Script generated by TTT

Title: Simon: Programmiersprachen (07.12.2012)

Date: Fri Dec 07 11:28:16 CET 2012

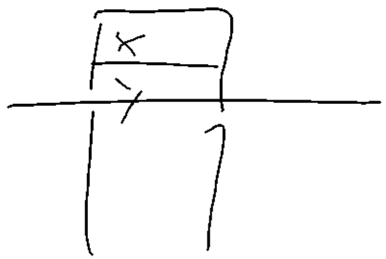
Duration: 61:21 min

Pages: 20

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

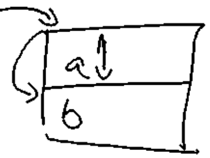

```
struct s {
    int x;
    double y;
} mystruct;
```

```
struct s a;
```




```
class A {
    int a;        size(a);
    int b;
    int m(...) { print(%b) }
}

m(A* this, ...) {
    this + size(a),
           const
}
```

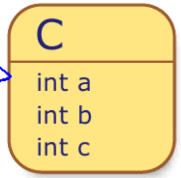


Object layout


```
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);
```

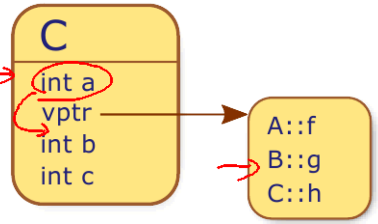


□· LLVM → IR

```
%c = alloca %class.C
%1 = getelementptr %c, i64 0, i32 0
%2 = call i32 @_g(%class.B* %1, i32 42) ; g is statically known
```


Object layout – virtual methods


```
class A {
  int a; virtual int f(int);
         virtual int g(int);   1st
         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);
```

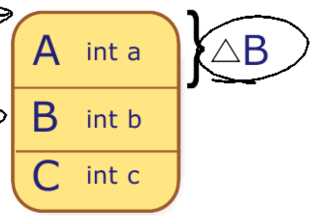



```
%c.vptr = getelementptr %c, i64 0, i64 0 ; select vptr-entry
%1 = load %c.vptr                         ; dereference vptr
%2 = getelementptr %1, i64 1              ; select g()-entry
%3 = load %2                              ; dereference g()-entry
%4 = call i32 %3(%class.B* %c, i32 42)
```


"So how do we include several parent objects?"

Multiple Base Classes

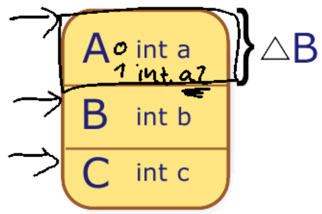

```
class A {
  int a; int f(int);
};
class B {
  int b; int g(int);
};
class C : public A , public B {
  int c; int h(int);
};
...
C c;
c.g(42);
```




```
%c = alloca %class.C
%1 = getelementptr %c, i64 0, i32 1      ; select B-offset in C
%2 = call i32 @_g(%class.B* %1, i32 %1)  ; 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);
```




```
%c = alloca %class.C
%1 = getelementptr %c, i64 0, i32 0 1 ,i32 1 ; select B-offset in C
%2 = call i32 @_g(%class.B* %1, i32 %1) ; g is statically known
```

⚠ `getelementptr` hides the ΔB here!

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

Inheritance Relation H

Defined by ancestors.

Multiplicity M

Defined by the order of multiple ancestors.

Principles

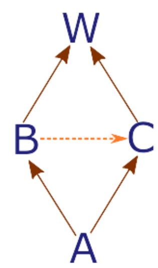
1. An inheritance mechanism (maps Object to sequence of ancestors) must follow the inheritance partial order H
2. The inheritance is a uniform mechanism, and its searches (→ total order) apply identical for all object properties (→fields/methods)
3. In any case the inheritance relation H excels the multiplicity M
4. When there is no contradiction between multiplicity M and inheritance H, the inheritance search must follow the partial order $H \cup M$.

