Script **generated by TTT**

Title: Petter: Programmiersprachen (26.11.2014)

Date: Wed Nov 26 14:19:11 CET 2014

Duration: 103:45 min

Pages: 49

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Programming Languages

Multiple Inheritance

Dr. Michael Petter
Winter term 2014

Outline

Outline

“Wouldn’t it be nice to inherit from several parents?”

Interface vs. Implementation inheritance

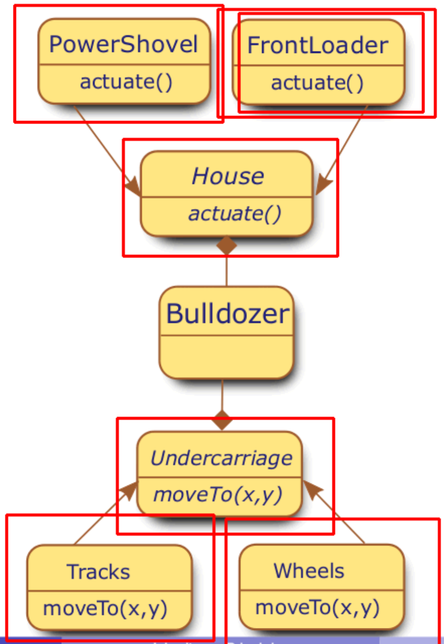
The classic motivation for inheritance is implementation inheritance

- *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

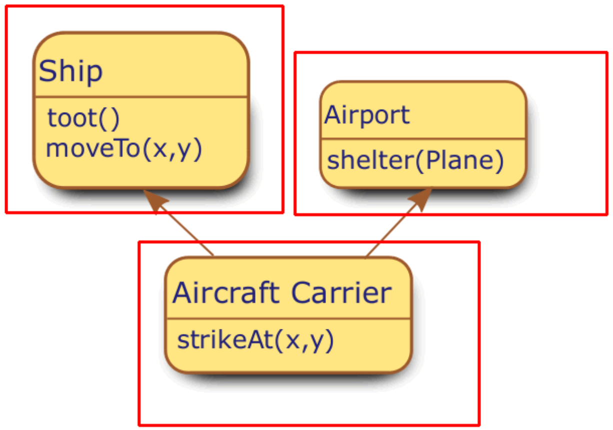
Code sharing in interface inheritance inverts this relation

- *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#

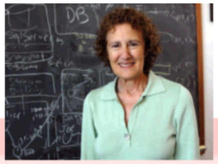
Interface Inheritance



Implementation inheritance



Excursion: LSP and Square-Rect-Problem



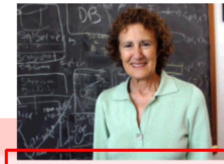
The Liskov Substitution Principle

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()*
        r.getWidth()==12;
```


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()*
        r.getWidth()==12;
```


⚠ Behavioural assumptions

“So how do we lay out objects in heap anyway?”

Excursion: Brief introduction to LLVM IR

Low Level Virtual Machine as reference semantics:

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

;allocation of objects
%a = alloca %struct.A
;adress computation for selection in structure (pointers):
%1 = getelementptr %struct.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 -cc1 -x c++ -v -fdump-record-layouts -emit-llvm source.cpp
```

Retrieve the IR Code of a compilation unit with:

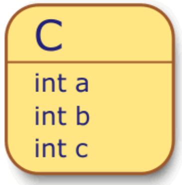
```
clang -O1 -S -emit-llvm source.cpp -o IR.llvm
```

Object layout

```
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);
```

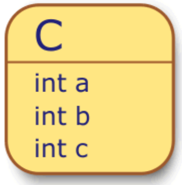



```
%class.C = type { %class.B, i32 }
%class.B = type { %class.A, i32 }
%class.A = type { i32 }
```

```
%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

```
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;
};
```




```
%class.C = type { %class.B, i32 }
%class.B = type { %class.A, i32 }
%class.A = type { i32 }
```

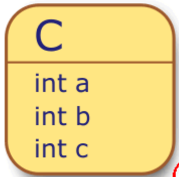
```
define i32 @_g(%class.B* %this, i32 %p) {
  %1 = getelementptr %class.B* %this, i64 0, i32 1
  %2 = load i32* %1
  %3 = add i32 %2, %p
  ret i32 %3
}
```


