

1.(30 points) This question deals with Steensgaard's paper. For the purpose of this question, assume that you have simple programs without any procedures or calls (including to "allocate" or "op"). In other words, ignore the type rules in Figure 3 that have anything to do with "lam" and "allocate". This should leave 4 rules for you to consider.

1.One of the weaknesses of this analysis is that it allows each storage shape graph node (type) to point to at most one storage shape graph node (type). Give an example program that exposes this weakness. Also walk through the steps that Steensgaard's algorithm would take to infer the types (and thus pointer information) for this example.

Example:

```
x = &l;  
y = &m;  
p = &x;  
p = &y;
```

Initially:

```
x: t1  
y: t2  
p: t3  
l: t4  
m: t5
```

At first assignment,

```
t1 = ref(t4 x _)
```

At second assignment

```
t2 = ref(t5 x _)
```

At third assignment,

```
t3 = ref(t1 x _)
```

At fourth assignment

t1 and t2 are joined, in other words, let's replace all t2's with t1. This recursively joins t4 and t5, in other words, all t5's are replaced by t4,

This gives:

```
x: t1 = ref(t4 x _)  
y: t1  
p: t3 = ref(t1 x _)  
l: t4 = ref(bot x bot)  
m: t4
```

Bottom line: l and m are considered aliases when they should not be.

2.Give and explain an example that illustrates the added precision due to the \leq (this is the triangle with the equal sign in the paper). Also explain using the same example how doing cjoin, rather than join allows the algorithm to get the greater precision. Also argue whether or not you feel that this added precision would actually translate into better results in practice.

Example:

```
i = 10
```

```

x= i
y= i
x= &l
y= &m

```

In the above example, if we didn't have the inequality (the triangle), the assignment of *i* to *x* and *y* would force both *x* and *y* to have the same type. This in the future would force *l* and *m* to become aliased. With the inequality, we don't merge *i* with *x* or with *y*, and thus don't merge *x* and *y* (and ultimately *l* and *m*) together. *cjoin* implements exactly the conditional join discussed above. In the assignment *x = i*, if *i* is a non-pointer (i.e., bottom), it won't merge *x* and *i*. If it turns out in the future that *i* may be a pointer, then it will do the join.

My feeling is that this improved precision doesn't arrive much in practice since it requires one to assign an integer variable to pointer types. If a code does this, it probably is pretty messed up.

3. Let's say you want each storage shape graph node to point to up to two other nodes (rather than just to a single node as in Steensgaard's analysis). In this case, the types may look as follows:

```

alpha ::= beta x gamma
beta  ::= tau1 x tau2
gamma ::= lambda1 x lambda2

```

Update the four type inference rules to reflect the new type system.

There is really quite a few ways to do this. Here is one solution (for brevity, I won't write the environment $A|-$)

```

x: ref(a1)
y: ref(a2)
a2<=a1
-----
welltyped (x=y)

```

i.e., first rule doesn't change. *y* is pointing to two things. *x* must also point to the same two things.

```

x: ref(t1 x t2 x _ x _)
y: t
t<= t1 or t <= t2
-----
welltyped(x = &y)

```

i.e., *y* must be one of the two things that *x* can point to

```

x: ref(a1)
y: ref(ref(a2) x ref(a3) x _ x _)

```

a2<= a1 and a3 <= a1

welltyped(x = *y)

i.e., x must point to both "sets" of things that *y points to

x: ref(ref(a1) x ref(a2) x _ x _)
y: ref(a3)
a3<= a1 and a3 <= a2

welltyped(*x = y)

i.e., "both" things that x points to must point to what y points to

caveat: the join/cjoin, of course here is a bit more complicated. If you are joining two pairs, then how do you join them?

2.(20 points) Give an example where Lackwit would report a problem when the code is actually correct.

```
zero= 0;  
student_id = zero;  
score= zero;
```

Let's say one uses a variable "zero" to initialize student_id and score. Then lackwit would give both student_id and score the same type (which is the same type as zero). So this will appear to be a problem.

3.(20 points) Give an example where Steensgaard's pointer analysis would compute more precise results than Diwan et al's tbaa (assume SMFieldTypeDecl). Give another example where SMFieldTypeDecl would yield better results than Steensgaard's analysis.

Assume a type-safe language.

Steensgaard more precise than TBAA:

```
T *t1, *t2; (t1 and t2 are both declared to point to objects of type T)  
t1= new T;  
t2= new T;
```

Steensgaard will say that t1 and t2 do not point to the same thing. SMFieldTypeDecl will say that they could both point to the same thing since they are both pointers to T.

TBAA more precise than Steensgaard:

```
T *t;  
S *s;  
U *u;
```

S and U are immediate subtypes of T.

After the code:

```
s = new S;  
u = new U;
```

t = s;

t = u;

Steensgaard will say that s and u may point to the same thing. TBAA will say that s and u point to different things because their types are incompatible.