Final Exam
CS164, Fall 2007
Dec 18, 2007

- Please read all instructions (including these) carefully.
- Write your name, login, and SID.
- No electronic devices are allowed, including cell phones used as watches.
- Silence your cell phones and place them in your bag.
- The exam is closed book, but you may refer to two (2) pages of handwritten notes.
- Solutions will be graded on correctness and ***clarity***. Each problem has a relatively simple and straightforward solution. Partial solutions will be graded for partial credit.
- There are 10 pages in this exam and 3 questions, each with multiple parts. If you get stuck on a question move on and come back to it later.
- You have 3 hours minutes to work on the exam.
- Please write your answers in the space provided on the exam, and clearly mark your solutions. You may use the backs of the exam pages as scratch paper. **Do not** use any additional scratch paper.

LOGIN: _____

NAME: _____

SID: _____

Problem	Max points	Points
1	40	
2	30	
3	30	
TOTAL	**100**	

Problem 1: Points-to Analysis [40 points]

You have started a company that develops high-assurance Java software. In exchange for a proof that your software does not throw any exception, a client will pay you 30-times the usual rate. This is a good deal for the client because the software is used to control brakes in their high-end cars, and any software problem may lead to expensive lawsuits. It is also a good deal for you since you believe that you can derive such a proof automatically.

One part of the proof is to show that no Class Cast exceptions can be thrown, for any inputs to your program. You decided to obtain the proof with the Andersen's algorithm for *points-to analysis* that we covered in the lecture and discussion.

As far as the *format of the proof* is concerned, you decided to provide the proof to the customer in the form of CFL reachability.

To make your task manageable, the program of interest, shown below, is similar to one from the lecture. The program contains a single cast expression (underlined), and your goal is to prove that this expression does not throw exception.

```
public class SimpleContainer {
     class Data { Object data; }

     Data a = new Data();
     void put (Object o) { a.data = o; }
     Object get() { return a.data; }

     public static void main(String [] arg) {
       SimpleContainer c1 = new SimpleContainer();
       SimpleContainer c2 = new SimpleContainer();

       c1.put(new Foo());
       c2.put("Hello");

       Object tmp = c1.get();
       Foo myFoo = (Foo) tmp;
     }
 }
```

Part 1 (4 points): Static program analysis computes facts that are guaranteed to hold when the program is executed on all possible inputs to the program. One example fact is whether a variable is constant and, if so, what is the value of that constant. What are the facts computed by *points-to* analysis?

Part 2 (8 points): When the points-to analysis finishes analyzing your program, you will examine the computed facts. What facts are of interest to you?

What must be true about these facts so that they serve as a proof that the underlined cast expression will not throw a Class Cast exception?

When will the computed facts **fail** to prove that the underlined cast expression will not throw a Class Cast exception?

Part 3 (5 points): Is the following statement true? *When the analysis fails to prove the absence of Class Cast exceptions, your program contains a bug, as there must be an input on which the program will throw an exception.* Select the correct answer and provide a justification.

a) Yes, the program must contain a bug. I know this because _____

_____

b) The program may contain a bug. If it does not, the analysis failed because _____