Object layout

```
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);
```



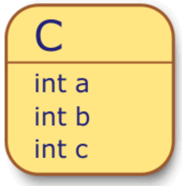

```
%class.C = type { %class.B, i32 }
%class.B = type { %class.A, i32 }
%class.A = type { i32 }
```

```
%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
```

Multiple Inheritance | Standard Object Heap Layout | Object layout & inheritance 10 / 44

Translation of a method body

```
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;
};
```



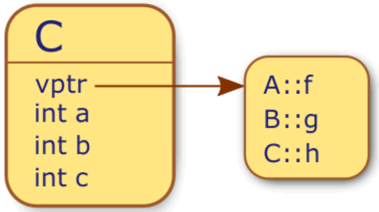

```
%class.C = type { %class.B, i32 }
%class.B = type { %class.A, i32 }
%class.A = type { i32 }
```

```
define i32 @_g(%class.B* %this, i32 %p) {
  %1 = getelementptr %class.B* %this, i64 0, i32 1
  %2 = load i32* %1
  %3 = add i32 %2, %p
  ret i32 %3
}
```

Multiple Inheritance | Standard Object Heap Layout | Object layout & inheritance 11 / 44

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);
```

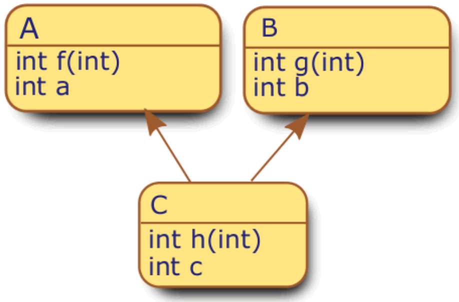



```
%class.C = type { %class.B, i32, [4 x i8] }
%class.B = type { [12 x i8], i32 }
%class.A = type { i32 (...)**, i32 }
```

```
%c.vptr = bitcast %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)
```

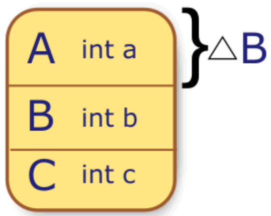

"So how do we include several parent objects?"

Multiple inheritance class diagram



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);
```

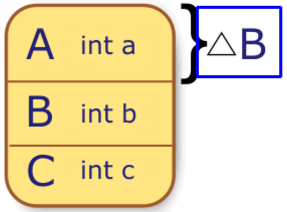



```
%class.C = type { %class.A, %class.B, i32 }
%class.A = type { i32 }
%class.B = type { i32 }
```

```
%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 @_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);
```



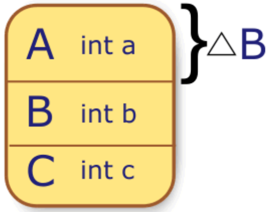

```
%class.C = type { %class.A, %class.B, i32 }
%class.A = type { i32 }
%class.B = type { i32 }
```

```
%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 @_g(%class.B* %2, i32 42)  ; g is statically known
```

⚠ `getelementptr` implements ΔB as $4 \cdot i8$!

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();
```




```
%class.C = type { %class.A, %class.B, i32 }
%class.A = type { i32 }
%class.B = type { i32 }
```

```
%1 = call i8* @_new(i64 12)
call void @_memset.p0i8.i64(i8* %1, i8 0, i64 12, i32 4, i1 false)
%2 = getelementptr i8* %1, i64 4
%b = bitcast i8* %2 to %class.B*
```


Ambiguities

```
class A { void f(int); };
class B { void f(int); };
class C : public A, public B {};

