

CS164: Midterm Exam 2

Fall 2004

- Please read all instructions (including these) carefully.
 - **Write your name, login, and circle the time of your section.**
 - Read each question carefully and think about what's being asked. Ask the proctor if you don't understand anything about any of the questions. **If you make any non-trivial assumptions, state them clearly.**
 - You will have 1 hours and 20 minutes to work on the exam. The exam is closed book, but you may refer to your one page of handwritten notes.
 - Solutions will be graded on correctness and **clarity**, so make sure your answers are neat and coherent. Remember that if we can't read it, it's wrong!
 - Each question has a relatively simple and straightforward solution. **We will deduct points if your solution is far more complicated than necessary.** Partial answers will be graded for partial credit.
 - Turn off your cell phone.
 - Good luck!
-

NAME and LOGIN: _____

Your SSN or Student ID: _____

Circle the time of your section: 10:00 11:00 2:00 4:00

Q1 (50 points)	Q2 (50 points)	Total (100 points)

1 Dataflow analysis

This question asks you to develop a simple dataflow analysis that can be used to optimize Decaf programs by removing from them unnecessary null checks.

Background. Because dereferencing a null pointer crashes the program, in Decaf, a run-time check ensures that a null pointer is never dereferenced. This check takes place just before the pointer is dereferenced using the “.” operator. For example, the statement “`p.f()` ;” might be translated by the compiler to

```
if (p == null) print_error_and_exit();
p.f();
```

The null checks can be costly, but luckily we can optimize some of them away. Specifically, a check can be safely removed if the compiler can guarantee that the pointer being dereferenced will not be null for any program input.

The problem. In this problem, you will develop a dataflow analysis to identify such redundant checks, so that they can be eliminated by the compiler. In particular, for every pointer dereference, you want to know whether the variable being dereferenced may be null; if it may not, then you can eliminate the null check.

Simplifications. In order to simplify the analysis, you can make the following assumptions:

- Assume that the method that you are going to analyze has only three local variables named `x`, `y`, `z`.
- These variables are initialized to `null`.
- All three variables are of type `B`, where `B` is a class defined as follows

```
class B {
    void foo();
    B moo();
}
```

- The return value of `moo` can be either a `null` pointer or a valid pointer.
- Within a basic block, the only valid statements are an assignment into a local variable and a method call (like `z.foo()` ;).
- The right hand side of an assignment can be any of the following:
 - A method call (like `z.moo()`),
 - `new B()`,
 - Another variable,
 - `null`.

The questions.

1. (4 points) Give a program with 3 statements or less where the null-check is unnecessary, and explain why it is unnecessary. (Make sure your example uses only the language constructs listed on the previous page.)

Answer: `a.foo(); a.foo();` The null-check is unnecessary on the second call to `foo`. `B b = new B(); b.foo();` The null check is unnecessary because we know that `b` is not null.

2. (4 points) Dataflow analysis can be classified as either forward or backward. Is your analysis going to be forward or backward? Explain why.

Answer: Forward. We want to base our decision on eliminating the null-check or not by looking at the instructions executed prior to this point.

3. (5 points) As mentioned earlier, your goal is to find, for all pointer dereference expressions, whether the pointer may be null. Give the values that the analysis will use to represent the fact that the pointer is (is not) null. Give the meaning of these values. (Hint: recall that in Liveness analysis we used two such values for each variable: "x is live" and "x is not live". Recall how we defined their meaning.)

Answer:

null: the variable may be null

not null: the variable must not be null

4. (3 points) What is the ordering of these two values (for now, assume that you are analyzing each variable independently).

Answer:

$$\text{nonnull} < \text{null}$$

5. (4 points) Assume now that you want to do the analysis for all three variables simultaneously. With this in mind, define the lattice you will use for the dataflow analysis (i.e. show the values that you are going to be assigning to program points, and the ordering you are going to impose on them).

This question was annulled, but the lattice would consist of triples, (vx, vy, vz) where each term vx, vy and vz would be either *null* or *nonnull*, depending on whether variables `x, y` or `z` are null or not null. The order would be defined as follows: $(vx, vy, vz) \leq (vx', vy', vz')$ iff $(vx \leq vx' \text{ and } vy \leq vy' \text{ and } vz \leq vz')$.

6. (12 points) If you are using a forward analysis, specify for each of the following statements how the value at program point B is computed given the value at the program point A (i.e. how do you decide what variables may be null at B given what you know about the variables at A).

