CS6848 - Principles of Programming Languages

Partial Evaluation

V. Krishna Nandivada

IIT Madras

Recap

- Modeling the store.
- Versioning Exceptions.

Announcements

- Last instructional day.
- Final exam on 28th May 11AM.
- Portion - Post mid-term.

Partial Evaluation

- Partial evaluation can be seen as “program specialization” in the presence of partial input.
- Can be used for program optimization, compilation, interpretation, and so on.
- Input: (Program, partial-input)
- Output: (Modified-program)
- (Modified-program, rest-of-the-input) $\rightarrow$ value.

Example

```
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;
}
```

- Knowing how the ‘interpreter’, you can compute `pow (3, 6)`.
- What if you know the value of one of the inputs (say `n`)’? Can you do anything?
- Can you generate code that specializes the code for specific value of `n`.

Example continued

```
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;
}
```


Partial evaluation

- The process of evaluating a program with partial inputs is called partial evaluation.
- The result of partial evaluation is a new program.
- The new program contains all parts of the original program that cannot be executed, due to missing inputs.
- The new program is thus the residual code.
- We used more than just partial evaluation - loop unrolling, and repeated function specialization/partial evaluation.

Example 2 - **Post-fix calculator with two named registers**

```
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; }
    ... //samefor -,*,/
    if (cmd == A)
       stack[++top] = a;
    if (cmd == B)
       stack[++top] = b; }
  return stack[0]; }
```

- calc (6,``A'',``*'',``B'',``+'',5,2)=32
- Say dont know the values of a and 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];
```

- Note: optimizations: top has been eliminated.

Idea of specialization

- Say, each program element (expression, function name, parameters keywords etc) may be annotated with an underline (= cannot be reduced).
- Idea:
  1. Evaluate all the non-underlined expressions.
  2. unfold all non-underlined function calls = Replace with new code.
  3. generate residual code for all underlined-expressions.
  4. generate residual function calls for all underlined function calls.

A two input program p =

```
a(m,n) = if m = 0 then n+1 else
            if n = 0 then a(m-1,1) else
            a(m-1,a(m,n-1))
```

Program p, specialized to static input m = 2:

p2 =

```
a2(n) = if n=0 then a1(1) else a1(a2(n-1))
a1(n) = if n=0 then a0(1) else a0(a1(n-1))
a0(n) = n+1
```

Sketch of an partial evaluator

- Input: Program and annotations.
  - An annotation can be: `eliminable` or `residual`.
- Output: A specialized program, that will have the same form as the original. Different:
  - definitions of specialized functions `(g, StaticValues)`.
  - `g` is part of the original program,
  - `StaticValues` – a set of (parameter, value) tuples.
  - The rest of the parameters of `g` are dynamic.
- Say the input program: `f1(s, d) = e1`. // `s` is static, `d` is dynamic.
- Read Program `P` and `s`
- Pending = {(f1, s)}; AlreadySeen = {};
- while Pending $\neq$ {}
  - Choose and remove a pair `(g, s)` from Pending.
  - Add `(g, s)` to AlreadySeen.
  - Say `g` is defined as `g(s, d) { e1 }`
  - Replace the target definition as $g_s(d)$ { Reduce(`E`)}
    - `E` = substituting the static values of parameters in `s` in `e1`.

Algorithm for Reduce

1. We will use RE to denote Reduce(`E`).
2. If `E` is a constant or a dynamic parameter then RE = E.
3. If `E` is a static parameter of `g` then RE = value of the parameter as given in `s`.
4. Say `E` = `primitiveOp(E1, ...En)`, then
   - if (v1 = Reduce`E1`, ... vn = Reduce `En`) all are reducible, then RE = value of primitiveOp(v1, ... vn).
   - Else the annotation is wrong.
5. if E is `primitiveOp(E1, ...En)` then
   - compute `E1' = Reduce(E1), ...En' = Reduce(En)`. RE = `primitiveOp(E1', ...En')`
6. Similarly if (E0) then E1 else E2 (two cases).

Algorithm contd.

1. Say `E = f(E1, ...En)`, and say `f` is defined as `f(x1, ...xn) {func-body}` then `RE = Reduce (E')`, where `E'` is obtained by substituting static parameters in the arguments and reducing `func-body`.
2. If `E = f(E1, ...En)` then
   1. For each static parameter of `f`, compute the (`static-parameter, value`) tuple and it to static-parameter-tuple list.
   2. If `value` is not a constant then the annotation is incorrect.
   3. For each dynamic parameter of `f`, invoke Reduce to compute a list of expressions.
   4. f' = a new function with parameters given by the list `new-dynamic-expressions`
   5. RE = `(f', static-parameter-tuple`
   6. if this RE is not AlreadySeen, then add it to Pending.