C* pc;
pc->f(42);
```

⚠ Which method is called?

Solution I: Explicit qualification

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

Solution II: Automagical resolution
Idea: The Compiler introduces a linear order on the nodes of the inheritance graph

Linearization

Principle 1: Inheritance Relation
Defined by parent-child. Example:
$C(A, B) \implies C \to A \wedge C \to B$

Principle 2: Multiplicity Relation
Defined by the succession of multiple parents. Example:
$C(A, B) \implies A \to B$

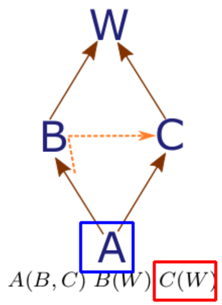
In General:

1. Inheritance is a uniform mechanism, and its searches ($\to$ total order) apply identically for all object fields or methods
2. In the literature, we also find the set of constraints to create a linearization as Method Resolution Order
3. Linearization is a best-effort approach at best

MRO via DFS

Leftmost Preorder Depth-First Search

A B W C



MRO via DFS

Leftmost Preorder Depth-First Search

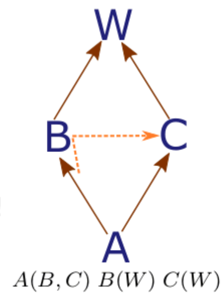
L[A] = A B W C

⚠ Principle 1 *inheritance* is violated

Python: classical python objects ($\leq$ 2.1) use LPDFS!

LPDFS with Duplicate Cancellation

A B ~~W~~ C W



MRO via DFS

Leftmost Preorder Depth-First Search

L[A] = A B W C

⚠ Principle 1 *inheritance* is violated

Python: classical python objects ($\leq$ 2.1) use LPDFS!

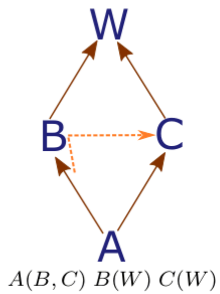
LPDFS with Duplicate Cancellation

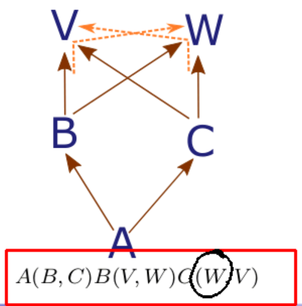
L[A] = A B C W

✓ Principle 1 *inheritance* is fixed

LPDFS with Duplicate Cancellation

A B ~~W V~~ C W V





MRO via DFS

Leftmost Preorder Depth-First Search

L[A] = A B W C

⚠ Principle 1 *inheritance* is violated

Python: classical python objects ($\leq$ 2.1) use LPDFS!

LPDFS with Duplicate Cancellation

L[A] = A B C W

✓ Principle 1 *inheritance* is fixed

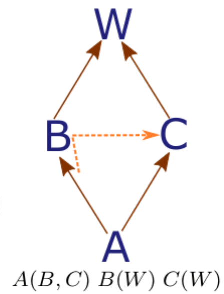
Python: new python objects (2.2) use LPDFS(DC)!

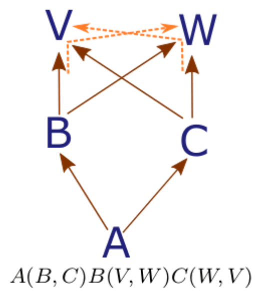
LPDFS with Duplicate Cancellation

L[A] = A B C W V

