Script generated by TTT

Title: Petter: Programmiersprachen (11.12.2019)

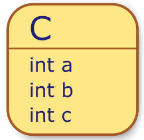
Date: Wed Dec 11 12:33:08 CET 2019

Duration: 85:22 min

Pages: 21

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 }
```

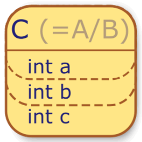
```
define i32 @_g(%class.B* %this, i32 %p) {
  %1 = getelementptr %class.B* %this, i64 0, i32 1
  %2 = load i32* %1
  %3 = add i32 %2, %p
  ret i32 %3
}
```

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);
```

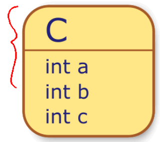



```
%class.C = type { %class.B, i32 }
%class.B = type { %class.A, i32 }
%class.A = type { i32 }
```

```
%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
```

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;
};
```

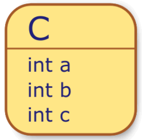



```
%class.C = type { %class.B, i32 }
%class.B = type { %class.A, i32 }
%class.A = type { i32 }
```

```
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
}
```

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
}
```

Single-Dispatching implementation choices

Single-Dispatching needs runtime action:

1. Manual search run through the super-chain (Java Interpreter ⇝ last talk)

```
call i32 @__dispatch(%class.C* %c,i32 42,i32* "f(int,void)")
```

Single-Dispatching implementation choices

Single-Dispatching needs runtime action:

1. Manual search run through the super-chain (Java Interpreter ⇝ last talk)

```
call i32 @__dispatch(%class.C* %c,i32 42,i32* "f(int,void)")
```

2. Caching the dispatch result (⇝ Hotspot/JIT)

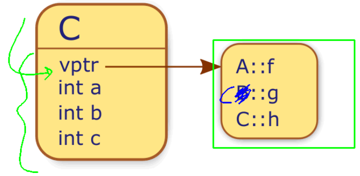
```
; caching the recent result value of the __dispatch function
; call i32 @__dispatch(%class.C* %c,i32 42)
assert (%c type %class.D) ; verify objects class presumption
call i32 @_f_from_D(%class.C* %c, i32 42) ; directly call f
```

3. Precomputing the dispatching result in tables
   1. Full 2-dim matrix
   2. 1-dim Row Displacement Dispatch Tables
   3. Virtual Tables
      (⇝ LLVM/GNU C++,this talk)

	f()	g()	h()	i()	j()	k()	l()	m()	n()
A	1								
B	1	2							
C	3		4						
D	3	2	4	5					
E						6		7	
F					8	9		7	

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);
```

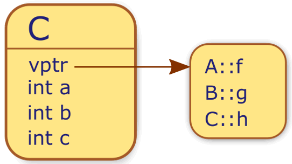



```
%class.C = type { %class.B, i32, [4 x i8] }
%class.B = type { [12 x i8], i32 }
%class.A = type { i32 (...)**, i32 }
```

```
%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)
```

Object layout – virtual methods

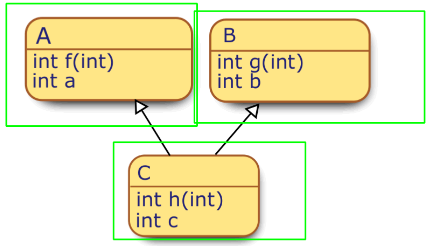
```
class A {
  int a; virtual int f(int);
         virtual int g(int);
         virtual int h(int);
};
class B : public A {
  int b; int g(int);
};
class C : public B {
  int c; int h(int);
};  ...
C c;
c.g(42);
```




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

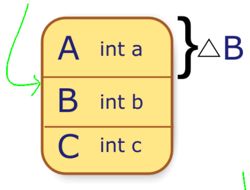
```
%c.vptr = bitcast %class.C* %c to i32 (%class.B*, i32)*** ; vtbl
%1 = load (%class.B*, i32)*** %c.vptr      ; dereference vptr
%2 = getelementptr %1, i64 1               ; select g()-entry
%3 = load (%class.B*, i32)** %2            ; dereference g()-entry
%4 = call i32 %3(%class.B* %c, i32 42)
```

Multiple inheritance class diagram



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();
```

