Programming Languages HW 4

Jiraphon Yenphraphai (jy3694)

Augutst 6, 2022

1 Unification

1.1 1

d(15) & c(15)
This expression doesn't unify as the two structures unify if and only if they have the same functor. In this case, it has d functor and c functor.

1.2 2

4 & X & 76
This expression can unify. Because the unification operator is left-associative, X will be evaluated to 4 first. Then, it will be reevaluated to 76. X=76.

1.3 3

a(X, b(3, 1, Y)) & a(4, Y)
This expression doesn't unify as the two structures unify if and only if they have the same functor and the same number of arguments and the corresponding argument unify recursively. In this case, we cannot unify b(3, 1, Y)=Y. There is no match when applying recursion infinitely. This situation can be caught by prolog with an occurs check.

1.4 4

b(1, X) & b(X, Y) & b(Y, 1)
This expression can unify. First, we unify b(1, X) = b(X, Y) since it has the same functor, same arity. X = Y, Y = 1. Then, we unify b(X, Y) = b(Y, 1). Again, X = Y, Y =1.

1.5 5

a(1,X) & b(X,Y) & a(Y,3)
This expression doesn't unify as the two structures unify if and only if they have the same functor. In this case, it has a functor and b functor.

1.6 6

a(X, c(2, B, D)) & a(4, c(A, 7, C))
This expression can unify. X = 4, A = 2, B = 7, D = C

1.7 7

e(c(2, D)) & e(c(8, D))
This expression doesn't unify as a constant unifies only with itself. 2 cannot unify with 8.

1.8 8

X & e(f(6, 2), g(8, 1))
This expression can unify. X = e(f(6, 2), g(8, 1)).

1.9 9

b(X, g(8, X)) & b(f(6, 2), g(8, f(6, 2)))
This expression can unify. X = f(6, 2).

1.10 10

a(1, b(X, Y)) & a(Y, b(2, c(6, Z), 10))
This expression doesn't unify as it has different arity. In this case, it has the first term of b functor has 2 arguments whereas the first term of b functor has 3 arguments.

1.11 11

d(c(1, 2, 1)) & d(c(X, Y, X))
This expression can unify. X = 1, Y = 2.

2 Virtual Functions

2.1 1

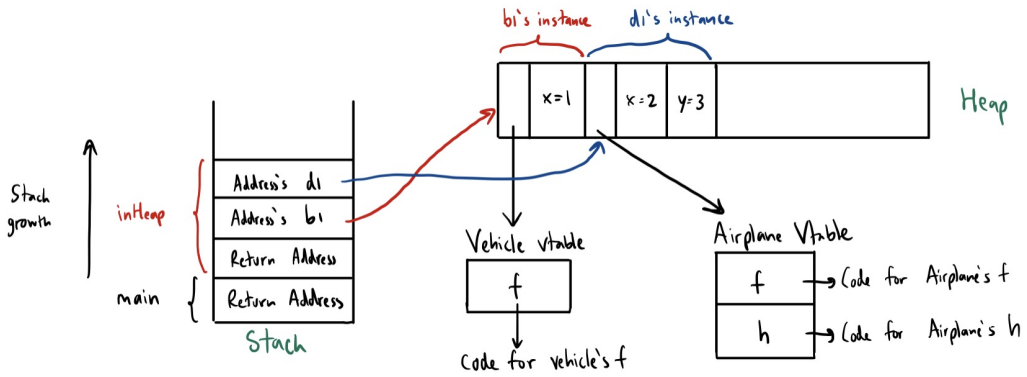

Figure 1: This figure shows the stack, heap and vtables just before `b1=d1`.

Before `b1 = d1` is executed, there are two subroutines called and unfinished, `main` and `inHeap`. There are 2 subroutines stored in stack as shown in the left side of Fig. 1. Since both `b1` and `d1` are on heap, the addresses of these objects are stored in the stack where these pointers points to `b1` and `d1` in heap shown in the red and blue arrows. The representation of `b1` begins with the address of the vtable for the class `Vehicle`. The `Vehicle` vtable consists of the address for the code of `Vehicle::f`. `b1` also stores `x = 1`. The representation of `d1` begins with the address of the vtable for the class `Airplane`. The `Airplane` vtable consists of the addresses for the code of `Airplane::f` and `Airplane::h`. `d1` also stores `x = 2` and `y = 3`.

2.2 2

Assuming that the subroutine `inHeap` is still on the stack. Thus, the stack is the same as fig. 1. The differences are on the heap and vtables as shown in Fig. 2. Since `b1` is assigned with `d1`, `b1` now points to an object of a derived type `d1`. It will use prefixes of `d1`'s field `x` and vtable. The value of `b1.x = 2`. `b1` vtables pointer also points to `Airplane` vtables. Note that `b1` cannot call `Airplane::h`. Suppose we call function `f` through `b1 (b1 → f)`. It will call code for `Airplane::f`, rather than `Vehicle::f`.

