多重ループ

```
void matmul(int n, myuint A[], myuint B[], myuint AB[])
{
  int i;
  for(i=0; i<n; i++)
  {
    int j;
    for(j=0; j<n; j++)
    {
      myuint s=0;
      int k;
      for(k=0; k<n; k++)
      {
        s = s + A[i*n+k]*B[k*n+j];
      }
      AB[i*n+j] = s;
    }
  }
}
```

これは，Reversible Languages and Incremental State Saving in Optimistic Parallel Discrete Event SimulationのListing 1.3.をC言語に直したコードである．このコードのi, j, kの値は各ループ終了時にnとなる．nは引数であるため，nの値は関数の外部で保存される．さらに，nは全く変更されないため，関数本体でnの値を保存しなくても，逆実行時にnの値は関数外部で復元される．したがって，nの値を保存する必要はない．また，i, j, kは通常の変換ではそれぞれの変数のスコープを抜ける際に保存されなければならないが，変数のスコープを抜ける際の値はすべてnの値であり，nの値は外部で保存されることにより，i, j, kの値は関数外部で保存されることになるため，i, j, kの値を保存する必要はない．したがって，次のような可逆化が提案できる．

	元のコード	順方向	逆方向

旧	void f(int x)
	{
	  s
	}
		void f(int x)
		{
		  s
		  SAVE(x);
		}
			void rev_f(int x)
			{
			  RESTORE(x);
			  rs
			}

新	void f(int x)
	{
	  s
	}
		void f(int x)
		{
		  s
		}
		（sでxに対する副作用がない）
			void rev_f(int x)
			{
			  rs
			}

	元のコード	順方向	逆方向

	void f(int n)
	{
	  int i;
	  i=0;
	  while(i<n)
	  {
	  s
	  i++;
	  }
	}
		void f(int n)
		{
		  int i;
		  int c=0;
		  i=0;
		  while(i<n)
		  {
		  c++;
		  s
		  i++;
		  }
		  SAVE(n);
		  SAVE(i);
		  SAVE(c);
		}
			void f(int n)
			{
			  int i;
			  int c;
			  RESTORE(c);
			  RESTORE(i);
			  RESTORE(n);
			  while(c)
			  {
			  --i;
			  rs
			  --c;
			  }
			}

	void f(int n)
	{
	  int i;
	  i=0;
	  while(i<n)
	  {
	  s
	  i++;
	  }
	}
		void f(int n)
		{
		  int i;
		  i=0;
		  while(i<n)
		  {
		  s
		  i++;
		  }
		}
		（sでnとiに対する副作用がない）
			void f(int n)
			{
			  int i;
			  i=n;
			  while(i)
			  {
			  --i;
			  rs
			  }
			}

上記は，引数の値が関数本体で変わらない場合の変換と，引数の値が関数本体で変わらず，ループの繰り返し回数に引数の値が使用されている場合の変換である．元の変換と比べると，関数の引数を保存せずに済み，追加の繰り返し回数の変数を保存せずに済み，元の繰り返し回数も保存せずに済む．これを先ほどのmatmul()に適用すると次のようになる．

標準化	順方向	逆方向

void matmul(int n, myuint A[],
myuint B[], myuint AB[])
{
  int i;
  i=0;
  while(i<n)
  {
  int j;
  j=0;
  for(j<n)
  {
  myuint s;
  s=0;
  int k;
	void matmul(int n, myuint A[],
	myuint B[], myuint AB[])
	{
	  int i;
	  i=0;
	  while(i<n)
	  {
	  int j;
	  j=0;
	  while(j<n)
	  {
	  myuint s;
	  s=0;
	  int k;
		void matmul(int n, myuint A[],
		myuint B[], myuint AB[])
		{
		  int i;
		  i=n;
		  while(i)
		  {
		  int j;
		  j=n;
		  --i;
		  while(j)
		  {
		  myuint s;
		  int k;

```
    k=0;
    for(k<n)
    {
      s = s + A[i*n+k]*B[k*n+j];
      k++;
    }
    AB[i*n+j] = s;
    j++;
   }
   i++;
  }
}
```

```
    k=0;
    while(k<n)
    {
      SAVE(s);
      s = s + A[i*n+k]*B[k*n+j];
      k++;
    }
    SAVE(AB[i*n+j]);
    AB[i*n+j] = s;
    j++;
   }
   i++;
  }
}
```

```
    k=n;
    --j;
    while(k)
    {
      --k;
      RESTORE(s);
    }
    RESTORE(AB[i*n+j]);
   }
  }
}
```

これで各ループ終了後の繰り返し回数の保存と，関数を抜ける際の保存がなくなった．以前の変換では，繰り返し回数が保存されるループに対して，2つの変数の保存が必要だった．最も内側のループはn^3回の繰り返し，1つ外側のループはn^2回の繰り返し，最も外側のループはn回の繰り返しであるため，2n^2+2n個の変数を保存しなくてもよくなった．

このコードでは最も内側のループにs = s + A[i*n+k]*B[k*n+j];があるため，SAVE()がn^3回呼び出されてしまう．しかし，s = s + A[i*n+k]*B[k*n+j];はs += A[i*n+k]*B[k*n+j];と同等の計算であり，変数sの型はmyuintである．myuintは8, 16, 32, 64bitの符号なし整数型のどれかなので，計算でオーバフローはしない．したがって，s += A[i*n+k]*B[k*n+j];にコードを変換すれば変数の値を保存する必要はない．ここで，次のような変換が提案できる．

元のコード	標準化
v = v + e	v += e （eでvが記述されていない）

この変換で，破壊的な代入である=演算を+=演算にすることにより，左辺の変数の型によってはSAVE()を呼び出す必要がなくなる．

Reverse C Compilerでは，配列の保存に対して議論されていなかった（配列をSAVE()すると配列の先頭のポインタが保存されるだけ）のでそれを拡張する必要がある．