Computing loop invariants

Contact: Keshav Pingali

In some programs, transformations can be applied only if certain properties hold between program variables. A relatively simple example is LU with pivoting, shown below. We know how to perform automatic blocking of this code [Menon et al], but the correctness of this transformation depends on knowing that p(k) >= k at the point shown in the code.

```c
do k = 1, N
  // Pick the pivot row
  p(k) = k
  do i = k+1, N
    if abs(A(i,k)) > abs(A(p(k),k))
      p(k) = i
  // Swap rows
  // assert p(k) >= k
  do j = 1, N
    tmp = A(k,j)
    A(k,j) = A(p(k),j)
    A(p(k),j) = tmp
  // Scale current column
  do i = k+1, N
    A(i,k) = A(i,k) / A(k,k)
  // Update from current column
  // to columns to right
  do j = k+1, N
    do i = k+1, N
      A(i,j) = A(i,j) - A(i,k)*A(k,j)
```

In this example, it is easy to eye-ball the code to determine this. There are exactly two assignments to p in the body of the k loop: the assignment p(k) = k, and the assignment p(k) = i within the i loop. Since the bounds of the i loop are [k+1, N], it is clear that p(k) >= k at the assertion point in the code.

In this project, we will consider the problem of determining all loop invariants that hold in a program. There is a lot of interest in this problem in the POPL community. For example, POPL 2004 has two papers on this subject, one on non-linear loop invariant generation [Sankarnarayan et al], and one on inter-procedural linear loop invariant generation [Muller-Olim et al]. Other papers on this subject can be found in the references in these papers.
This project has two parts. In the first part, you must do a literature search to understand what is known about this subject. In the second part, you must implement in BCCK an algorithm for determining such loop invariants that is at least powerful enough to determine the information required to transform LU with pivoting as discussed above.

References:

