“What advanced techniques are there besides multiple implementation inheritance?”

The Adventure Game

```
Door

ShortDoor
  canPass(Person p)

LockedDoor
  canOpen(Person p)

ShortLockedDoor
  canOpen(Person p)
  canPass(Person p)
```
The Wrapper

FileStream
read()
write()

SocketStream
read()
write()

SynchRW
acquireLock()
releaseLock()

⚠ Cannot inherit from both separately
⚠ Creating new wrapping Classes duplicates code

The Wrapper

Stream
read()
write()

FileStream
read()
write()

SocketStream
read()
write()

SynchRW
acquireLock()
releaseLock()

⚠ Undoes specialization
⚠ Needs common ancestor

Classes and Methods

The building blocks for classes are
- a countable set of method names $\mathcal{N}$
- a countable set of method bodies $\mathbb{B}$

Classes map names to elements from the flat lattice $\mathcal{B}$ (called bindings), consisting of:
- method bodies $\in \mathbb{B}$ or classes $\in \mathcal{C}$
- attribute offsets $\in \mathbb{N}^+$
- $\bot$ (yet) undefined
- $\top$ in conflict
and the partial order $\bot \sqsubseteq m \sqsubseteq \top$ for each $m \in \mathbb{B}$

Definition (Abstract Class $\in \mathcal{C}$)

A partial function $c : \mathcal{N} \mapsto \mathbb{B}$ is called abstract class.

Definition (Interface and Class)

An abstract class $c$ is called

- interface if $\forall n \in \text{pre}(c) \cdot c(n) = \bot$
- (concrete) class if $\forall n \in \text{pre}(c) \cdot \bot \sqsubseteq (a) \sqsubseteq \top$.
Computing with Classes and Methods

**Definition (Family of classes \(\mathcal{C}\))**
We call the set of all maps from names to bindings the family of abstract classes \(\mathcal{C} := \mathcal{N} \rightarrow \mathcal{B}\).

Several possibilities for composing maps \(\mathcal{C} \sqcup \mathcal{C}\):

- the symmetric join \(\sqcup\), defined componentwise:
  \[
  (c_1 \sqcup c_2)(n) = b_1 \sqcup b_2 = \begin{cases} 
  b_2 & \text{if } b_1 = \bot \text{ or } n \notin \text{pre}(c_1) \\
  b_1 & \text{if } b_2 = \bot \text{ or } n \notin \text{pre}(c_2) \\
  b_2 & \text{if } b_1 = b_2 \\
  \top & \text{otherwise}
  \end{cases}
  \]

  where \(b_i = c_i(n)\)

- in contrast, the asymmetric join \(\sqcap\), defined componentwise:
  \[
  (c_1 \sqcap c_2)(n) = \begin{cases} 
  c_1(n) & \text{if } n \in \text{pre}(c_1) \\
  c_2(n) & \text{otherwise}
  \end{cases}
  \]

**Example: Smalltalk-Inheritance**

**Smalltalk** inheritance
- is the archetype for inheritance in mainstream languages like Java or C#
- inheriting smalltalk-style establishes a reference to the parent

**Definition (Smalltalk inheritance (\(\triangleright\)))**
Smalltalk inheritance is the binary operator \(\triangleright : \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}\), defined by
\[
  c_1 \triangleright c_2 = \{\text{super} \mapsto c_2\} \sqcup \{c_1\} \sqcap c_2
\]

**Example: Doors**

- \(\text{Door} = \{\text{canPass} \mapsto \bot, \text{canOpen} \mapsto \bot\}\)
- \(\text{LockedDoor} = \{\text{canOpen} \mapsto 0x4204711\} \triangleright \text{Door}\)
- \(\text{LockedDoor} = \{\text{super} \mapsto \text{Door}\} \sqcup \{\text{canOpen} \mapsto 0x4204711\} \sqcap \{\text{canPass} \mapsto \bot\}\)

**Excursion: Beta-Inheritance**

In **Beta**-style inheritance
- the design goal is to provide security from replacement of a method by a different method.
- methods in parents dominate methods in subclass
- the keyword \texttt{inner} explicitly delegates control to the subclass

**Definition (Beta inheritance (\(<\))**
Beta inheritance is the binary operator \(< : \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}\), defined by
\[
  c_1 < c_2 = \text{inner} \mapsto c_1 \sqcap (c_2 \sqcup c_1)
\]

**Example (equivalent syntax):**

```java
class Person {
    String name = "Axel Simon";
    public String toString() { return name+"\n"; }
}

class Graduate extends Person {
    public extension String toString() { return ", Ph.D."; }
}
```

**Adventure Game with Code Duplication**

- \(\text{Door}\)
- \(\text{ShortDoor}\)
- \(\text{LockedDoor}\)
- \(\text{ShortLockedDoor}\)