2.3 3

Fig. 3 shows the stack, heap and vtables before `b2=d2`. `InHeap` was already done and popped out of the stack, and `OnStack` is in stack instead. There are no objects on the heap in this case as `b2` and `d2` are placed in a variable-size area at the top of the frame. Again, the representation of `b2` begins with the address of the vtable for the class `Vehicle`. The `Vehicle` vtable consists of the address for the code of `Vehicle::f`. The value of `b2.x = 4`. The representation of `d2` begins with the address of the vtable

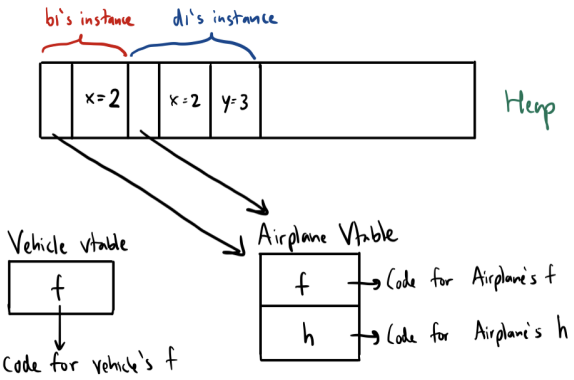

Figure 2: This figure shows the stack, heap and vtables just after `b1=d1`.

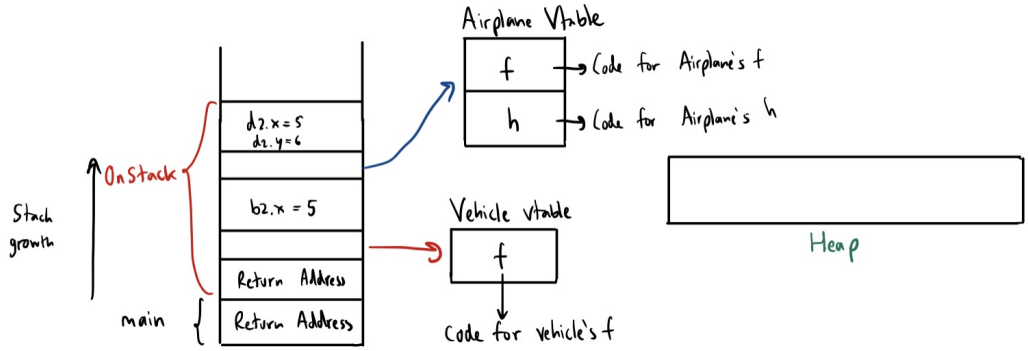

Figure 3: This figure shows the stack, heap and vtables just before `b2=d2`.

for the class `Airplane`. The `Airplane` vtable consists of the addresses for the code of `Airplane::f` and `Airplane::h`. `d2` also stores `x = 5` and `y = 6`.

2.4 4

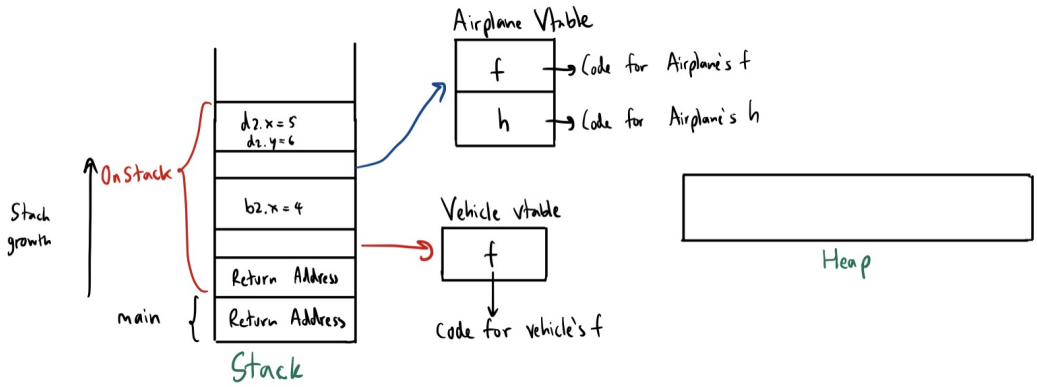

Figure 4: This figure shows the stack, heap and vtables just after `b2=d2`.

Fig. 4 shows the stack, heap and vtables just after `b2=d2` which causes `b2.x = 5`. However, C++ only allows a reference or pointer to refer to an object of a derived type. Thus, even if we assign `b2` with `d2`, hidden vtable pointer of `b2` still points to `Vehicle` vtable. Thus, when we call `f` through `b2`, it will call `f` from the base class, not the derived class.

2.5 5

`b1 = d1` and `b2 = d2` is allowed because they can use prefixes of `d1, d2`'s data space and vtable. `d1 =`

`b1` and `d2 = b2` are not allowed because both `b1` and `b2` lack the field `y` and vtable entries of an Airplane which are required for `d1` and `d2`.

3 Prototype OOLs

1. `x` field is contained locally to `obj1`.
2. `x, y` fields are contained locally to `obj2`. It inherits field `x` from the prototype `obj1`.
3. `x, y, z` fields are contained locally to `obj3`. It inherits fields `x, y` from the prototype `obj2`.
4. `x` field is contained locally to `obj4`.
5. `obj1.x` would evaluate to 20.
6. `obj2.x` would evaluate to 20 as it inherits field `x` from `obj1`.
7. `obj3.x` would evaluate to 20 as it inherits field `x` from `obj2`.
8. `obj4.x` would evaluate to 10. It hides 20 from `obj1`.
9. `obj4.y` would be undefined. There is no field `y` in `obj1` and `obj4`.
10. `obj2.y` would evaluate to 5 as it adds field `y`.
11. `obj3.y` would evaluate to 5 as it inherits field `y` from `obj2`.
12. `obj3.z` would evaluate to 30 as it adds field `z`.