ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Γλώσσες Προγραμματισμού ΙΙ

Άσκηση 11 Στατική Ανάλυση

Νταντάμης Φοίβος

03111079

3.1

```
class A {
	A  self() {
		return this;
	}

	public void printA () {
		System.out.println("In A");
	}
}

class B extends A {
	public void printB() {
		System.out.println("In B");
	}
}

public class WontCompile{
public static void main(String args[])
{
	new B().self().printB();
}
}
```

Method self will return an object of the class it is called from. That means that the self method of an A object will return an A object while self method of a B object will return a B object. Knowing this, B().self().printB() would not encounter any runtime error, since object B has a method printB. The compiler though, doesn't know this and because of type checking, it regards the result of new B().self to be an object of type A, exactly as it is annotated in the code of class A. Hence it will throw an error at compile time, since class A doesn't have any printB method.

3.2

```
public class MyClass {
	public static void main(String args[]) {
		Integer[] myInts = {1,2,3,4};
		Number[] myNumbers = myInts;
		myNumbers[0] = 2.79;
	}
}
```

The above program will compile without any problem. We actually fooll the compiler in a way. Because arrays are covariant in Java and Integer is a subtype of Number, Integer[] will also be a subtype of Number[]. This allows us to assign myInts to myNumbers. In reality though, myInts can only contain Integer values. That is why when we try to assign a Double number like 2.79 to myNumbers[0] we will get the following exception

```
Exception in thread "main" java.lang.ArrayStoreException: java.lang.Double
	at MyClass.main(MyClass.java:5)
```

3.9

```
var x,y;
x = alloc 1;
y = alloc (alloc 2);
x = y;
```

the above code allocates a new cell in the heap initialized with the value 1 and stores the returned pointer in a variable x. Then it allocates a new cell initialized with the value 2 and stores the pointer return in a new allocated cell which also return a pointer that is stored in a variable y. Lastly x's value is changed so that it points where y points which means that the cell with the value 1 isn't reachable any more by this part of code.

We have the following constraints

[[1]] = int
[[alloc 1]] = ↑[[1]]
[[x]] = [[alloc 1]]
[[2]] = int
[[alloc 2]] = ↑[[2]]
[[alloc (alloc 2)]] = ↑[[alloc 2]]
[[y]] = [[alloc (alloc 2)]]
[[x]] = [[y]]

For each term τ that occurs in the above constraints we invoke MAKESET(τ). Then we will try to UNIFY all terms appearing in the constraints. So initially we have the following sets

{[[1]]}
{int}
{[[alloc 1]]}
{↑[[1]]}
{[[x]]}
{[[2]]}
{[[alloc 2]]}
{↑[[2]]}
{[[alloc (alloc 2)]]}
{↑[[alloc 2]]}
{[[y]]}

now we apply UNIFY

---

for constraint [[1]] = int

UNIFY([[1]], int):
τ_1^r = [[1]]
τ_2^r = int
UNION([[1]], int)

so now we have

{int, [[1]]}
{[[alloc 1]]}
{↑[[1]]}
{[[x]]}
{[[2]]}
{[[alloc 2]]}
{↑[[2]]}
{[[alloc (alloc 2)]]}
{↑[[alloc 2]]}
{[[y]]}

---

for constraint [[alloc 1]] = ↑[[1]]

UNIFY([[alloc 1]], ↑[[1]]):
    τ_1^r = [[alloc 1]]
    τ_2^r = ↑[[1]]
    UNION ([[alloc 1]], ↑[[1]])

{int, [[1]]}
{↑[[1]], [[alloc 1]]}
{[[x]]}
{[[2]]}
{[[alloc 2]]}
{↑[[2]]}
{[[alloc (alloc 2)]]}
{↑[[alloc 2]]}
{[[y]]}

---

for constraint [[x]] = [[alloc 1]]

UNIFY([[x]], [[alloc 1]]):
    τ_1^r = [[x]]
    τ_2^r = ↑[[1]]
    UNION ([[x]], ↑[[1]])

{int, [[1]]}
{↑[[1]], [[alloc 1]], [[x]]}
{[[2]]}
{[[alloc 2]]}
{↑[[2]]}
{[[alloc (alloc 2)]]}
{↑[[alloc 2]]}
{[[y]]}

---

for constraint [[2]] = int
UNIFY([[2]], int):
    τ_1^r = [[2]]
    τ_2^r = int
    UNION ([[2]], int)

{int, [[1]], [[2]]}
{↑[[1]], [[alloc 1]], [[x]]}
{[[alloc 2]]}
{↑[[2]]}
{[[alloc (alloc 2)]]}
{↑[[alloc 2]]}
{[[y]]}
---

for constraint [[alloc 2]] = ↑[[2]]

UNIFY([[alloc 2]], ↑[[2]]):
τ_1^r = [[alloc 2]]
τ_2^r = ↑[[2]]
UNION ([[alloc 2]], ↑[[2]])

{int, [[1]], [[2]]}
{↑[[1]], [[alloc 1]], [[x]]}
{↑[[2]], [[alloc 2]]}
{[[alloc (alloc 2)]]}
{↑[[alloc 2]]}
{[[y]]}
---

for constraint [[alloc (alloc 2)]] = ↑[[alloc 2]]

UNIFY([[alloc (alloc 2)]], ↑[[alloc 2]]):
τ_1^r = [[alloc (alloc 2)]]
τ_2^r = ↑[[alloc 2]]
UNION ([[alloc (alloc 2)]], ↑[[alloc 2]])

{int, [[1]], [[2]]}
{↑[[1]], [[alloc 1]], [[x]]}
{↑[[2]], [[alloc 2]]}
{↑[[alloc 2]], [[alloc (alloc 2)]]}
{[[y]]}
---

for constraint [[y]] = [[alloc (alloc 2)]]

UNIFY([[y]], [[alloc (alloc 2)]]):
τ_1^r = [[y]]
τ_2^r = ↑[[alloc 2]]
UNION ([[y]], ↑[[alloc 2]])

{int, [[1]], [[2]]}
{↑[[1]], [[alloc 1]], [[x]]}
{↑[[2]], [[alloc 2]]}
{↑[[alloc 2]], [[alloc (alloc 2)]], [[y]]}
---

for constraint [[x]] = [[y]]
UNIFY([[x]], [[y]]):

$\tau_1^r = \uparrow$[[1]]

$\tau_2^r = \uparrow$[[alloc 2]]

unification failure

(since we have two proper types e.g pointers, with different type constructors)