

Homework 5

Due 5 November

Handout 9
CS242: Autumn 2008
29 October

Reading

1. Read Chapter 9, Data Abstraction and Modularity, skipping section 9.2.5 on datatype induction and the material on ML modules and CLU abstraction mechanisms (unless you are interested).
2. Read Chapter 10, Concepts in Object-Oriented Languages.
3. Read Chapter 11 on Simula and Smalltalk.

Problems

1. Templates and Polymorphism

ML and C++ both have mechanisms for creating a generic stack implementation that can be used for stacks with any type of element. In ML, polymorphic stacks could be written

```
datatype 'a stack = Empty | Push of 'a * 'a stack

fun top(Empty) = raise EmptyStack
  | top(Push(x,s)) = x

fun pop(Empty) = raise EmptyStack
  | pop(Push(x,s)) = x
```

To achieve a similar effect in C++, we could write a template for stack objects of the following form

```
template <typename A> class node {
public:
    node(A v,node<A>* n) {val=v; next=n;}
    A val;
    node<A>* next;
};

template <typename A> class stack {
    node<A>* first;
public:
    stack () { first=0; }
    void push(A x) { node<A>* n = new node<A>(x,first); first=n; }
    void pop() { node<A>* n=first; first=first->next; delete n; }
    A top() { return(first->val); }
};
```

Assume we are writing a program that uses five or six different types of stacks.

- (a) For which language will the compiler generate a larger amount of code for stack operations? Why?
- (b) For which language will the compiler generate more efficient run-time representations of stacks? Why?

2. Simula Inheritance and Access Links

In Simula, a class is a procedure that returns a pointer to its activation record. Simula prefixed classes are a precursor to C++ derived classes, providing a form of inheritance. This question asks about how inheritance might work in an early version Simula, assuming that the standard static scoping mechanism associated with activation records is used to link the derived class part of an object with the base class part of the object.

Sample `Point` and `ColorPt` classes are given in the text (Section 11.2). For the purpose of this problem, assume that if `cp` is a `ColorPt` object, consisting of a `Point` activation record followed by a `ColorPt` activation record, the access link of the parent class (`Point`) activation record points to the activation record of the scope in which the class declaration occurs, and the access link of the child class (`ColorPt`) activation record points to activation record of the parent class.

(a) Fill in the missing information in the following activation records, created by executing the following code:

```
ref(Point) r;
ref(ColorPt) cp;
r := new Point(2.7, 4.2);
cp := new ColorPt(3.6, 4.9, red);
cp.distance(r);
```

Remember that function values are represented by closures, and that a closure is a pair consisting of an environment (pointer to an activation record) and compiled code.

In this drawing, a bullet (•) indicates that a pointer should be drawn from this slot to the appropriate closure or compiled code. Since the pointers to activation records cross and could become difficult to read, each activation record is numbered at the far left. In each activation record, place the number of the activation record of the statically enclosing scope in the slot labeled “access link.” The first two are done for you. Also use activation record numbers for the environment pointer part of each closure pair. Write the values of local variables and function parameters directly in the activation records.

<i>Activation Records</i>			<i>Closures</i>	<i>Compiled Code</i>
⁽⁰⁾	r	•		
	cp	•		
⁽¹⁾ Point(...)	access link	(0)		
	x			
	y		$\langle (\quad), \bullet \rangle$	code for equals
	equals	•		
	distance	•	$\langle (\quad), \bullet \rangle$	
⁽²⁾ Point part of cp	access link	(0)		
	x			
	y		$\langle (\quad), \bullet \rangle$	code for distance
	equals	•		
	distance	•	$\langle (\quad), \bullet \rangle$	
⁽³⁾ ColorPt(...)	access link	()		
	c		$\langle (\quad), \bullet \rangle$	code for cpt equals
	equals	•		
⁽⁴⁾ cp.distance(r)	access link	()		
	q	(r)		

(b) The body of `distance` contains the expression

$$\text{sqrt}((x - q.x)**2 + (y - q.y)**2)$$

which compares the coordinates of the point containing this distance procedure to the coordinate of the point q passed as an argument. Explain how the value of x is found when `cp.distance(r)` is executed. Mention specific pointers in your diagram. What value of x is used?

- (c) This illustration shows that a reference `cp` to a colored point object points to the `ColorPt` part of the object. Assuming this implementation, explain how the expression `cp.x` can be evaluated. Explain the steps used to find the right x value on the stack, starting by following the pointer `cp` to activation record (3).
- (d) Explain why the call `cp.distance(r)` only needs access to the `Point` part of `cp` and not the `ColorPt` part of `cp`.
- (e) If you were implementing Simula, would you place the activation records representing objects r and `cp` on the stack, as shown here? Explain briefly why you might consider allocating memory for them elsewhere.

