Programming Languages

Multiple Inheritance

Dr. Michael Petter
Winter term 2014
“Wouldn’t it be nice to inherit from several parents?”

**Interface vs. Implementation inheritance**

The classic motivation for inheritance is implementation inheritance

- **Code reusage**
- Child specializes parents, replacing particular methods with custom ones
- Parent acts as library of common behaviours
- Implemented in languages like C++ or Lisp

Code sharing in interface inheritance inverts this relation

- **Behaviour contract**
- Child provides methods, with signatures predetermined by the parent
- Parent acts as generic code frame with room for customization
- Implemented in languages like Java or C#

---

**Interface Inheritance**

- **PowerShovel**
  - `actuate()`

- **FrontLoader**
  - `actuate()`

- **House**
  - `actuate()`

- **Bulldozer**

- **Undercarriage**
  - `moveTo(x,y)`

- **Tracks**
  - `moveTo(x,y)`

- **Wheels**
  - `moveTo(x,y)`

**Implementation inheritance**

- **Ship**
  - `toot()`
  - `moveTo(x,y)`

- **Airport**
  - `shelter(Plane)`

- **Aircraft Carrier**
  - `strikeAt(x,y)`
Excursion: LSP and Square-Rect-Problem

The Liskov Substitution Principle
Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

class Rectangle {
    void setWidth (int w){ this.w=w; }
    void setHeight (int h){ this.h=h; }
    void getWidth () { return w; }
    void getHeight () { return h; }
}
class Square extends Rectangle {
    void setWidth (int w){ this.w=w; h=w; }
    void setHeight (int h){ this.h=h; w=h; }
}

Rectangle r = new Square(2);
    r.setWidth(3);
    r.setHeight(4);
    assert r.getHeight() == 12;

class Square extends Rectangle {
    void setWidth (int w){ this.w=w; h=w; }
    void setHeight (int h){ this.h=h; w=h; }
}

Rectangle r = new Square(2);
    r.setWidth(3);
    r.setHeight(4);
    assert r.getHeight() == 12;

Behavioural assumptions

Excursion: Brief introduction to LLVM IR

Low Level Virtual Machine as reference semantics:

; (recursive) struct definitions
%struc.A = type { i32, %struc.B, i32(i32)* } %struc.B = type { i64, [10 x [20 x i32]], i8 }

; allocation of objects
%a = alloc %struc.A
; address computation for selection in structure (pointers):
%1 = getelementptr %struc.A* %a, i64 0, i64 2
; load from memory
%2 = load i32(i32)* %1
; indirect call
%retval = call i32 (i32)* %2(i32 42)

Retrieve the memory layout of a compilation unit with:
clang -ccl --x c++ -v -fdump-record-layouts -emit-llvm source.cpp

Retrieve the IR Code of a compilation unit with:
clang -G1 -S -emit-llvm source.cpp -o IR.lyoutu
Object layout

```plaintext
class A {
    int a; int f(int);
};
class B : public A {
    int b; int g(int);
};
class C : public B {
    int c; int h(int);
};
...
C c;
c.g(42);
%
c = alloca %class.C
%1 = bitcast %class.C* %c to %class.B*
%2 = call i32 @g(%class.B* %1, i32 42); g is statically known
```

Translation of a method body

```c
class A {
    int a; int f(int);
};
class B : public A {
    int b; int g(int);
};
class C : public B {
    int c; int h(int);
};
int B::g(int p) {
    return p+b;
};
```

Object layout

```plaintext
class A {
    int a; int f(int);
};
class B : public A {
    int b; int g(int);
};
class C : public B {
    int c; int h(int);
};
...
C c;
c.g(42);
%
c = alloca %class.C
%1 = bitcast %class.C* %c to %class.B*
%2 = call i32 @g(%class.B* %1, i32 42); g is statically known
```