⚠ Principle 2 *multiplicity* not fullfillable

⚠ However $B \rightarrow C \implies W \rightarrow V$??

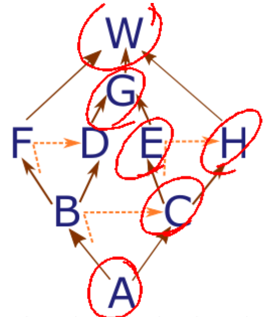




MRO via Refined Postorder DFS

Reverse Postorder Rightmost DFS

E G H W



$A(B,C)$ $B(F,D)$ $C(E,H)$
$D(G)$ $E(G)$ $F(W)$ $G(W)$ $H(W)$

MRO via Refined Postorder DFS

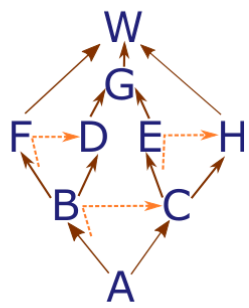
Reverse Postorder Rightmost DFS

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

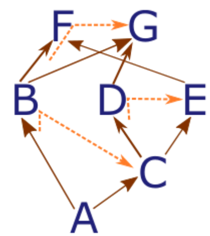
✓ Linear extension of inheritance relation

⇝ Topological sorting

RPRDFS



$A(B,C)$ $B(F,D)$ $C(E,H)$
$D(G)$ $E(G)$ $F(W)$ $G(W)$ $H(W)$



$A(B,C)$ $B(F,G)$ $C(D,E)$
$D(G)$ $E(F)$

MRO via Refined Postorder DFS

Reverse Postorder Rightmost DFS

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

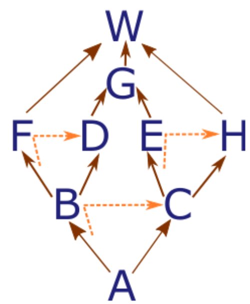
✓ Linear extension of inheritance relation

⇝ Topological sorting

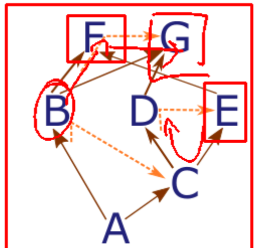
RPRDFS

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

⚠ But principle 2 *multiplicity* is violated!



$A(B,C)$ $B(F,D)$ $C(E,H)$
$D(G)$ $E(G)$ $F(W)$ $G(W)$ $H(W)$



$A(B,C)$ $B(F,G)$ $C(D,E)$
$D(G)$ $E(F)$

MRO via Refined Postorder DFS

Reverse Postorder Rightmost DFS

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

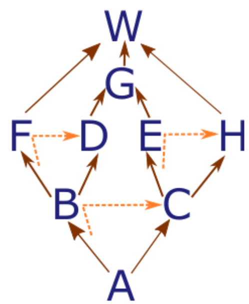
✓ Linear extension of inheritance relation

⇝ Topological sorting

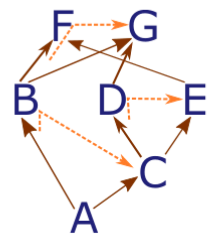
RPRDFS

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

⚠ But principle 2 *multiplicity* is violated!



$A(B,C)$ $B(F,D)$ $C(E,H)$
$D(G)$ $E(G)$ $F(W)$ $G(W)$ $H(W)$



$A(B,C)$ $B(F,G)$ $C(D,E)$
$D(G)$ $E(F)$

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

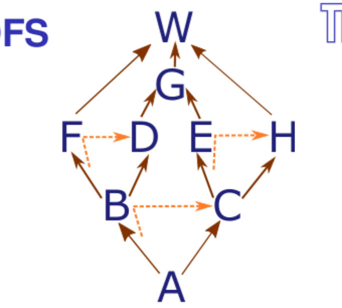
⇝ Topological sorting

RPRDFS

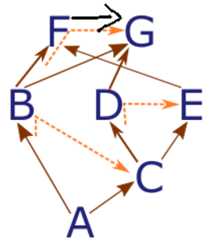
L[A] = A B C D G E F

⚠ But principle 2 *multiplicity* is violated!

Refined RPRDFS



$A(B,C)$ $B(F,D)$ $C(E,H)$
$D(G)$ $E(G)$ $F(W)$ $G(W)$ $H(W)$



$A(B,C)$ $B(F,G)$ $C(D,E)$
$D(G)$ $E(F)$

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