If you are using a backward analysis, specify how you are going to compute the value at program point A from the value at program point B.

Your analysis should not be overly conservative. You can assume you are analyzing each variable separately.

<p>(A) x = null; (B)</p> <p>Ans: x is null at (B) y and z are the same as at (A)</p>	<p>(A) x = new B(); (B)</p> <p>Ans: x is notnull at (B) y and z remain the same</p>	<p>(A) x = y; (B)</p> <p>Ans: x at (B) is whatever y was at (A). y and z remain the same.</p>	<p>(A) x = y.moo(); (B) (hint: If y were null, you would never reach point (B))</p> <p>Ans: x is null at B y is notnull at B z remains the same.</p>
--	---	---	--

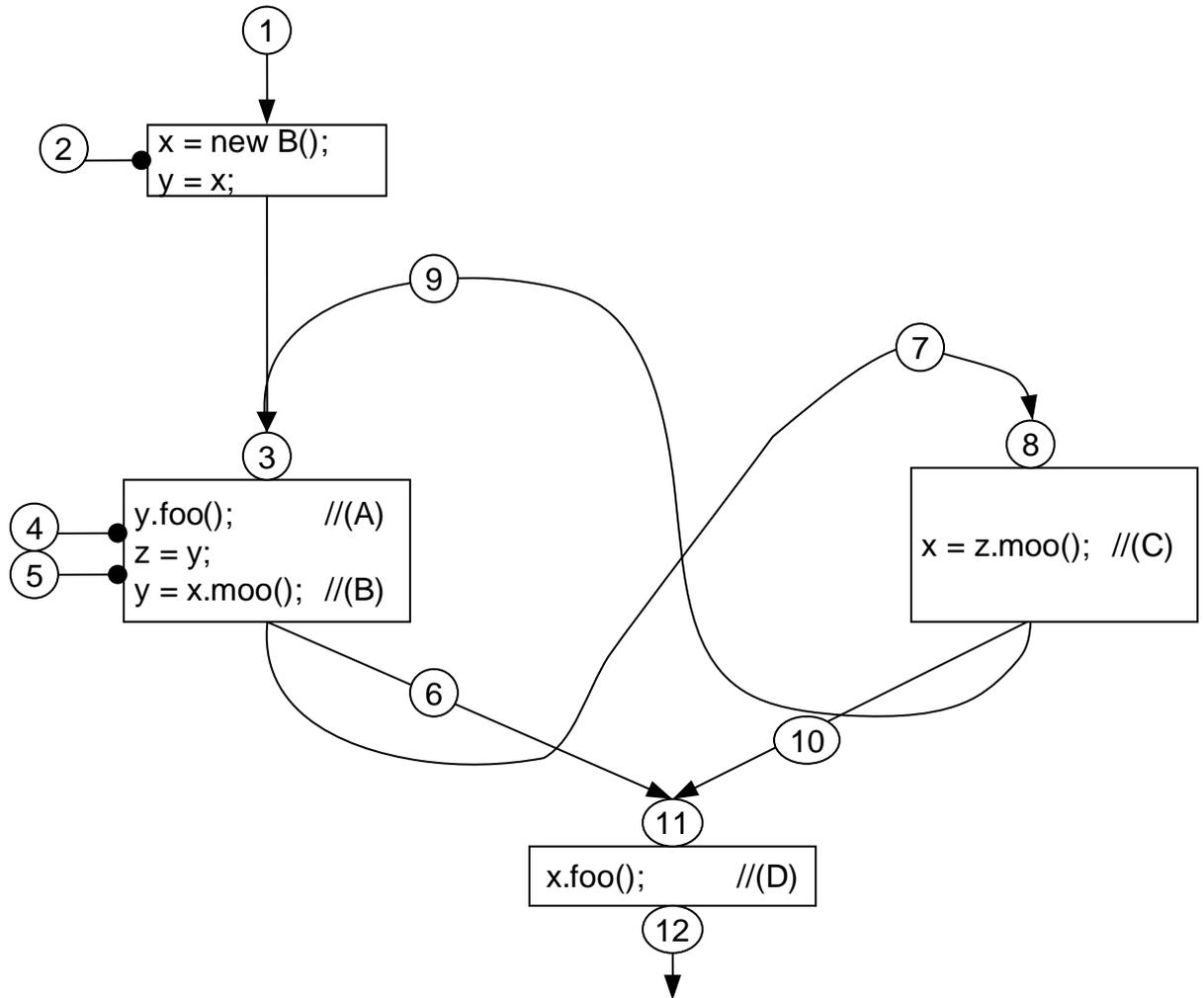
7. (4 points) How are you going to handle join points (or split points if you are doing a backward analysis)? In particular, if you have two different values coming into a join point from two different edges, how are you going to define the value for the join point?

