June 21, 2007 at 18:47

1. Intro. This is a solution for the problem ASCIRC. Please read the statement before continuing.

After reading the problem carefully I now believe that the main reason why the success rate is so low is that it is poorly phrased. In particular it is not clear if the operations are commutative or not. The drawing with the gates certainly suggests so. The description of assembly assignments (vaguely) suggests otherwise. For the example given, both interpretations work!

I believe that hash-consing is sufficient to solve this problem. All we want is to make sure that we don't count more than once the same operation and that we don't count operations whose results are discarded. We can't do any other smart stuff because we don't know anything about the 26 operations.

Hash-consing means that we make sure that structural equality implies reference equality. To my knowledge, the technique was first presented by Ershov in [*Doklady, AN USSR, vol. 118, no. 3 (1957)*] and was translated in [*Communication of ACM, vol. 1, no. 8 (1958)*]. Here, we shall build expression trees that represent what the assembly statements compute. We hash-cons the nodes of these trees.

2. The overall structure of the program is

```
#define verbose 0
#include <map>
#include <set>
#include <queue>
#include <stdio.h>
#include <assert.h>
  using namespace std;
  ⟨ Hash-consing for Nodes 7 ⟩
  int main( )
  {
    ⟨ Main body 3 ⟩;
  }
```

3. The main body is easy. We can maintain an array of expressions, each corresponding to one output. We update this array instruction by instruction (using the hash-consing mechanism). At the end we count how many non-leafs are reachable from this array and we are done.

```
#define foreach(i, c) for (typeof((c).begin( ))i = (c).begin( ); i ≠ (c).end( ); ++i)
⟨ Main body 3 ⟩ ≡
  int tests;
  scanf("%d", &tests);
  while (tests−−) {
    foreach(n, hashcons)delete n→second;
    hashcons.clear( );
    Node *out[256];
    for (char c = 'a'; c ≤ 'z'; ++c) out[c] = mk_node(c, Λ, Λ);
    int assignments;
    scanf("%d", &assignments);
    while (assignments−−) ⟨ Process one assignment 4 ⟩;
    ⟨ Count internal nodes in out and report 5 ⟩;
  }
```

This code is used in section 2.

4. To process one assignment we read its description and simply create a node for it.

```
⟨ Process one assignment 4 ⟩ ≡
  {
    char asg[5];
    scanf("%s", asg);
    out[asg[3]] = mk_node(asg[0], out[asg[1]], out[asg[2]]);
  }
```

This code is used in section 3.

5. I will use BFS to count the internal nodes.

```
#define in(e, c) ((c).find(e) ≠ (c).end())

⟨ Count internal nodes in out and report 5 ⟩ ≡
  set⟨Node *⟩internal_nodes;   /* reachable from out */
  queue⟨Node *⟩q;   /* the BFS queue */
  for (char c = 'a'; c ≤ 'z'; ++c)
    if (out[c]→left ≠ Λ) {
      if (¬in(out[c], internal_nodes)) q.push(out[c]);
      internal_nodes.insert(out[c]);
    }
  while (¬q.empty()) {
    Node *n = q.front();
    q.pop();
    Node *nn = n→left;
    ⟨ Push nn 6 ⟩;
    nn = n→right;
    ⟨ Push nn 6 ⟩;
  }
  printf("%d\n", internal_nodes.size());
```

This code is used in section 3.

```
6. ⟨ Push nn 6 ⟩ ≡
  if (nn→left ≠ Λ ∧ ¬in(nn, internal_nodes)) {
    q.push(nn);
    internal_nodes.insert(nn);
  }
```

This code is used in section 5.

7. Hash-consing is implemented using a map from **Node** (structure) to **Node** ∗ (pointer).

⟨ Hash-consing for **Node**s 7 ⟩ ≡

```
  struct Node {
    char op;
    Node ∗left;
    Node ∗right;
    Node(char op, Node ∗left, Node ∗right)
    : op(op), left(left), right(right) { }
    bool operator < (const Node &o) const
    {
      if (op ≠ o.op) return op < o.op;
      if (left ≠ o.left) return left < o.left;
      return right < o.right;
    }
  };
  map⟨Node, Node ∗⟩hashcons;
  Node ∗mk_node(char op, Node ∗left, Node ∗right)
  {
    Node ∗r = new Node(op, left, right);
    map⟨Node, Node ∗⟩ :: iterator it = hashcons.find(∗r);
    if (it ≠ hashcons.end()) {
      delete r;
      return it→second;
    }
    return hashcons[∗r] = r;
  }
```

This code is used in section 2.

8. Index.

ASCIRC