Linearization algorithm candidates

Depth-First Search

ABWC

ABCW

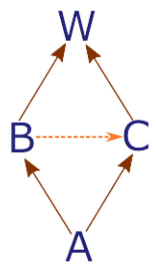


Linearization algorithm candidates

Depth-First Search

A B W C

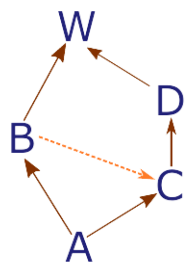
⚠ Principle 1 *inheritance* is violated



Leftmost

Breadth-First Search

ABCWD

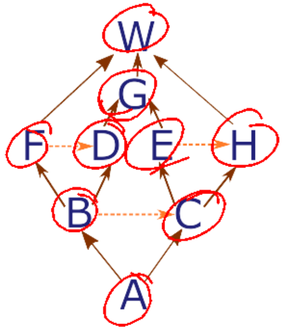


Linearization algorithm candidates

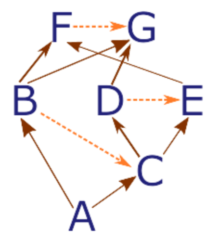
Reverse Postorder Rightmost DFS

WHGECDFBA

ABFDCEGHW



Reverse Postorder Rightmost DFS

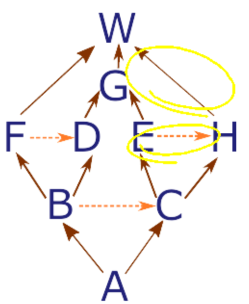


Linearization algorithm candidates

Reverse Postorder Rightmost DFS

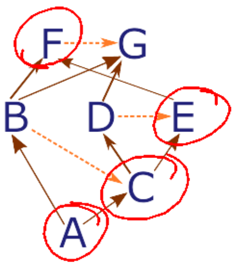
A B F D C E G H W

✓ Linear extension of inheritance relation



Reverse Postorder Rightmost DFS

FEGDCBA

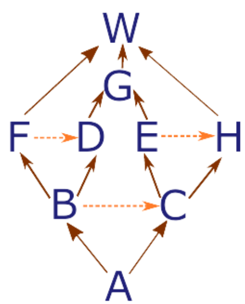


Linearization algorithm candidates

Reverse Postorder Rightmost DFS

A B F D C E G H W

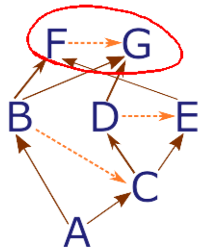
✓ Linear extension of inheritance relation



Reverse Postorder Rightmost DFS

A B C D G E F

⚠ But principle 4 *multiplicity* is violated!



Linearization Algorithm

Idea [Ducournau and Habib(1987)]

Successively perform Reverse Postorder Rightmost DFS and refine inheritance graph G with *contradiction arcs*.

The reservoir set of potential *contradiction arcs* CA is initially M, while the inheritance graph G starts from H.

do

1. $search \leftarrow \text{RPDFS}_G$
2. $CA \leftarrow \{\text{contradiction arcs of upper search}\} \cap M$
3. $G \leftarrow G \cup CA;$

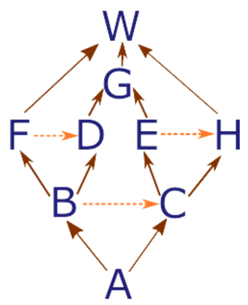
while $(CA \neq \emptyset) \ \wedge \ (search \text{ violates } H \cup M)$

Linearization algorithm candidates

Reverse Postorder Rightmost DFS

A B F D C E G H W

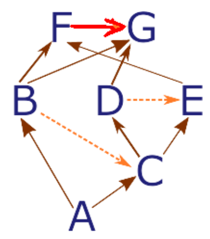
✓ Linear extension of inheritance relation



Reverse Postorder Rightmost DFS

A B C D G E F

⚠ But principle 4 *multiplicity* is violated!



Linearization Algorithm

24

Idea [Ducournau and Habib(1987)]

Successively perform Reverse Postorder Rightmost DFS and refine inheritance graph G with *contradiction arcs*.

The reservoir set of potential *contradiction arcs* CA is initially M, while the inheritance graph G starts from H.

do

1. $search \leftarrow \text{RPDFS}_G$
2. $CA \leftarrow \{\text{contradiction arcs of upper search}\} \cap M$
3. $G \leftarrow G \cup CA;$

while $(CA \neq \emptyset) \ \wedge \ (search \text{ violates } H \cup M)$

Multiple Inheritance | Implementation of Multiple inheritance | Linearization 17 / 26

Linearization vs. explicit qualification

Linearization

- No switch/duplexer code necessary
- No explicit naming of qualifiers
- Unique `super` reference

Qualification

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

Languages with automatic linearization exist

- *CLOS* Common Lisp Object System
- Prerequisite for → Mixins

Linearization vs. explicit qualification

Linearization

- No switch/duplexer code necessary
- No explicit naming of qualifiers
- Unique `super` reference

Qualification

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

Languages with automatic linearization exist

- *CLOS* Common Lisp Object System
- Prerequisite for → Mixins