Take the least upper bound (Which in this case is the same as the maximum) of all branches for each variable. i.e. if any branch is null, the value of the join point is null.

8. (4 points) What value are you going to initialize all program points too? What about the entry point (or the exit point if you are running a backward analysis)?

You must initialize all internal program points to notnull. Otherwise the analysis will be overly conservative. The entry point has to be initialized to null, since you were told all variables start out null.

9. (10 points) Run your data-flow analysis on the graph given below, and fill out the two tables provided.



Program point	Analysis value
1	x,y,z=null,null,null
2	nonnull,null,null
3	null,null,null
4	null,nonnull,null
5	null,nonnull,nonnull
6	nonnull,null,nonnull
7	nonnull,null,nonnull
8	nonnull,null,nonnull
9	null,null,nonnull
10	null,null,nonnull
11	null,null,nonnull
12	nonnull,null,nonnull

Pointer Dereference	safe to remove the check? (Yes/No)
(A)	NO
(B)	NO
(C)	YES
(D)	NO

2 Run-time Organization for Interfaces

This question asks you to extend Decaf with Java-style interfaces. Your main goal is to design dispatch tables needed to support interface calls, which are analogous to virtual calls.

Background. If you are not familiar with Java interfaces, don't despair. We will introduce the running example with sufficient background so that you can solve this problem.

First, you need to know that the goal of Java interfaces is to provide a limited form of multiple inheritance. This is how multiple inheritance works in Java: each Java class extends exactly one class and implements zero or more interfaces. (The only exception is the special class `Object`.)

Now, what is an interface? For the purpose of this question (we simplify Java somewhat), a Java interface is simply a class whose members are all abstract methods.

Also, what does it mean to implement an interface? A class C is said to implement an interface I when C implements (that is, defines bodies of) all methods of I .

Example. The simple example below defines two interfaces (`IK` and `IM`) and three classes that implement (some) of the two interfaces. Notice that class `C` doesn't itself define method `m`, yet it is considered to implement the interface `IM`. Make sure you understand why this is so.

```
public interface IK {
    abstract public void k();
}
public interface IM {
    abstract public void m();
}
public class A implements IK, IM {
    public void k() { System.out.println("A:k"); }
    public void m() { System.out.println("A:m"); }
}
public class B implements IK {
    public void k() { System.out.println("B:k"); }
    public void m() { System.out.println("B:m"); }
}
public class C extends B implements IM {
    public void k() { System.out.println("C:k"); }
}
```

Interface references. To make use of interfaces, Java introduces *interface reference* variables, which are like object reference variables, except that an interface variable of type *I* is allowed to point to objects of *any* type *T* such that the class *T* implements the interface *I*. For example, the interface reference variable *t* of type *IM* (introduced above) is allowed to point to an object of type *A* or *C* but not of type *B*, even though *B* happens to contain a method *m()*:

```
IM t;  
t = new A(); // LEGAL  
t = new B(); // ILLEGAL (even though B contains method m)  
t = new C(); // LEGAL  
  
t.m(); // LEGAL  
t.k(); // ILLEGAL (even though t is guaranteed to point  
// to an object with method k)
```

Compiling interfaces. Compiling Java interfaces boils down to supporting a new kind of method call *t.m()*, named *interface method call*. To the programmer, this call behaves like the *virtual method call* used to invoke instance methods, except that the object on which you invoke the method *m* comes from an interface reference variable *t*, as opposed to from an object reference variable.

Example (continued). The code below makes both kinds of calls.

```
B b; // b can point to any object of type B or subclass of B  
IK ik; // ik can point to any object that implements interface IK  
  
if (x==1) {  
    b = new B();  
    ik = new A();  
} else {  
    b = new C();  
    ik = new C();  
}  
b.k(); // virtual call  
ik.k(); // interface call
```

When *x==1*, the program outputs

```
B:k  
A:k
```

Turn over.

The questions.

1. (3 points) What output does the example print when $x != 1$?

