1 encoding

1.1 Introduction

For the type system that contains "Ostrusted" and "Osuntrusted" two types, every slot could be annotated either one of these two types. After running a program with checker-framework-inference we could get the constraints between two or three slots. In order to get the actual annotation for each slot, we can encode these derived constraints in Logiql queries, and the database will compute the every slot's annotation. In the following part, we will introduce how to encode all these constrains, and in the end we will show you an example.

1.2 Basic encoding

In order to encode every constraint appropriately, we need to create some auxiliary predicates for slots and annotations. First, a predicate called "Variable" is created for every slots.

```
Variable(v),hasVariableName(v:i)->int(i).
```

This statement means that predicate Variable is entity type, and hasVariableName is a refmode predicate for it. Every slot has exactly one name, and the name is represented by a integer.

Then we create a predicate called AnnotationOf.

```
AnnotationOf[v] = a ->Variable(v),string(a).
```

This statement means if the annotation of v is a, then v is a variable and a is a string. This predicate will store the final result we want, if we use the statement

```
print AnnotationOf
```

we can see all slots' annotation.

Next, we will show another predicate called isAnnotatedTrusted:

```
isAnnotatedTrusted[v]=i->Variable(v),boolean(i).
```

For every slot, if the annotation of it is Ostrusted, the boolean value i will be true, if the annotation of it is Osuntrusted, the boolean value i will be false. So in predicate isAnnotatedTrusted, we can get the "boolean form" of every variable's annotation, then we need to transform the "boolean form" to "string form", which is the final result we want. The following statements can perform this transformation.

```
AnnotationOf[v]="Ostrusted"<-isAnnotatedTrusted[v]=true.
AnnotationOf[v]="Osuntrusted"<-isAnnotatedTrusted[v]=false.
```

These two statements mean if a variable's boolean value in predicate isAnnotatedTrusted is true (false), then the annotation of this variable is "Ostrusted"("Osuntrusted").

The last auxiliary predicate is called SpecialVariable:

```
SpecialVariable[v]=sv->Variable(v), string(sv).
```

Here, we set two special -1 and -2, and at the beginning, we need insert the two special variables to this predicate:

```
+Variable(-1).
+Variable(-2).
+SpecialVariable[-1]="Ostrusted".
+SpecialVariable[-2]="Osuntrusted".
```

We use -1 denote the annotation "Ostrusted", and use -2 denote "Osuntrusted". Meanwhile the below assignment is also necessary:

```
isAnnotatedTrusted[v]=true <- hasVariableName(v:-1).
isAnnotatedTrusted[v]=false <- hasVariableName(v:-2).
```

The reason of we do this is that for the output from checker-framework-inference, there may be some outputs like:

```
EqualityConstraint:[@ostrusted.quals.OsTrusted,
    VariableSlot(1)]
```

That means for variable 1, we need to set the annotation of it be "Ostrusted" first of all. Together with the encoding of equality constraint, which will be introduced below, we can use the special variable to set the annotation of variable 1 be "Ostrusted":

```
+Equality[1,-1]=true.
```

And if we want to set annotation of variable 1 be false, we can use the statement:

```
+Equality[1,-2]=true.
```

1.3 Equality constraint

Now, we can focus on every constraint. First we will look at equality constraint. If the output from checker-framework-inference contains statement like:

```
EqualityConstraint:[VariableSlot(0),VariableSlot(1)]
```

This means the annotation of variable 0 and variable 1 are the same. We can encode this constraint to logiql in this way:

```
EqualityConstraint[v1,v2]=e-> Variable(v1),Variable(v2),
    boolean(e).
isAnnotatedTrusted[v2]=true <- isAnnotatedTrusted[v1]=
    true, EqualityConstraint[v1,v2]=true.
isAnnotatedTrusted[v2]=false <- isAnnotatedTrusted[v1]=
    false, EqualityConstraint[v1,v2]=true.
isAnnotatedTrusted[v2]=true <- isAnnotatedTrusted[v1]=
    true, EqualityConstraint[v2,v1]=true.
isAnnotatedTrusted[v2]=false <- isAnnotatedTrusted[v1]=
    false, EqualityConstraint[v2,v1]=true.
```

The first statement means that if the annotation of variable v1 is equal to the annotation of variable v2, then the boolean value e would be set true. And the following four statements ensure that if the annotation of two variables are same, and if we have already know one variable's annotation, we can get another variable's annotation, and save the result into predicate isAnnotatedTrusted.

1.4 Inequality constraint

In this part, we will consider about inequality constraint. If there are some outputs from checker-framework-inference like

```
InequalityConstraint: [VariableSlot(0),VariableSlot(1)]
```

This means the annotation of variable 0 and variable 1 are different. We can encode this constraint to logiql in this way:

```
InequalityConstraint[v1,v2]=n->Variable(v1),Variable(v2),
    boolean(n).
isAnnotatedTrusted[v2]=false <- isAnnotatedTrusted[v1]=
    true, InequalityConstraint[v1,v2]=true.
isAnnotatedTrusted[v2]=true <- isAnnotatedTrusted[v1]=
    false, InequalityConstraint[v1,v2]=true.
isAnnotatedTrusted[v2]=false <- isAnnotatedTrusted[v1]=
    true, InequalityConstraint[v2,v1]=true.
isAnnotatedTrusted[v2]=true <- isAnnotatedTrusted[v1]=
    false, InequalityConstraint[v2,v1]=true.
```

The first statement means that if the annotation of variable v1 is not equal to the annotation of variable v2, then the boolean value e would be set true. And the following four statements ensure that if the annotation of two variables are different, and if we have already known one variable's annotation, we can get another variable's annotation, and save the result into predicate isAnnotatedTrusted.

1.5 Subtype constraint

There is another constraint called subtype constraint, the form of this constraint output by checker-framework-inference is

```
SubtypeConstraint: [VariableSlot(0),VariableSlot(1)]
```

This means that the annotation of variable 0 is the subtype of the annotation of variable 1. The encoding is shown below:

```
SubtypeConstraint[v2,v1] = s -> Variable(v1),Variable(v2)
    ,boolean(s).
isAnnotatedTrusted[v2]=true <- isAnnotatedTrusted[v1]=
    true, SubtypeConstraint[v2,v1]=true.
isAnnotatedTrusted[v2]=false <- isAnnotatedTrusted[v1]=
    false, SubtypeConstraint[v1,v2]=true.
```

The first statement says that if the annotation of variable v2 is the subtype of annotation of variable v1, then the boolean value s would be set true. And the following two statement ensure that if the annotation of variable v2 is the subtype of the annotation of variable v1, and the annotation of variable v1 is Ostrusted, then the annotation of variable v2 is also Ostrusted. Another condition is that if the annotation of variable v1 is the subtype of the annotation of variable v2, and the annotation of variable v1 is Osuntrusted, then the annotation of variable v2 is also Osuntrusted. The reason we encode the constaint in this way is that in this type system, the subtype of Ostrusted could only be Ostrusted, and the supertype of Osuntrusted could only be Osuntrusted.

1.6 example