Maple

ProgramAnalysis

The ProgramAnalysis package is a new subpackage of the CodeTools package. There are commands to analyze the data dependencies in for loops and to verify the correctness of while loops.

```
with(CodeTools[ProgramAnalysis])
```

```
[CreateLoop, DependenceCone, GenerateProcedure, IsLoopInvariant, IterationSpace, LoopInvariant,
    TrajectoryPoints, UnimodularTransformation, VerifyLoop]
```

For loops

For loops can be analyzed to determine their data dependencies and apply transformations to assist in their parallelization. The commands support array indices that are polynomials with rational coefficients. Here is an example of how a nested for loop can be transformed so that the inner loop can be parallelized:

The following procedure has dependencies across both the i and the j loop indices and cannot be easily parallelized in this form:

```
p := proc(A)
  local i, j :
  for i from 1 to 3 do
    for j from 1 to 3 do
      A[i, j] := A[i - 1, j] + A[i, j - 1] :
    end do:
  end do:
end proc:
```

```
loop := CreateLoop(p) :
```

The dependencies in this loop cross both the i and j dimension, as can be seen in the equation of its dependence cone:

```
DependenceCone(loop, A, [di, dj]);
```

$$[0 \le dj, dj \le 1, di = 1 - dj]$$

Applying a transformation to the loop will change the relationship between the data dependencies:

```
transformation_matrix := ⟨⟨1 | 1⟩, ⟨0|1⟩⟩;
```

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

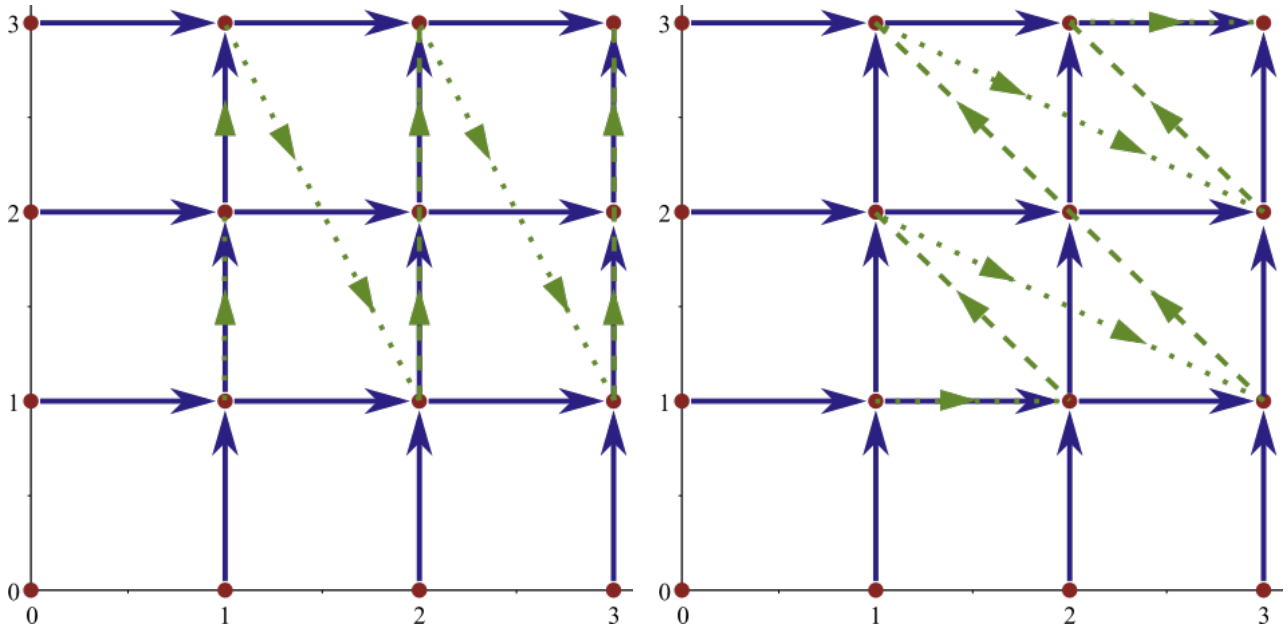
```
transformed_loop := UnimodularTransformation(loop, transformation_matrix) :
GenerateProcedure(transformed_loop);
proc(A)
    local i, j;
    for i from 2 to 6 do
        for j from max(1, i − 3) to min(3, i − 1) do
            A[i − j, j] := A[−1 + i − j, j] + A[i − j, j − 1]
        end do
    end do;
    return
end proc
```

This transformed loop no longer has a relationship between the dimensions its distance vectors:

```
DependenceCone(transformed_loop, A, [dn, dm]);
```

$$[dn = 1, 0 \le dm, dm \le 1]$$

The data dependencies in the loop body only depend on previously computed elements. The pattern for updating the array in the original and transformed loops are shown below. The inner loop of the transformed program corresponds to the anti-diagonal lines of the array updates. These updated operations can be performed simultaneously since they only refer to previously updated array entries (those entries that point toward a particular array entry).



Original loop's array updates Transformed loop's array updates

Legend

Red dots: Data points in array

Blue arrows: Read/write dependency between array entries

Green dotted lines: Order in which elements of the array are updated

The set of array indices for all values of the loop's index variables can also be computed:

$ispace := IterationSpace(loop);$

$$[1 \le i, i \le 3, 1 \le j, j \le 3]$$

$isolve(convert(ispace, set));$

$\{i=1, j=1\}, \{i=1, j=2\}, \{i=1, j=3\}, \{i=2, j=1\}, \{i=2, j=2\}, \{i=2, j=3\}, \{i=3, j=1\}, \{i=3, j=2\}, \{i=3, j=3\}$

While Loops

The while loop related commands can be used to formally verify that a procedure meets its specification. The invariants of a while loop can be computed and, when combined with the pre-condition and guard condition of the loop, used to verify whether or not the post-condition will be satisfied. This verification indicates whether or not the *ASSERT* statement after the loop will always be true. Loops whose assignments are polynomials with rational coefficients are supported.

The following example shows how the invariants of a while loop can be computed and

used to verify that the following program is free from errors and meets its specification encoded in the last *ASSERT* statement of the procedure:

```
z3sqrt := proc (a, err)
  local r, q, p;
  r := a - 1;
  q := 1;
  p := 1/2;
  ASSERT(a ≤ 4 and 1 ≤ a and 0 < err and p > 0 and r ≥ 0 );
  while err ≤ 2 * p * r do
    if 0 ≤ 2 * r - 2 * q - p then
      r := 2 * r - 2 * q - p;
      q := q + p;
      p := 1/2 * p :
    else
      r := 2 * r;
      p := 1/2 * p :
    end if:
  end do;
  ASSERT(q^2 ≤ a and a < q^2 + err);
  return q :
end proc:
```

Create the WhileLoop data structure:

$loop := CreateLoop(z3sqrt) :$

In this case, a loop invariant needs to be computed to formally verify the loop:

$invariant := LoopInvariant(loop, invarianttype = absolute);$

$$2\,p\,r + q^2 - a = 0$$

The return value of *true* indicates that the procedure is guaranteed to meet its specification:

$VerifyLoop(loop, invariant);$

$$true$$