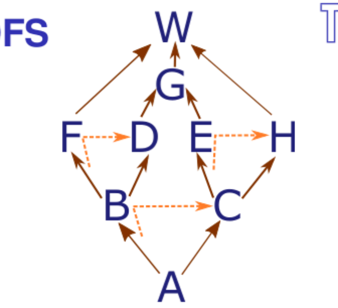
⚠ But principle 2 *multiplicity* is violated!

CLOS: uses Refined RPDFS [3]

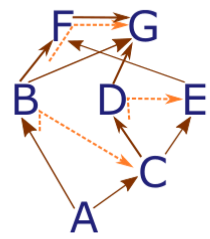
Refined RPRDFS

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

✓ Refine graph with conflict edge & rerun RPRDFS!



$A(B,C)$ $B(F,D)$ $C(E,H)$
$D(G)$ $E(G)$ $F(W)$ $G(W)$ $H(W)$

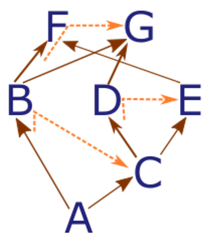


$A(B,C)$ $B(F,G)$ $C(D,E)$
$D(G)$ $E(F)$

MRO via Refined Postorder DFS

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.



$A(B,C)$ $B(F,G)$ $C(D,E)$
$D(G)$ $E(F)$

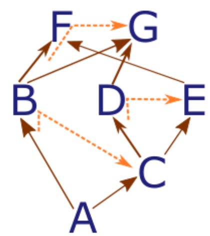
MRO via Refined Postorder DFS

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.



$A(B,C)$ $B(F,G)$ $C(D,E)$
$D(G)$ $E(F)$

MRO via Refined Postorder DFS

Refined RPRDFS

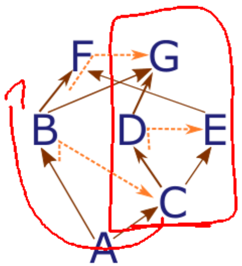
⚠ *Monotonicity* is not guaranteed!

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.

$$L[A] = A\ B\ C\ D\ E\ F\ G \quad \Longrightarrow \quad \boxed{F \rightarrow G}$$

$$\boxed{L[C] = \boxed{D\ G\ E\ F}} \quad \Longrightarrow \quad G \rightarrow F$$



$A(B,C)\ B(F,G)\ C(D,E)$
$D(G)\ E(F)$

MRO via C3 Linearization

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

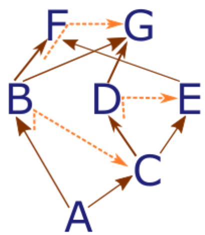
$$L[C(B_1 \dots B_n)] = C \cdot \bigsqcup(L[B_1] \dots L[B_n], B_1 \dots B_n)$$
$$L[Object] = Object$$

with

$$\bigsqcup(L_i) = \begin{cases} c \cdot (tail(L_k) \sqcup \bigsqcup_{j \neq k}(L_j \setminus c)) & \text{if } \exists_{\text{min } k}\ c = head(L_k) \notin tail(L_j) \\ ⚠ \text{ fail} & \text{else} \end{cases}$$

MRO via C3 Linearization

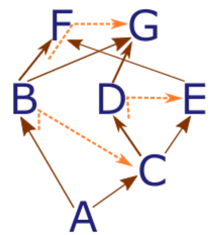
$L[G]$ G
$L[F]$ F
$L[E(F)]$
$L[D(G)]$
$L[B(F,G)]$
$L[C(D,E)]$
$L[A(B,C)]$



$A(B,C)\ B(F,G)\ C(D,E)$
$D(G)\ E(F)$

MRO via C3 Linearization

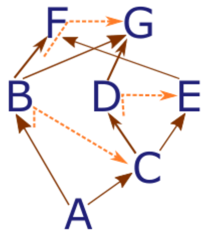
$L[G]$ G
$L[F]$ F
$L[E(F)]$ $E\ F$
$L[D(G)]$ $D\ G$
$L[B(F,G)]$
$L[C(D,E)]$
$L[A(B,C)]$



$A(B,C)\ B(F,G)\ C(D,E)$
$D(G)\ E(F)$

MRO via C3 Linearization

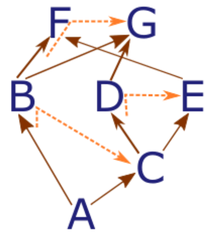
