merrett, School of Computer Science, McGill University, fax 514 398 3883.

The author gratefully acknowledges support from the taxpayers of Québec and of Canada who have paid his salary and research grants while this work was developed at McGill University, and from his students (who built the implementations and investigated the data structures and algorithms) and their funding agencies.

T. H. Merrett

©98/11

Object-Oriented Relations

- “Object-orientation” is about *instantiation*.
- Instantiation is needed for programming language “functions” with *state*.
- Such states are values of stored variables, or *attributes*.
- Database relations are stored values: a state could be a tuple.
- It could also be a tuple of a nested relation; or a subrelation.
- Thus, nesting is usually a part of OODB, called *complex objects*.
- Aldat has nesting, in ways efficient for secondary storage.
- Aldat also can instantiate, in bulk, computations with state.
- Instances form *classes*.
- Classes may contain each other: *inheritance* can save code.

Relations as Classes

relation *Couch*(*Id*, *Length*, *Width*);

relation *Chair*(*Id*, *Base*);

relation *Furniture*(*Id*, *Manuf*);

Couch **isa** *Furniture*;

Chair **isa** *Furniture*;

Now the projection,

[*Manuf*] **in** *Couch*

is syntactic sugar for

[*Manuf*] **in** (*Couch natjoin Furniture*)

and similarly for any other use of *Manuf* in *Couch* or *Chair*

<i>Furniture</i>		<i>Couch</i>			
(<i>Id</i>	<i>Manuf</i>)	(<i>Id</i>	<i>Length</i>	<i>Width</i>)	<i>Manuf</i>
1	Mobel	1	15	5	Mobel
2	Furn	2	17	5	Mobel
3	Mobel	3	18	6	Mobel
21	Mobel				
22	Furn	<i>Chair</i>	(<i>Id</i>	<i>Base</i>)	<i>Manuf</i>
			21	4	Mobel
			22	5	Furn

Inheritance as join

- Inheritance could also be *implemented* as a join,
- but most such O-O uses are *low activity* operations,
- and best implemented with pointer dereferencing instead of the full join.
- So the special case of O-O gets a special implementation:
- another example of syntactic sugar hiding specialized algorithms and data structures,
- although *defined* in terms of the general operator.

Inheritance as join

Here is a variant.

```
relation Couch(Id, Length, Width);  
relation Furniture(Fid, Manuf);  
Couch [Id isa Fid] Furniture;
```

[*Manuf*] **in** *Couch*
is syntactic sugar for

[*Manuf*] **in** (*Couch* [*Id natjoin Fid*] *Furniture*)

- So **isa** translates directly into a precise, if possibly complex, specification for natural join.

- Inclusion dependence,

[*Id*] **in** *Couch* \subseteq [*Fid*] **in** *Furniture*

is not guaranteed: this would move **isa** from a purely syntactic specification to a *semantic* constraint.

Attaching Computations to Classes— Polymorphism

(a sketch)

```
comp PolyArea(Area) is  
{ comp Area(A) is  
  {A ← Length × Width;};           << public variables  
} also  
{ comp Area(A) is  
  {A ← Base**2;};                 << —not hidden  
};
```

```
FurnMethod ← Furniture natjoin PolyArea;  
Couch isa FurnMethod;  
Chair isa FurnMethod;  
let FootPrint be Area{}  
CouchPrint ← [Id, FootPrint] in Couch;  
ChairPrint ← [Id, FootPrint] in Chair;
```

FurnMethod
(*Id* *Manuf* *Area*)

1	Mobel	:
2	Furn	:
3	Mobel	:
21	Mobel	:
22	Furn	:

<i>CouchPrint</i>		<i>ChairPrint</i>	
(<i>Id</i>	<i>FootPrint</i>)	(<i>Id</i>	<i>FootPrint</i>)
1	75	21	16
2	85	22	25
3	108		