Translation of a method body

```c
class A {
    int a; int f(int);
};
class B : public A {
    int b; int g(int);
};
class C : public B {
    int c; int h(int);
};
int B::g(int p) {
    return p+b;
};
```
Object layout – virtual methods

class A {
    int a; virtual int f(int);
    virtual int g(int);
    virtual int h(int);
};
class B : public A {
    int b; int g(int);
};
class C : public B {
    int c; int h(int);
};
...
C c;
c.g(42);

C

\%c.vptr = bitcast \%c.class.C* \%c to i32 (%class.B*, i32)***; vtbl
\%1 = load (%class.B*, i32)*** \%c.vptr ; dereference vptr
\%2 = getelementptr \%1, i64 1 ; select g()-entry
\%3 = load (%class.B*, i32)** \%2 ; dereference g()-entry
\%4 = call i32 %3(%class.B*, \%c, i32 42)

“So how do we include several parent objects?”

Multiple inheritance

Multiple inheritance class diagram

A
int f(int)
int a

B
int g(int)
int b

C
int h(int)
int c

Multiple Base Classes

class A {
    int a; int f(int);
};
class B {
    int b; int g(int);
};
class C : public A , public B {
    int c; int h(int);
};
...
C c;
c.g(42);

\%c = alloca \%class.C
\%1 = bitcast \%class.C* \%c to i8*
\%2 = getelementptr i8* \%1, i64 4 ; select B-offset in C
\%3 = call i32 0.g(%class.B* \%2, i32 42) ; g is statically known
Multiple Base Classes

class A {
    int a; int f(int);
};
class B {
    int b; int g(int);
};
class C : public A, public B {
    int c; int h(int);
};
...
C c;
c.g(42);

$class.C = \text{type} (\underbrace{\text{C}.A, \text{C}.B, 132})$
$class.A = \text{type} (132)$
$class.B = \text{type} (132)$

$\Diamond C = \text{alloca} \%\text{class.C}$
$\%1 = \text{bitcast} \%\text{class.C} \rightarrow \text{i8}*$
$\%2 = \text{getelementptr} \text{i8}* \%1, \text{i64} 4$ ; select B-offset in C
$\%3 = \text{call} \text{i32} \text{O}_g(\%\text{class.B} \rightarrow \%2, 132 42)$ ; g is statically known

\text{getelementptr} \text{implements} \text{$\Delta B$ as} 4 \cdot 8$

Static Type Casts

class A {
    int a; int f(int);
};
class B {
    int b; int g(int);
};
class C : public A, public B {
    int c; int h(int);
};
...
B* b = new C();

calls $\%1 = \text{call} \text{i8}* \text{O}_\text{new}(\text{i64} 12)$
calls $\text{void} \text{O}_\text{memset}.\text{pO18.164}(\text{i8}*, \%1, \text{i8} 0, \text{i64} 12, \text{i32} 4, \text{i1} \text{false})$
$\%2 = \text{getelementptr} \text{i8}* \%1, \text{i64} 4$
$\%b = \text{bitcast} \text{i8}*$

Ambiguities

class A {
    void f(int);
};
class B {
    void f(int);
};
class C : public A, public B {
    C* pc;
    pc->f(42)

$\Diamond$ Which method is called?

\text{Solution I: Explicit qualification}

pc->A::f(42);
pc->B::f(42);

\text{Solution II: Automagical resolution}

\text{Idea: The Compiler introduces a linear order on the nodes of the inheritance graph}

Linearization

\text{Principle 1: Inheritance Relation}

\text{Defined by parent-child. Example:}

$C(A, B) \implies C \rightarrow A \rightarrow C \rightarrow B$

\text{In General:}

\begin{itemize}
  \item Inheritance is a uniform mechanism, and its searches (\rightarrow total order) apply identically for all object fields or methods
  \item In the literature, we also find the set of constraints to create a linearization as Method Resolution Order
  \item Linearization is a best-effort approach at best
\end{itemize}

\text{Principle 2: Multiplicity Relation}

\text{Defined by the succession of multiple parents. Example:}

$C(A, B) \implies A \rightarrow B$
MRO via Refined Postorder DFS

**Reverse Postorder Rightmost DFS**

\[ \mathcal{L}(A) = A B F D C E G H W \]

- Linear extension of inheritance relation
- Topological sorting

**RPRDFS**

\[ \mathcal{L}(A) = A B C D G E F \]

⚠️ But principle 2 *multiplicity* is violated!