3. Subtyping of Refs in Simula

In Simula, the procedure call `assignA(b)` in the following context is considered statically type correct:

```

class A ... ; /* A is a class */
A class B ... ; /* B is a subclass of A */

ref (A) a; /* a is a variable pointing to an A object */
ref (B) b; /* b is a variable pointing to a B object */

proc assignA (ref (A) x)
begin
  x := a
end;

assignA(b);

```

- (a) Assume that *if $B <: A$ then $ref(B) <: ref(A)$* . Using this principle, explain why both the procedure `assignA` and the call `assignA(b)` can be considered statically type correct.
4in
- (b) Explain why actually executing the call `assignA(b)`, and performing the assignment given in the procedure, may lead to a type error at run time.
3in
- (c) The problem is that the “principle”, *if $B <: A$ then $ref(B) <: ref(A)$* , is not semantically sound. However, type checking using this principle can be made sound by inserting run-time tests. Explain the run-time test you think a Simula compiler should insert in the compiled code for procedure `assignA`. Can you think of reason why the designers of Simula might have decided to use run-time tests instead of disallowing `ref` subtyping in this situation? (You don’t have to agree with them; just try to imagine what rationale might have been used at the time.)
4in

4. Protocol Conformance

We can compare Smalltalk interfaces to classes using *protocols*, which are lists of operation names (selectors). When a selector allows parameters, as in `at: put:` , the selector name includes the colons but not the spaces. More specifically, if `dict` is an updatable collection object, such as a dictionary, then we could send `dict` a message by writing `dict at: 'cross' put: 'angry'`.

(This makes our dictionary definition of “cross” the single word “angry.”) The protocol for updatable collections will therefore contain the seven-character selector name `at:put:`. Here are some example protocols.

```

stack: {isEmpty, push:, pop }
queue: {isEmpty, insert:, remove }
priority_queue: {isEmpty, insert:, remove }
dequeue: {isEmpty, insert:, insertFront:, remove, removeLast }
simple_collection: {isEmpty }

```

Briefly, a stack can be sent the message `isEmpty`, returning `true` if empty, `false` otherwise; `push:` requires an argument (the object to be pushed onto the stack), `pop` removes the top element from the stack and returns it. Queues work similarly, except they are first-in/first-out instead of first-in/last-out. Priority queues are first-in/minimum-out and dequeues are doubly-ended queues with the possibility of adding and removing from either end. The `simple_collection` class just collects methods that are common to all the other classes. We say that the protocol for A *conforms* to the protocol for B if the set of A selector names contains the set of B selector names.

- (a) Draw a diagram of these classes, ordered by protocol conformance. You should end up with a graph that look’s like William Cook’s drawing shown in the text, without the dotted arrows showing inheritance.
- (b) Choose two classes, A and B, from your diagram from part (a) such that B conforms to the protocol for A. Describe briefly, in words, a way of implementing each class so that the implementation of B inherits from the implementation of A.
- (c) Choose two classes, A and B, from your diagram from part (a) such that neither conforms to the protocol of the other. Describe briefly, in words, a way of implementing each class so that the implementation of A inherits from the implementation of B.
- (d) Choose two classes, A and B, from your diagram from part (a) such that B conforms to the protocol for A. Describe briefly, in words, a way of implementing each class so that the implementation of A inherits from the implementation of B.

5. Smalltalk Run-time Structures

Here is a Smalltalk `Point` class whose instances represent points in the two-dimensional Cartesian plane. In addition to accessing instance variables, an instance method allows point objects to be added together.

class name	Point
superclass	Object
class variables	<i>comment: none</i>
instance variables	x y
class messages and methods	<i>comment: instance creation</i> newX: xValue Y: yValue ↑ self new x: xValue y: yValue
instance messages and methods	<i>comment: accessing instance vars</i> x: xCoordinate y: yCoordinate x ← xCoordinate y ← yCoordinate x ↑ x y ↑ y <i>comment: arithmetic</i> + aPoint ↑ Point newX: (x + aPoint x) Y: (y + aPoint y)

- (a) Complete the top half of the drawing of the Smalltalk run-time structure shown in Figure 1 for a point object with coordinates (3,4) and its class. Label each of the parts of the top half of the figure, adding to the drawing as needed.