- \(\text{canPass(Person p)}\)
- \(\text{canOpen(Person p)}\)
Adventure Game with Mixins

```java
class Door {
    boolean canOpen(Person p) { return true; }
    boolean canPass(Person p) { return p.size() < 210; }
}
mixin Locked {
    boolean canOpen(Person p) {
        if (!p.hasItem(key)) return false; else return super.canOpen(p);
    }
}
mixin Short {
    boolean canPass(Person p) {
        if (p.height() > 1) return false; else return super.canPass(p);
    }
}
class ShortDoor = Short(Door);
class LockedDoor = Locked(Door);
mixin ShortLocked = Short o Locked;
class ShortLockedDoor = ShortLocked(Door);
class ShortLockedDoor2 = ShortLocked(Door);
```

Abstract model for Mixins

A Mixin is a *unary second order type expression*. In principle it is a curried version of the Smalltalk-style inheritance operator. In certain languages, programmers can create such mixin operators:

**Definition (Mixin)**

The mixin constructor \( \text{mixin} : C \rightarrow (C \rightarrow C) \) is a unary class function, creating a unary class operator, defined by:

\[
\text{mixin}(c) = \lambda x. c \triangleright x
\]

⚠️ Note: Mixins can also be composed \( \circ \):

**Example: Doors**

\[
\text{Locked} = \{ \text{canOpen} \rightarrow 0x1234 \}
\]

\[
\text{Short} = \{ \text{canPass} \rightarrow 0x4711 \}
\]

\[
\text{Composed} = \text{mixin}(\text{Short}) \circ (\text{mixin}(\text{Locked})) = \lambda x. \text{Short} \triangleright (\text{Locked} \triangleright x)
\]

= \lambda x. \{ \text{super} \rightarrow \text{Locked} \} \cup \{ \{ \text{canOpen} \rightarrow 0x1234, \text{canPass} \rightarrow 0x4711 \} \triangleright x \}
Wrapper with Mixins

Mixins on Implementation Level

```java
class Door {
    boolean canOpen(Person p) ... 
    boolean canPass(Person p)...
}
mixin Locked {
    boolean canOpen(Person p)...
} mixin Short {
    boolean canPass(Person p)...
}
class ShortDoor
    = Short(Door); 
class ShortLockedDoor
    = Short(Locked(Door));
...
ShortDoor d
    = new ShortLockedDoor();
```

⚠️ non-static super-References
≈ dynamic dispatching without precomputed virtual table

"Surely multiple inheritance is powerful enough to simulate mixins?"

Simulating Mixins in C++

```cpp
template <class Super>
class SyncRW : public Super {
public: virtual int read() {
    acquireLock();
    int result = Super::read();
    releaseLock();
    return result;
};
virtual void write(int n) {
    acquireLock();
    Super::write(n);
    releaseLock();
};
// ... acquireLock & releaseLock
};
```
Simulating Mixins in C++

template <class Super>
class LogOpenClose : public Super {
    public: virtual void open(){
        Super::open();
        log("opened");
    }
    virtual void close(){
        Super::close();
        log("closed");
    }
    protected: virtual void log(char*s) { ... };
};
class MyDocument : public SyncRW(LogOpenClose<Document>> {};

Ruby

class Person
    attr_accessor :size
    def initialize
        @size = 160
    end
    def hasKey
        true
    end
end
class Door
    def canOpen(p)
        true
    end
    def canPass(person)
        person.size < 210
    end
end

class ShortLockedDoor < Door
    include Short
    include Locked
end

module Short
    def canPass(p)
        p.size < 160 and super(p)
    end
end

module Locked
    def canOpen(p)
        p.hasKey() and super(p)
    end
end

true Mixins
- `super` natively supported
- Mixins as Template do not offer composite mixins
- C++ Type system not modular
- Mixins have to stay source code
- Hassle-free simplified version of multiple inheritance

C++ Mixins
- Mixins reduced to templated superclasses
- Can be seen as coding pattern

Common properties of Mixins
- Linearization is necessary
- Exact sequence of Mixins is relevant

Ruby

class Door
    def canOpen (p)
        true
    end
    def canPass(person)
        person.size < 210
    end
end

class Person
    attr_accessor :size
    def initialize
        @size = 160
    end
    def hasKey
        true
    end
    def canOpen(p)
        true
    end
    def canPass(person)
        person.size < 210
    end
    include Short
    include Locked
end

module ShortLocked
    include Short
    include Locked
end

p = Person.new
d = Door.new
d.extend ShortLocked
puts d.canPass(p)
Lessons Learned

- Formalisms to model inheritance
- Mixins provide soft multiple inheritance
- Multiple inheritance can not compensate the lack of super reference
- Full extent of mixins only when mixins are 1st class language citizens