---

MRO via Refined Postorder DFS

**Reverse Postorder Rightmost DFS**

\[ \mathcal{L}(A) = A B F D C E G H W \]

- Linear extension of inheritance relation
- Topological sorting

**RPRDFS**

\[ \mathcal{L}(A) = A B C D G E F \]

⚠️ But principle 2 *multiplicity* is violated!
MRO via Refined Postorder DFS

**Reverse Postorder Rightmost DFS**

L[A] = A B F D C E G H W

- Linear extension of inheritance relation

- Topological sorting

**RPRDFS**

L[A] = A B C D G E F

⚠️ But principle 2 *multiplicity* is violated!

**Refined RPRDFS**

\[ A(B, C) \ B(F, D) \ C(E, H) \ D(G) \ E(G) \ F(W) \ G(W) \ H(W) \]

MRO via Refined Postorder DFS

**Reverse Postorder Rightmost DFS**

L[A] = A B F D C E G H W

- Linear extension of inheritance relation

- Topological sorting

**RPRDFS**

L[A] = A B C G E F

⚠️ But principle 2 *multiplicity* is violated!

- CLOS: uses Refined RPDFS [3]

**Refined RPRDFS**

\[ A(B, C) \ B(F, G) \ C(D, E) \ D(G) \ E(F) \]

⚠️ *Monotonicity* is not guaranteed!

**Extension Principle: Monotonicity**

If \( C_1 \rightarrow C_2 \) in \( C \)'s linearization, then \( C_1 \rightarrow C_2 \) for every linearization of \( C \)'s children.
MRO via Refined Postorder DFS

Refined RPRDFS

⚠️ Monotonicity is not guaranteed!

Extension Principle: Monotonicity
If \( C_1 \to C_2 \) in \( C \)'s linearization, then \( C_1 \to C_2 \) for every linearization of \( C \)'s children.

\[
L[A] = A B C D E F G \implies F \to G
\]

\[
L[C] = D G E F \implies G \to F
\]

MRO via C3 Linearization

A linearization \( L \) is an attribute \( L[C] \) of a class \( C \). Classes \( B_1 \ldots B_n \) are superclasses to child class \( C \), defined in the local precedence order \( C(B_1 \ldots B_n) \). Then

\[
L[C(B_1 \ldots B_n)] = C \cdot \bigsqcup \{ L[B_1] \ldots L[B_n], B_1 \ldots B_n \}
\]

\[
L[\text{Object}] = \text{Object}
\]

with

\[
\bigsqcup (L_i) = \begin{cases}
  c \cdot (\text{tail}(L_k) \cup \bigsqcup_{j \neq k} (L_j \setminus c)) & \text{if } \exists_{\min k} \ c = \text{head}(L_k) \notin \text{tail}(L_j) \\
  \text{fail} & \text{else}
\end{cases}
\]
C3 detects and reports a violation of {	extit{monotonicity}} with the addition of A(B,C) to the class set.

C3 linearization [1]: is used in OpenDylan, Python, and Perl 6
Linearization vs. explicit qualification

Linearization
- No switch/duplexer code necessary
- No explicit naming of qualifiers
- Unique super reference
- Reduces number of multi-dispatching conflicts

Qualification
- More flexible, fine-grained
- Linearization choices may be awkward or unexpected

Languages with automatic linearization exist
- CLOS Common Lisp Object System
- Dylan, Python and Perl 6 with C3
- Prerequisite for → Mixins

“And what about dynamic dispatching in Multiple Inheritance?”