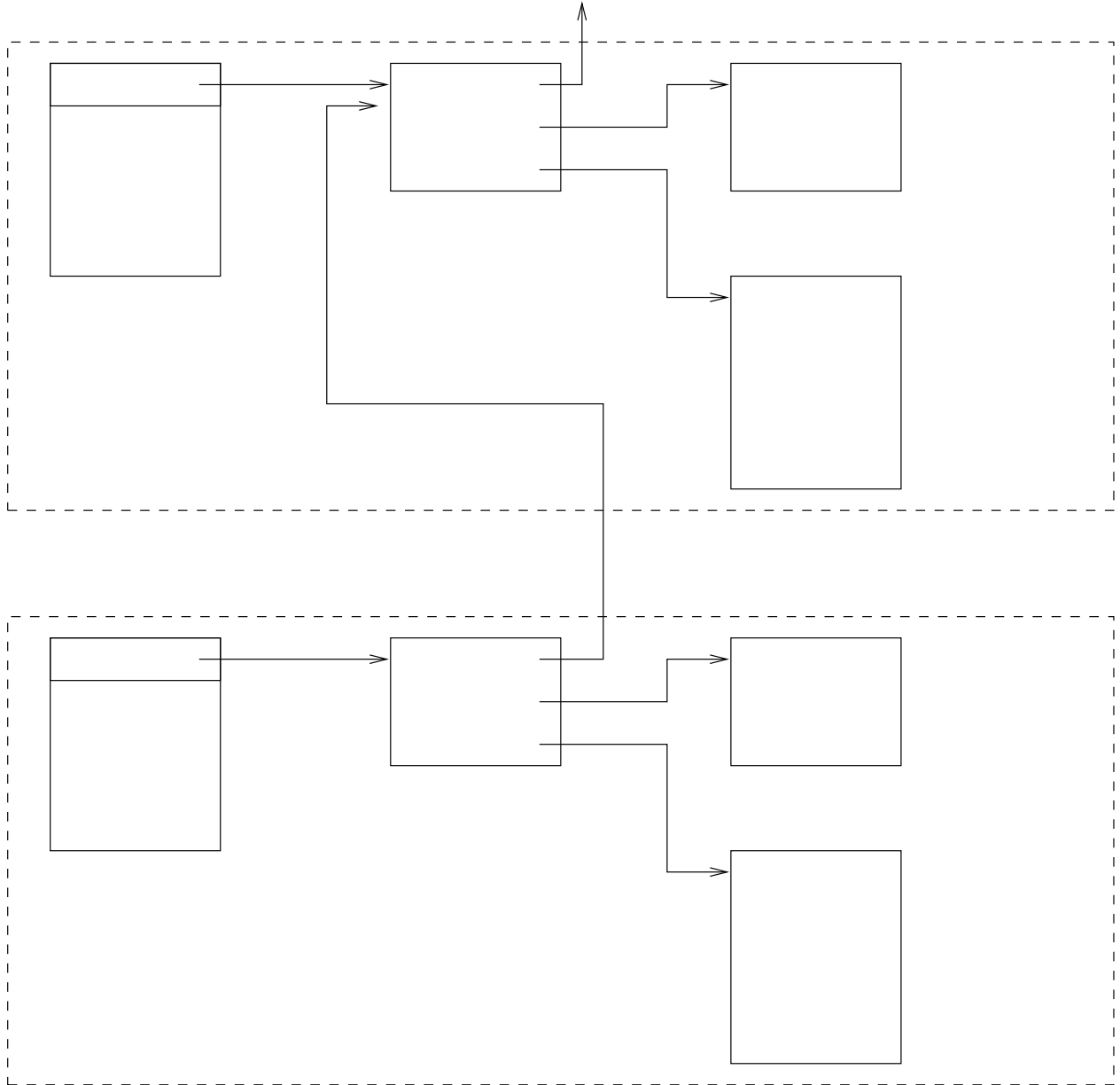


Figure 1: Smalltalk Run-Time Structures for Point and PolarPoint

- (b) A Smalltalk programmer has access to a library containing the `Point` class, but she cannot modify the `Point` class code. In her program, she wants to be able to create points using either cartesian or polar coordinates, and she wants to calculate both the polar coordinates (radius and angle) and the Cartesian coordinates of points. Given a point (x, y) in cartesian coordinates, the radius is $((x * x) + (y * y))$ `squareRoot`, and the angle is (x/y) `arctan`. Given a point (r, θ) in polar coordinates, the x coordinate is $r * (\theta \text{ cos})$ and the y coordinate is $r * (\theta \text{ sin})$
- i. Write out a subclass, `PolarPoint`, of `Point` and explain how this solves the programming problem.
 - ii. Which parts of `Point` could you reuse and which would you have to define differently for `PolarPoint`?
- (c) Complete the drawing of the Smalltalk run-time structure by adding a `PolarPoint` object and its class to the bottom half of the figure you already filled in with `Point` structures. Label each of the parts and add to the drawing as needed.