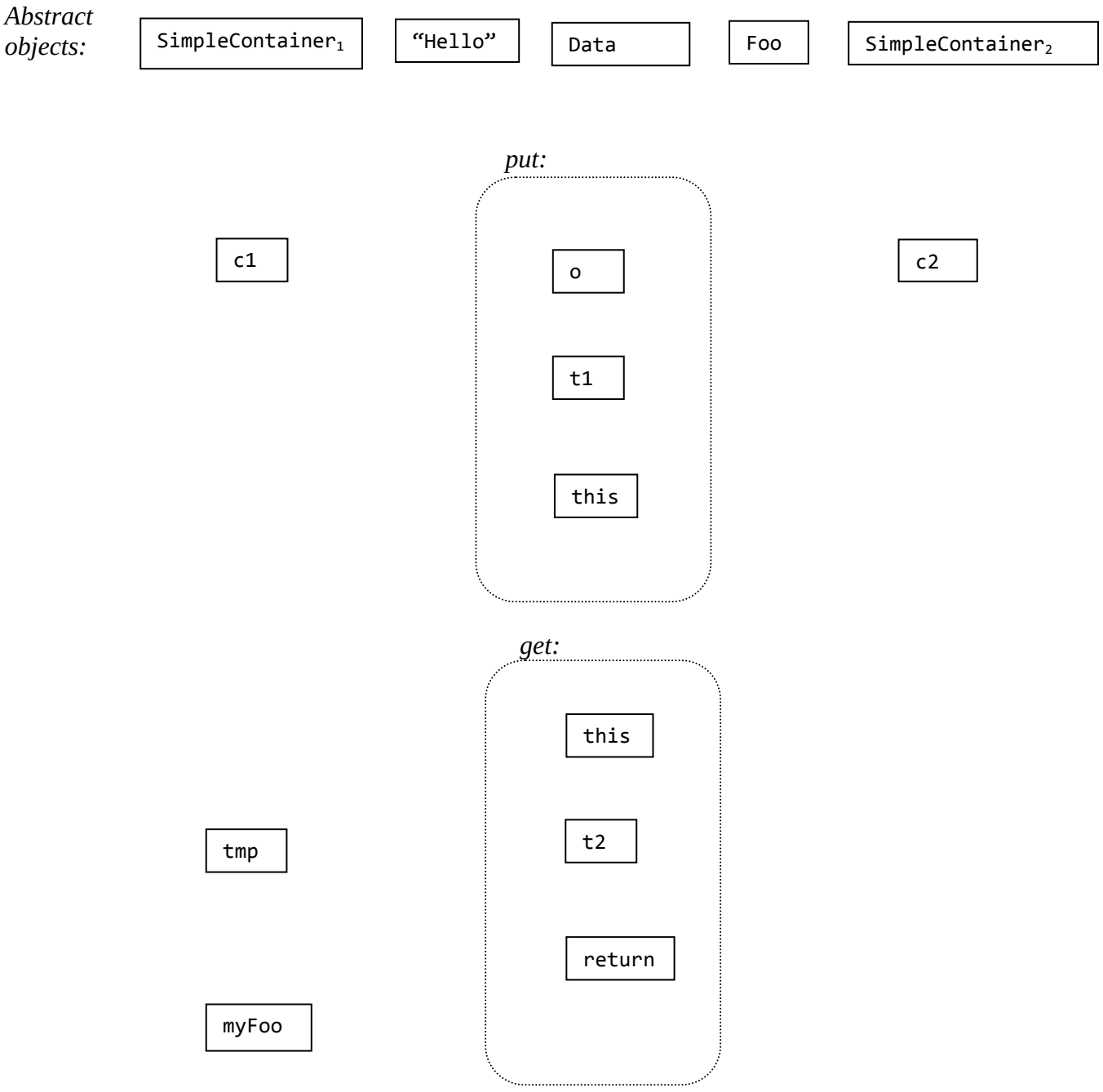
_____

c) The program definitely does not contain a bug. The analysis failed because _____

_____

Part 4 (5 points): Draw the graph representation of the above problem. This is the graph on which you will compute the CFL reachability. Label edges with the appropriate relations (assign, new, get, and put). Remember to include intermediate nodes for function parameters and returns. Draw at the bottom of the page. Some (all?) of the nodes have been pre-drawn for you.

Part 5 (10 points): Compute *PointsTo*-, *FlowsTo*-, and *Alias*-reachability. Draw all non-terminal edges that would be added by the CYK parsing algorithm.



Part 6 (3 points): Was the analysis able to prove absence of Class Cast exceptions?

YES *NO*

Part 7.1 (5 points): If *YES*, answer this question: Do we need to run the CYK algorithm (which computes the CFL-reachability) all the way to termination, or could we terminate it as soon as a suitable fact was discovered by the algorithm?

