Final Exam, CS6848, IIT Madras

02-May-2013

1. [6] **Flow analysis I** How do method duplication and class duplication help improve the precision of flow analysis. Give examples. Describe the drawbacks using examples.
2. [6] **Flow analysis II** We will assume a simplified procedural Java subset: there can be only one class; it must have a main method, which can call many other methods. As part of compound statements we will only admit if-then-else statements. Method overloading is not allowed. We extend this Java subset with two parallel constructs: `async` and `finish`. The construct `async {S1}` creates an asynchronous task (or thread) to execute `S1`; tasks may also be nested. `finish {S}` works as a "join" point for all the tasks (even nested ones) created in `S`. Let

```
    main() {
L1:        finish { // waits for tasks created at L3, L5, L7.
L2:                S1;
L3:                async {
L4:                  S3;
L5:                  async
L6:                    foo();}
L7:                async {
L8:                  S5;
L9:                  finish { // waits for task created at L10.
L10:                    async
L11:                      foo();
L12:                    S6;
L13:                  } // end-finish
L14:                  S7}  // starts only after the finish at L9 has terminated
L15:                S8
L16:        } // end-finish
L17:        S9 // starts only after the finish at Line L1 has terminated.
... }
L18: void foo(){
L19:     S;
L20: }
```

us assume that the body of the "main" function is implicitly surrounded by a `finish async`. Thus every reachable statement in the program is part of a task and every task is inside a finish.

We will assume that every statement is labelled. Let us call the label of the outermost task surrounding the main body as `L01`, and that of outermost finish as `L00`. In the above code, `L1`, `L2`, `L3`, `L7`, `L15`, `L16` and `L17` are part of the activity labelled `L01`. Statements `L4`, `L5` are part of the activity labelled `L3`. `L6` is part of the activity `L5`. Similarly, `L8, L9, L10, L12, L13, L14` are part of the activity labelled `L7` and `L11` is part of the activity `L10`. On the other hand, `L18, L19` and `L20` are part of two tasks L10 and L5.

Our goal is to a) identify the tasks for each statement, and b) finish statements for each task. [Note: a function may be called from multiple places; thus the statements/tasks inside that function may be part of multiple async/finish statements.] The list of statements that may execute in a task gives an over approximation of the statements that may run in serial. The flow set $F : Labels \to P(Labels)$ of each label is the set of async/finish labels that may "flow into" it. Give a procedure to generate the flow constraints for any input program. Solving these constraints will help realize our goal.

Hint: For each finish or async statement you can iterate over all the statements of its body.

3. [6] **Exceptions I** Let us consider a simple expression language with, async, finish, try and raise.

$$e ::= c | e_1; e_2 | \texttt{async } e | \texttt{finish } e | \texttt{raise} | \texttt{try } e_1 \texttt{ with } e_2$$

The semantics of `async` and `finish` are similar to that explained in Q2. The behavior of a `raise` statement inside a serial expression is standard – the control is transferred to the nearest exception handler. The statement `try` e_1 `with` e_2 evaluates e_1 and transfers the control to e_2 if an exception is raised by e_1. If an exception is raised inside an `async` and not caught, then the execution of the `async` is terminated. The exception however does not interrupt other asynchronous tasks running in parallel with this `async`, and is propagated to the `finish`. The `finish` statement waits for all the asyncs to terminate. It has an implicit exception handler whose job is to raise an exception if any of the tasks has thrown an uncaught exception. The program may either terminate with an exception or return a value.

Briefly write about the types, values, type system, and operational semantics.

4. [6+2] **Exceptions II** Our goal is to introduce versioning exceptions in Java. As a first step, let us introduce versioning exceptions in a subset of Featherweight Java (FJ). An FJ program consists of a sequence of class declarations, followed by a closed expression. Provide the type rules and operational semantics; feel free to skip constructs that are not essential in the context of exceptions.

Grammar of our FJ subset:

```
L::=class C {C f;K M}          //  Class declarations
M::=C m(C x) {return e;}       //  Method declarations
K::=C(C f){this.f=f}           //  Constructor
e::=x | e.f | e.m(e) | new C(e) | (C)e |  try  (x, e) |  restore (p,q)
```

[Bonus]: (*Attempt at the end.*) Briefly describe a scheme to automatically infer the locations to save/restore.

5. [6] **Type system for program analysis** Here is a new expression language that can be used to encode synchronized accesses using locks.

$$e ::= e_1; e_2 | lock\ Obj_i | unlock\ Obj_i | \epsilon$$

- Obj_i represents a predefined object, which can be used as the lock handle. Assume $1 \leq i \leq n$, where n is a constant.
- `lock` Obj_i: tries to take a lock on Obj_i.
- `unlock` Obj_i: tries to release the lock on Obj_i.
- It is illegal to lock an already locked object.
- Similarly, it is illegal to unlock an object that is not locked.

Write a type system that guarantees that a well typed program don't perform any illegal accesses. You can assume that no locks are taken, to start with. Examples accesses:

- `lock Obj1; lock Obj2; unlock Obj2; unlock Obj1` – should type check.
- `lock Obj1; lock Obj2; unlock Obj1; unlock Obj2` – should type check.
- `lock Obj1; unlock Obj2` – should not type check.
- `lock Obj1; lock Obj1` – should not type check.

6. [3] (*Bonus*) Choose ONE of the following papers that we discussed in the class, and critically analyze how it is limited and how it can be extended/improved.

   (a) Communicating Sequential Processes

   (b) Eiffel an introduction (focus on annotations)

   (c) Proof Carrying Code

   (d) Type safe method inlining.

   (e) Type preserving garbage collectors.