Lecture notes on Partial Evaluation – page 1

Consider the power function, which raises x to the n-th power. We can write this function iteratively or recursively:

```
int pow(n, x) {
    int result = 1;
    while (n > 0) {
        result *= x;
        n -= 1;
    }
    return result;
}
```

```
int pow(n, x) {
    if (n > 0)
        return  x * pow(n – 1, x);
    else
        return 1;
}
```

You know how to execute the pow procedure given two arguments, e.g. pow(3, 6) = 6^3 = 216. You evaluate this by first testing if n is greater than zero, choosing the right branch, etc…

But what if you know n but do not x? Can you still execute the procedure in part? Imagine executing the parts you can execute, while collecting up all the code that cannot be executed without knowing x. Lets call this new program pow_n(x) , for example pow_3(x) = pow(3, x)

Write down the result, for both the iterative and recursive cases. Note that you may need to define additional functions.

```
int pow_3(x) {
```

```
int pow_3(x) {
```

Lecture notes on Partial Evaluation - page 2

Now consider a simple HP-style calculator using post-fix operators: 3+4 is expressed as {3, 4, "+"}. The calculator has two registers A and B. The calculator takes an expression and values for A and B, then executes the commands in order. Each command is either an integer, an arithmetic operation (+,-,*,/) or a command to load the value of A or B.

```
int calc(object[] prog, a, b) {
    int[] stack = new int[100];
    int top = -1;
    for each (cmd in prog)
        if (cmd instanceof Integer)
            stack[++top] = cmd;
        if (cmd == "+") {
            int x = stack[top--];
            int y = stack[top];
            stack[top] = x + y;
        }
        …  // same for -, *, /
        if (cmd == "A")
            stack[++top] = a;
        if (cmd == "B")
            stack[++top] = b;
    }
    return stack[0];
}
```

Again, we can call the calc function normally:

```
calc( { 6, "A", "*", "B", "+"}, 5, 2)
    = 6*5 + 2
    = 32
```

The interesting thing is what happens when A and B are not known:

```
calc( { 6, "A", "*", "B", "+"}, ?, ?) = ???
```

Use the technique for partial evaluation to determine the value of calc in this case.

Lecture notes on Partial Evaluation – page 3

The process of evaluating a program with partial inputs is called *partial evaluation*. The result of partial evaluation is a new *program*, not a single integer value. The program contains all the parts of the original program that cannot be executed, due to missing inputs. One thing that helps is to annotate (underline) the parts of the program that only depend on the values that are known. These are the parts that can be removed.

```
int pow(n, x) {
    int result = 1;
    while (n > 0) {
        result *= x;
        n -= 1;
    }
    return result;
}
```

```
int pow(n, x) {
    if (n > 0)
        return  x * pow(n – 1, x);
    else
        return 1;
}
```

The resulting program is called the *residual* code. Here is the residual code for partially evaluating the pow function with n=3:

```
int pow_3(x) {
    int result = 1;
    result *= x;
    result *= x;
    result *= x;
    return result;
}
```

```
int pow_3(x) {    return  x * pow_2(x);    }
int pow_2(x) {    return  x * pow_1(x);    }
int pow_1(x) {    return  x * pow_0(x);    }
int pow_0(x) {    return  1;          }
```

The program on the left illustrates a technique called *loop unrolling*. Because n is known to be 3, the loop will execute exactly 3 times. Thus the loop can be removed and the body of the loop duplicated. The program on the right illustrates *function specialization.* When pow(3, x) calls pow(2, x), a new specialized version of pow is needed. This process continues to create new specialized functions until n=0.

The calculator can be optimized in a similar way. Given `p1 = { 6, “A”, “*”, “B”, “+”}`

```
int calc-p1(a, b) {
    int[] stack = new int[100];
    stack[0] = 6;
    stack[1] = a;
    int x1 = stack[1];
    int y1 = stack[0];
    stack[0] = x1 * y1;
    stack[1] = b;
    int x2 = stack[1];
    int y2 = stack[0];
    stack[0] = x2 + y2;
    return stack[0];
}
```

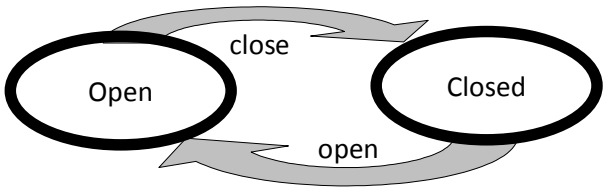
This could be further optimized by a smart compiler. Note that the local variable `top` has been removed, because its values depend only on `p 1`, not on A and B.

Calc is an interepreter. Note that the residual code looks like assembly language of a compiled version of the input commands. This is because partial evaluation of an interpreter with respect to a program generates a compiled version of the program.

Finally, lets consider a state machine interpreter.

```
void execute(StateMachine M, Scanner s) {
        run(M.getStart(), s);
}
void run(State current, Scanner s) {
    System.out.println("State: " + current.getName());
    String input = s.nextLine();
    for (Transition t : current.getOuts()) {
        if (t.getEvent().equals(input))
            return run(t.getTarget(), s);
    }
    // otherwise, not found
    System.out.println("Unknown event " + input);
    run(current, s);
    }
}
```

Consider the example state machine Door:



What would it mean to partially evaluate

```
StateMachine Door = new StateMachine();
State open = Door.addState("Open");
State closed = Door.addState("Closed");
Door.newTransition("close", open, close);
Door.newTransition("open", close, open);

execute(Door, input)
```

where Door is a known value representing the state machine.

```
class StateMachine {
  State getStart();
  State addState(String name);
  new Transition(String e,
          State from, State to);
}

class State {
   String getName();
   Transition[] getOuts();
}

class Transition {
  String getEvent();
  State getSource();
  State getTarget();
}
```

The same partial evaluation techniques can be used on model interpreters. Here is the residual code for executing the Door model. Note that the information in the model has been turned into code. The model no longer appears anywhere explicitly.

```
void execute_Door(Scanner s) {
        run_Closed(s);
}
void run_Closed(Scanner s) {
    System.out.println("State: Closed");
    String input = s.nextLine();
    if (“open”.equals(input)) {
        return run_Opened(s);
    System.out.println("Unknown event " + input);
    run_Closed(s);
}
void run_Opened(Scanner s) {
    System.out.println("State: Opened");
    String input = s.nextLine();
    if (“close”.equals(input)) {
        return run_Closed(s);
    System.out.println("Unknown event " + input);
    run_Open(s);
}
```

The code that is generated is the same code that was written explicitly by the template-based approach. Thus we can suggest:

template = interpreter + partial evaluation

If you want to know more, there is a wealth of literature on partial evaluation and its application to interpreters.