Part 7.2 (5 points): If *NO*, answer this question: Explain why the analysis failed. Is there a bug in the program or is there a limitation in the analysis? If the former, show the bug. If the latter, describe the limitation that is the root of the problem. For extra credit, rewrite the program so that it computes the same thing but the analysis succeeds on it. You are not allowed to delete any statement in the program.

Problem 2: Static and Dynamic Type Checks [30 points]

Part 1 (4 points): Indicate whether the following statements are correct.

- Static type checking may reject programs that are correct in the sense that a dynamically typed version of the program would not cause any runtime error.

  True *False*

- Static type checking may catch bugs that would show up in a dynamically typed version of the program only on a rare input.

  True *False*

Part 2 (10 points): Describe one method for guarding against stack smashing. (We discussed several methods during the lecture).

Your method (ideally, show pseudo-code on how checking is done):

Is your method static or dynamic? *static* *dynamic*

Is your method complete? That is does it catch all stack smashing attacks? *YES* *NO*

If no, what attacks may it miss?

If yes, argue that no attacks are missed:

If your method is dynamic, does it have false alarms? That is, when it reports an attack, could it be that there is no attack?

Part 3 (10 points): A friend tells you that the Java Virtual Machine (JVM) installed on many desktops implements the run-time check for an assignment into an array `p[i]=r` in the following way:

```
if (p==null) throw new NullPointerException();
if (i<0 || i>=p.length) throw new ArrayIndexOutOfBoundsException();
// symbol table tells us that p has static type T[], that is
// p is declared as follows: T p[];
if (dynamicTypeOf(r) is not compatible with T)
      throw new ArrayStoreException();
```

Another friend tells you that when the user runs a particular application, the JVM stores an important password at the address 1000. You now realize that if you can make the desktop user download your Java applet, you can read the user's password.

Write the applet that will read the value stored at the address 1000. As a hint, we provide the following code. Complete the Java method attack() such that it returns the value at the location 1000.

```
public class A { int a; }
public class B extends A { int b; }
public class C extends A { A a; }

int attack () {
      A[] a = new A[2];
      B[] b = new B[2];
      C[] c = new C[2];

      a = c;
      a[1] = new B();









}
```

Part 4 (6 points): Correct the run-time check so that it guarantees that the program is type safe (i.e., cannot not access any memory location outside an object).

Problem 3: Language Design and Implementation [30 points]

Part 1 (15 points):

Consider the following code snippet from our implementation of FlickrVision in Project 5b:

```
<div>
  <img id="img0" width={! getSize(0) !} src={! flickr[0].media.m !} />
  <img id="img1" width={! getSize(1) !} src={! flickr[1].media.m !} />
  …
</div>
```

Several groups of students wrote similar code, and suggested adding SkipJax code to CSS style specifications. This way, CSS stores not only the style but also the behavior:

```
<style type=”text/css”>
  -- The following sets the attributes for all nodes with a given DOM path
   #container img {   -- here is the DOM path: anything.container.img
       width: {! getSize(this.num) }!;
       background-image: {! ‘url(‘ + flickr[this.num].media.m + ‘)’ !}
   }
</style>
<div id="container">
  <img num=”0” />
  <img num=”1” />
  …
</div>
```

We can translate the above code to HTML + our existing SkipJax library:

```
<div id="container">
  <img num=”0” id=”n0”/>
  <script type=”text/javascript”>
        insertValue( lift(getSize, liftDots(this, ‘num’)),
                     ‘n0’, ‘style’, ‘width’ );
        insertValue(
              lift(function(v){ return ‘url(‘ +  v + ‘)’; },
                   liftDots(flickr, liftDots(this, ‘num’), ‘media’, ‘m’),
              ‘n0’, ‘style’, ‘backgroundImage’ );
  </script>
  <img num=”1” id=”n1”/>
  …
</div>
```

However, there is a bug above we’d like to fix: in the source code we intended the keyword **`this`** to correspond to the HTML node `n0`. However, in the above translation `this` is instead bound to the global context (the `document` object).