C:k

C:k

2. (3 points) In the table below, indicate for each class which interfaces it implements. Note that a class is considered to implement an interface if its superclass implements the interface. It may help if you draw the class hierarchy in which you show with edges which class implements which interfaces.

class	A	B	C
implemented interfaces	IK, IM	IK	IK, IM

3. (3 point) In the table below, indicate which methods can be invoked on a reference variable of a particular static type. For example, list method `m` in column `A` if the following code is legal: `A a; ...; a.m();`

static type	A	B	C	IK	IM
methods that can be invoked	k, m	k, m	k, m	k	m

4. (3 point) What types of objects can `b` and `ik` point to at run time?

call statement	<code>b.k()</code> virtual call	<code>ik.k()</code> interface call
dynamic types of the object	B, C	A, C

5. (4 point) Looking at the immediately preceding table, what is the property that the types of the virtual call have that the types of the interface call don't?

C is a subclass of B. (A is not a subclass of C. C is not a subclass of A.)

6. (6 points) The figures below show the (traditional) dispatch tables for two executions of the example (left: $x = 1$; right: $x \neq 1$). These dispatch tables are used in virtual calls. From these two figures, it appears that virtual dispatch tables can be used to correctly implement interface calls. However, this impression is incorrect.

To show why virtual dispatch tables are insufficient, modify our running example (given on pages 6 and 7) so that the dispatch tables don't work in interface calls. Write your modification in the blank space below.

To reflect your modification to the example, modify also the dispatch tables shown below.



Answer.

Code modification: add a method called `j()` as the first method in class B.

As a result of the code modification, the dispatch table pointed to by the object of type C now contains three pointers (to methods B:`j`, C:`k`, and B:`m`, in this order). We are assuming here that the compiler places methods in the dispatch table in the order they are given in the class.

7. (4 points) Explain (concisely) the reason why interface method calls cannot rely on the traditional dispatch tables to decide which method to invoke.

Answer: The reason is that methods that can be invoked in a particular interface call may belong to classes unrelated by inheritance, and thus these methods are not in general placed by the compiler at the same offset in dispatch tables of their classes. This mismatch of offsets prevents the use of the efficient dynamic dispatch, which relies on the called method being placed at the same offset in all dispatch tables that may be used by the call.

For example, in the call `ik.k()`, we may invoke methods A:`k` and B:`k`. These methods are unrelated by inheritance (i.e., the first doesn't override the other or vice versa), and hence can be placed by the compiler at different offsets.

Questions on this page will take you some time.

8. (14 points) Design run-time support for interface calls. Note that the new run-time support need not look like a dispatch table at all.

What you need to draw: Using the running example above, show the data structures that your compiler would generate to perform the dispatch at an interface call.

Note: To earn full credit, the space requirements of your run-time data structures must not be a function of the number of interfaces in the entire program, only a function of how many interfaces a class implements.

Simplifications. (1) Assume interfaces cannot extend each other (as they can in Java). (2) Assume the program is completely available prior to the execution, as in Decaf (in contrast, in Java, classes are loaded and compiled during the execution).

9. (5 points) If your solution stores pointers (to methods or tables) in an array, give your rules for determining the index in the array where the pointer will be stored.

(An example of a rule: in the case of “traditional” virtual dispatch tables, the rule is that the pointer to a method f is (a) placed at the same offset in the dispatch tables of all classes related by inheritance; and (b) the methods of a subclass are placed immediately below the offset of the superclass.)

10. (5 points) Show the code that your compiler would generate at an interface call. Use a pseudo-code, not an assembler. Your code should assume the run-time data structures you designed above.

Answer: First, we give a solution that's space-inefficient but good enough for a significant partial credit. The idea is to implement an *interface dispatch table (IDT)*. The IDT is nothing more than a 3-dimensional array $D[c, i, m]$ that is looked up at the time of the call. There is one copy of D for the entire program.

The three indexes into the array are (i) c : object's dynamic type, i.e., the class id obtained from the object header; (ii) i : the interface id, determined at compile time from the static type of the variable (e.g., in $ik.k()$, the interface is K , which is the static type of ik); and (iii) m : a unique method id.

The code generated by the compiler for the call $ik.foo()$ may look as follows:

```
c = ik.dynamicType;    % load type from object's header
target = D[c,K,foo];  % lookup the IDT, K and foo are compile-time constants
call [target];
```

To meet the space restrictions requirements, we need to represent the IDT more compactly than a 3-D array. One simple solution that would give you full credit is to make the IDT a 1-D array, called D' and store in the $D'[c]$ a hash map that maps each (i, m) pair to $D[c, i, m]$. The generated code may look as follows:

```
c = ik.dynamicType;    % load type from object's header
map = D'[c];
target = map.get(K,foo);
call [target];
```