$L[G]$	G
$L[F]$	F
$L[E(F)]$	$E\ F$
$L[D(G)]$	$D\ G$
$L[B(F,G)]$	$B \cdot (L[F] \sqcup L[G] \sqcup \{F,G\})$
$L[C(D,E)]$	
$L[A(B,C)]$	



$A(B,C)\ B(F,G)\ C(D,E)\ D(G)\ E(F)$

MRO via C3 Linearization

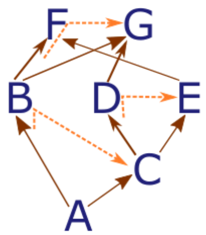
$L[G]$	G
$L[F]$	F
$L[E(F)]$	$E\ F$
$L[D(G)]$	$D\ G$
$L[B(F,G)]$	$B \cdot (\{F\} \sqcup \{G\} \sqcup \{F,G\})$
$L[C(D,E)]$	
$L[A(B,C)]$	



$A(B,C)\ B(F,G)\ C(D,E)\ D(G)\ E(F)$

MRO via C3 Linearization

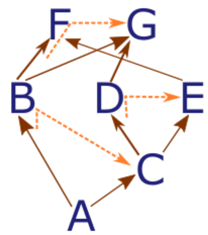
$L[G]$	G
$L[F]$	F
$L[E(F)]$	$E\ F$
$L[D(G)]$	$D\ G$
$L[B(F,G)]$	$B\ F\ G$
$L[C(D,E)]$	$C \cdot (\{D,G\} \sqcup \{E,F\} \sqcup \{D,E\})$
$L[A(B,C)]$	



$A(B,C)\ B(F,G)\ C(D,E)\ D(G)\ E(F)$

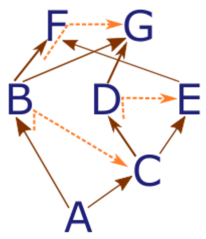
MRO via C3 Linearization

$L[G]$	G
$L[F]$	F
$L[E(F)]$	$E\ F$
$L[D(G)]$	$D\ G$
$L[B(F,G)]$	$B\ F\ G$
$L[C(D,E)]$	$C \cdot D \cdot (\{G\} \sqcup \{E,F\} \sqcup \{E\})$
$L[A(B,C)]$	



$A(B,C)\ B(F,G)\ C(D,E)\ D(G)\ E(F)$

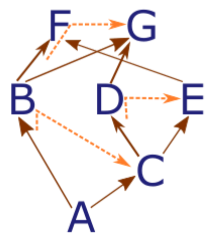
MRO via C3 Linearization



$A(B,C)\ B(F,G)\ C(D,E)$
$D(G)\ E(F)$

$L[G]$	G
$L[F]$	F
$L[E(F)]$	$E\ F$
$L[D(G)]$	$D\ G$
$L[B(F,G)]$	$B\ F\ G$
$L[C(D,E)]$	$C\ D\ G\ E\ F$
$L[A(B,C)]$	$A \cdot (\{B,F,G\} \sqcup \{C,D,G,E,F\} \sqcup \{B,C\})$

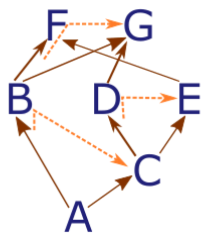
MRO via C3 Linearization



$A(B,C)\ B(F,G)\ C(D,E)$
$D(G)\ E(F)$

$L[G]$	G
$L[F]$	F
$L[E(F)]$	$E\ F$
$L[D(G)]$	$D\ G$
$L[B(F,G)]$	$B\ F\ G$
$L[C(D,E)]$	$C\ D\ G\ E\ F$
$L[A(B,C)]$	$A \cdot B \cdot C \cdot D \cdot (\{F,G\} \sqcup \{G,E,F\})$

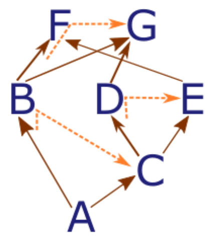
MRO via C3 Linearization



$A(B,C)\ B(F,G)\ C(D,E)$
$D(G)\ E(F)$

$L[G]$	G
$L[F]$	F
$L[E(F)]$	$E\ F$
$L[D(G)]$	$D\ G$
$L[B(F,G)]$	$B\ F\ G$
$L[C(D,E)]$	$C\ D\ G\ E\ F$
$L[A(B,C)]$	⚠ fail

MRO via C3 Linearization



$A(B,C)\ B(F,G)\ C(D,E)$
$D(G)\ E(F)$

$L[G]$	G
$L[F]$	F
$L[E(F)]$	$E\ F$
$L[D(G)]$	$D\ G$
$L[B(F,G)]$	$B\ F\ G$
$L[C(D,E)]$	$C\ D\ G\ E\ F$
$L[A(B,C)]$	⚠ fail

C3 detects and reports a violation of *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?"