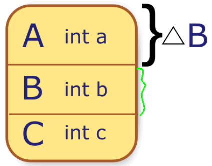



```
%class.C = type { %class.A, %class.B, i32 }
%class.A = type { i32 }
%class.B = type { i32 }
```

```
%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              ; select B-offset in C
%b = bitcast i8* %2 to %class.B*
```

Keeping Calling Conventions

```
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
```

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:
$C(A,B) \implies C \multimap A \wedge C \multimap B$

Principle 2: Multiplicity Relation

Defined by the succession of multiple parents. Example: $C(A,B) \implies A \rightarrow B$

In General:

1. Inheritance is a uniform mechanism, and its searches (→ 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

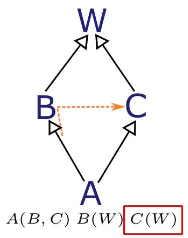
$L[A] = ABWC$

⚠ Principle 1 *inheritance* is violated

Python: classical python objects (≤ 2.1) use LPDFS!

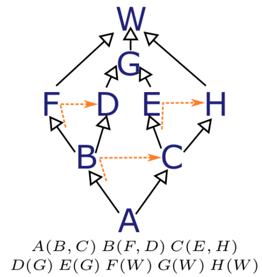
LPDFS with Duplicate Cancellation

A B ~~W~~ C W



MRO via Refined Postorder DFS

Reverse Postorder Rightmost DFS



MRO via DFS

Leftmost Preorder Depth-First Search

$$L[A] = ABWC$$

⚠ Principle 1 *inheritance* is violated

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

LPDFS with Duplicate Cancellation

$$L[A] = ABCW$$

✓ Principle 1 *inheritance* is fixed

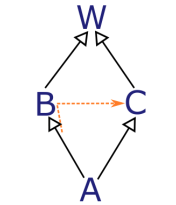
Python: new python objects (2.2) use LPDFS(DC)!

LPDFS with Duplicate Cancellation

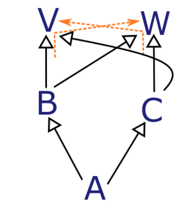
$$L[A] = ABCWV$$

⚠ Principle 2 *multiplicity* not fulfillable

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



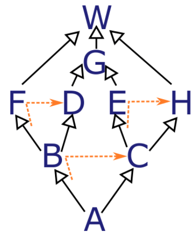
$A(B,C)\ B(W)\ C(W)$



$A(B,C)B(V,W)C(W,V)$

MRO via Refined Postorder DFS

Reverse Postorder Rightmost DFS



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

MRO via Refined Postorder DFS

Reverse Postorder Rightmost DFS

$$L[A] = ABFDCEGHW$$

✓ Linear extension of inheritance relation

RPRDFS

$$L[A] = ABCDGEF$$

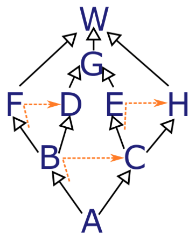
⚠ But principle 2 *multiplicity* is violated!

CLOS: uses Refined RPDFS [3]

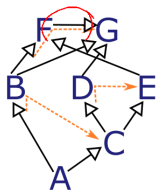
Refined RPRDFS

$$L[A] = ABCDEFG$$

✓ Refine graph with conflict edge & rerun RPRDFS!



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



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

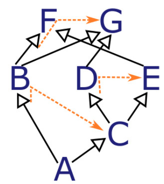
MRO via Refined Postorder DFS

Refined RPRDFS

⚠ *Monotonicity* is not guaranteed!

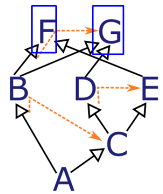
Extension Principle: Monotonicity

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 C3 Linearization



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

$L[G]$	G
$L[F]$	F
$L[E]$	$E \cdot F$
$L[D]$	$D \cdot G$
$L[B]$	$B \cdot F \cdot G$
$L[C]$	$C \cdot D \cdot G \cdot E \cdot F$
$L[A]$	⚠ 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 *Python 3*, *Perl 6*, and *Solidity*

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
- *Solidity, Python 3* and *Perl 6* with C3
- Prerequisite for → Mixins