Write a Python mutating visitor methods that will modify the above target code so that `this` is bound as necessary. There are at least two ways to rewrite the above

HTML+JavaScript, so we leave this up to you. You may assume the visitor that were available in your lifting compiler. Thus you are only overriding existing methods. *Hint: While we expect it to take some time to answer this question, it is actually sufficient to modify only a couple of methods.*

Part 2 (15 points): Design an extension to HTML

(You can solve this Part even if you did not solve Part 1.) Consider again the fragment from our solution to the FlickrVision assignment:

```
<div>
   <img id="img0" width={! getSize(0) !} src={! flickr[0].media.m !} />
   <img id="img1" width={! getSize(1) !} src={! flickr[1].media.m !} />
   ...
   <img id="img4" width={! getSize(4) !} src={! flickr[4].media.m !} />
</div>
```

There is too much repetitive code in this fragment and the code is not easily modified to support a different number of `img` elements, indicating a problem with the HTML language. The problem is that HTML does not offer a notion of an *array* or *grid* of elements. As a result, we have been forced to define the `img` elements one by one. The array construct would instead allow us to create a number of elements at once, initializing them all with a single expression that defines their `width` and `src` attributes (both SkipJax expressions).

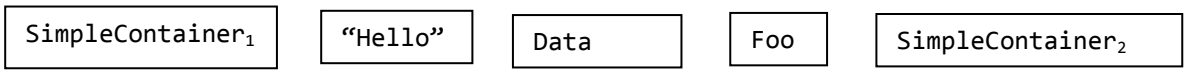
(i) Invent a suitable array extension to HTML. Rewrite the above code snippet using your new syntax and comment your code to clarify the meaning of your new construct(s).

(ii) Implement your language extension by rewriting a *source AST* into a *target AST* using a mutating visitor. Your target AST is exactly like the source AST in Project 5a: it supports HTML with SkipJax extensions. Your source AST is the target AST extended with AST nodes for HTML arrays (one new node should suffice). Your mutating visitor will rewrite these new nodes away; describe how your visitor does the rewrite:

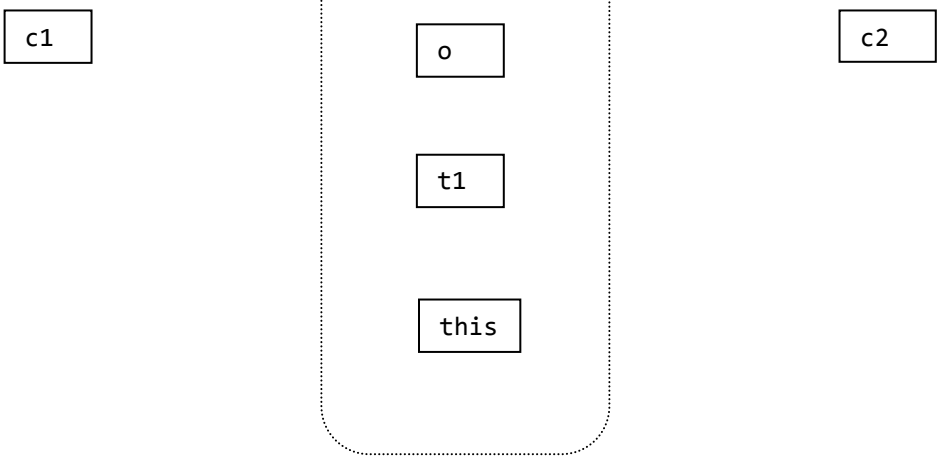
Work out your solution here. The program, for your convenience:

```
public class SimpleContainer {
    class Data { Object data; }
    Data a = new Data();
    void put (Object o) { a.data = o; }
    Object get() { return a.data; }
}
        SimpleContainer c1 = new SimpleContainer();
        SimpleContainer c2 = new SimpleContainer();
        c1.put(new Foo());        c2.put("Hello");
        Object tmp = c1.get();    Foo myFoo = (Foo) tmp;
```

Abstract objects:



put:



get: