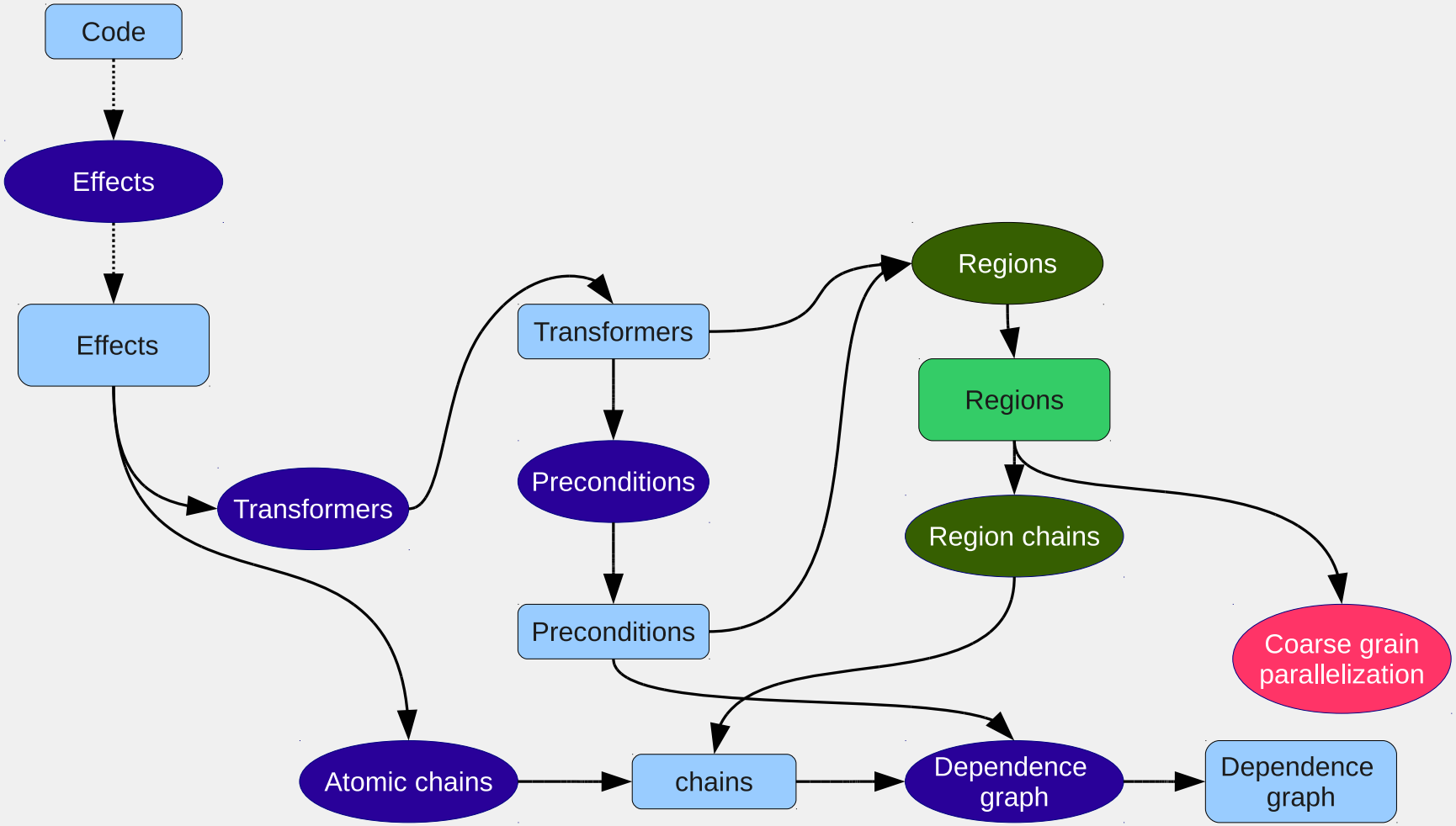
Integrating Pointer Analyses in PIPS Effect Analyses for an effective parallelization of C programs

Béatrice Creusillet

HPC Project

1st PIPS'days, october 2010, Paris

PIPS phases scheduling for Fortran programs



What are Effects?

- Memory effects of a program component
- Related to a memory store
- Proper and Cumulated
- May or Exact
- Read, Write, IN or OUT
- Simple or Convex (regions)

```
K = 1
C loop effects:
C <may be written>: A(*,*)
C <may be read>: A(*,*)>
do I = 1,N
C loop body effects:
C <may be written>: A(*,*)

C <must be written>: A(I,K)
   A(I,K) = B(K,I)
   K = K+1
C <may be read>: A(*,*)
C <must be read>: A(I,K)
   C(K,I) = A(I,K)
enddo
```

From Fortran...

- variable names always refer to the same memory location

... to C

$\Longrightarrow$ **Pointer Analysis**

From Fortran...

- variable names always refer to the same memory location

... to C

- a single variable name may refer to several memory locations

$\Longrightarrow$ **Pointer Analysis**

From Fortran...

- variable names always refer to the same memory location
- symbolic reference variable parts are restricted to array indices

... to C

- a single variable name may refer to several memory locations

$\Longrightarrow$ **Pointer Analysis**

From Fortran...

- variable names always refer to the same memory location
- symbolic reference variable parts are restricted to array indices
- an effect on `p(i,k)` is an effect on $\mathtt{p}[\sigma(i)][\sigma(k)]$

... to C

- a single variable name may refer to several memory locations

$\Longrightarrow$ **Pointer Analysis**

From Fortran...

- variable names always refer to the same memory location
- symbolic reference variable parts are restricted to array indices
- an effect on `p(i,k)` is an effect on $\mathtt{p}[\sigma(i)][\sigma(k)]$

... to C

- a single variable name may refer to several memory locations
- every element of a symbolic reference may be a variable part

$\Longrightarrow$ **Pointer Analysis**

From Fortran...

- variable names always refer to the same memory location
- symbolic reference variable parts are restricted to array indices
- an effect on `p(i,k)` is an effect on $\mathtt{p}[\sigma(i)][\sigma(k)]$

... to C

- a single variable name may refer to several memory locations
- every element of a symbolic reference may be a variable part
- an effect on `p[i][k]` is an effect on $\sigma(\sigma(\mathtt{p})[\sigma(i)])[\sigma(k)]$!

$\Longrightarrow$ **Pointer Analysis**

What Pointer Analysis?

- Pointer analyses are ... global ... hence costly/unprecise
- Precise enough
  - for dependence tests (may analyses)
  - for region analyses (exact analyses)
  - for targeted applications / program transformations
- Trade-off between time/memory consumption and precision
- Compatible with existing PIPS phases...

Landi and Ryder (20 years of PLDI, 2003)

We predict that the future will not see a best Pointer May-alias algorithm whose results are suitable for any application, but rather algorithms designed to optimize the tradeoffs to best meet the requirements of some particular application.

Constant Path Effects (François Irigoin's idea)

- Build effects with no dereferencements
- Fully compatible with existing phases ☺
- Orthogonal to the choice of a pointer analysis
- Choice criteria
  - precision
  - cost

Constant Path Effects: Example

```
int **p, **q, *a, *b;
a = (int *) malloc(10*sizeof(int)); // malloc_1
b = (int *) malloc(10*sizeof(int)); // malloc_2

p = &a;


p[0][2] = 0;

p = &b;
q=p;


p[0][2] = 0;


q[0][2] = 0;
```

Constant Path Effects: Example

```
int **p, **q, *a, *b;
a = (int *) malloc(10*sizeof(int)); // malloc_1
b = (int *) malloc(10*sizeof(int)); // malloc_2

p = &a;


p[0][2] = 0;

p = &b;
q=p;


p[0][2] = 0;


q[0][2] = 0;
```

Constant Path Effects: Example

```
int **p, **q, *a, *b;
a = (int *) malloc(10*sizeof(int)); // malloc_1
b = (int *) malloc(10*sizeof(int)); // malloc_2

p = &a;


p[0][2] = 0;

p = &b;
q=p;


p[0][2] = 0;


q[0][2] = 0;
```

Constant Path Effects: Example

```
int **p, **q, *a, *b;
a = (int *) malloc(10*sizeof(int)); // malloc_1
b = (int *) malloc(10*sizeof(int)); // malloc_2

p = &a;
//                  <must be read    >: a p
//                  <must be written>: malloc_1[2]
p[0][2] = 0;

p = &b;
q=p;
//                  <must be read    >: b p
//                  <must be written>: malloc_2[2]
p[0][2] = 0;
//                  <must be read    >: b q
//                  <must be written>: malloc_2[2]
q[0][2] = 0;
```

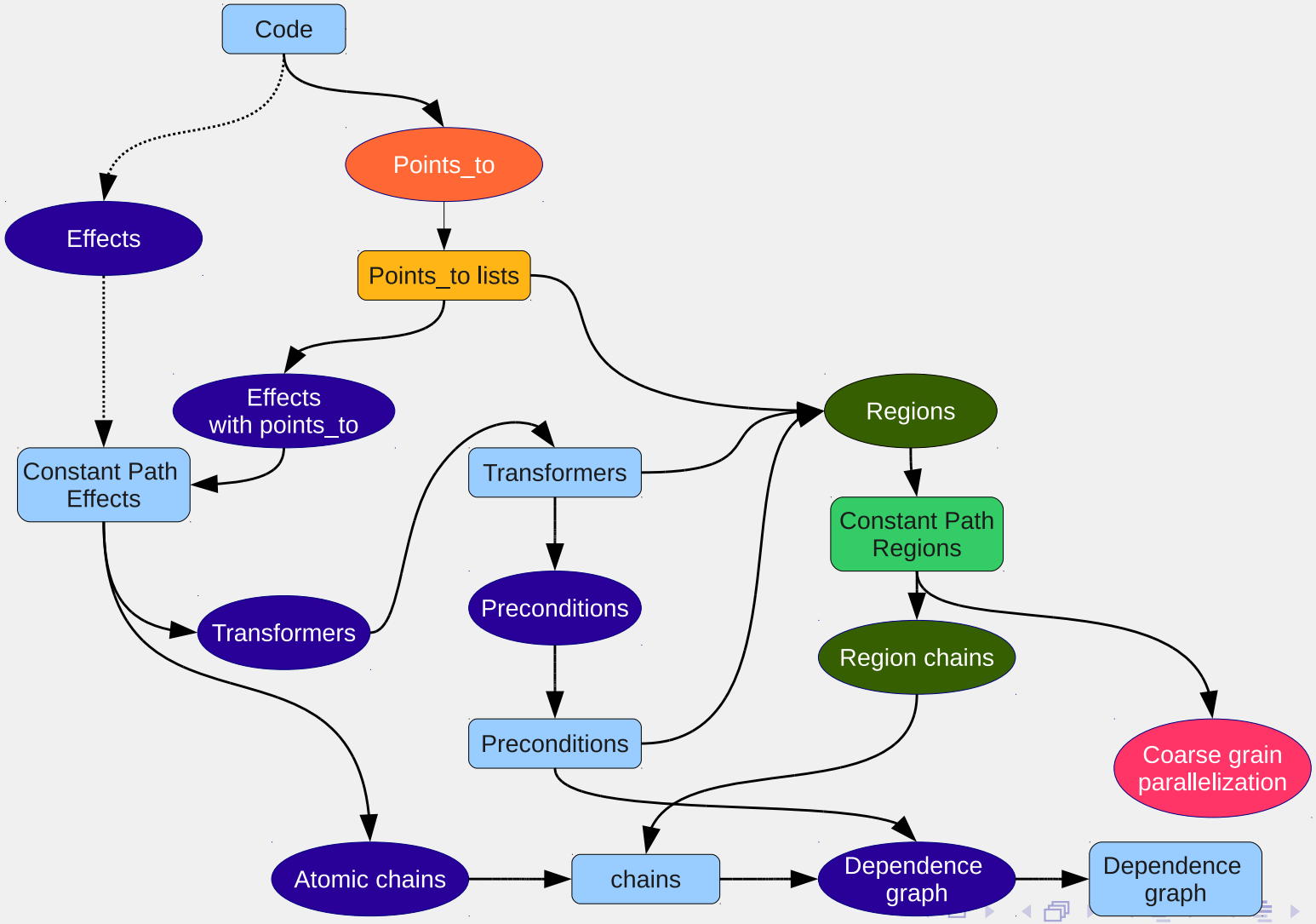
Constant Path Effects: Example

```
int **p, **q, *a, *b;
a = (int *) malloc(10*sizeof(int)); // malloc_1
b = (int *) malloc(10*sizeof(int)); // malloc_2

p = &a;
//                  <must be read    >: a p
//                  <must be written>: malloc_1[2]
p[0][2] = 0;

p = &b;
q=p;
//                  <must be read    >: b p
//                  <must be written>: malloc_2[2]
p[0][2] = 0;
//                  <must be read    >: b q
//                  <must be written>: malloc_2[2]
q[0][2] = 0;
```

PIPS phase scheduling with Constant Path Effects



Constant Path Effects may prevent parallelization

```
float *p;
p = foo(); // something that cannot be exactly represented
....
for(i=0; i<n; i++)

    p[i] = 0.0;
```

Constant Path Effects may prevent parallelization

```
float *p;
p = foo(); // something that cannot be exactly represented
....
for(i=0; i<n; i++)
// original effect:
//         <must be written>: p[i]


     p[i] = 0.0;
```

Constant Path Effects may prevent parallelization

```
float *p;
p = foo(); // something that cannot be exactly represented
....
for(i=0; i<n; i++)
// original effect:
//         <must be written>: p[i]
// points-to: {(p, *ANY_MODULE*:*ANYWHERE*, May)}


     p[i] = 0.0;
```

Constant Path Effects may prevent parallelization

```
float *p;
p = foo(); // something that cannot be exactly represented
....
for(i=0; i<n; i++)
// original effect:
//         <must be written>: p[i]
// points-to: {(p, *ANY_MODULE*:*ANYWHERE*, May)}
// constant path effect:
//         <may be written >: *ANY_MODULE*:*ANYWHERE*
     p[i] = 0.0;
```

Pointer Effects

- Preserve locally valid effects with dereferencements
- Replace modified pointers with their values
- Coarse Grain Parallelization: small adaptations
- But not compatible with other existing phases!
- What pointer analysis?

Constant Path Points-to may not be precise enough

```
float **p, **a;
...
if (...) p = &a[0]; else p = &a[1];


for(i=0; i<n; i++)




{
  float * q = p[i];


  for (j=0; i<n; j++)   q[j] = 0.0;
}
```

Constant Path Points-to may not be precise enough

```
float **p, **a;
...
if (...) p = &a[0]; else p = &a[1];
// points-to: {(p, a[0], may),(p, a[1], may)}

for(i=0; i<n; i++)




{
  float * q = p[i];



  for (j=0; i<n; j++)   q[j] = 0.0;
}
```

Constant Path Points-to may not be precise enough

```
float **p, **a;
...
if (...) p = &a[0]; else p = &a[1];
// points-to: {(p, a[0], may),(p, a[1], may)}

for(i=0; i<n; i++)




{
  float * q = p[i];

// pointer effect:
//         <must be written>: q[0:n-1]
  for (j=0; i<n; j++)   q[j] = 0.0;
}
```

Constant Path Points-to may not be precise enough

```
float **p, **a;
...
if (...) p = &a[0]; else p = &a[1];
// points-to: {(p, a[0], may),(p, a[1], may)}

for(i=0; i<n; i++)
```

```
{
  float * q = p[i];
// points-to: {(q, a[i], may),(q, a[i+1], may)}
// pointer effect:
//         <must be written>: q[0:n-1]
  for (j=0; i<n; j++)   q[j] = 0.0;
}
```

Constant Path Points-to may not be precise enough

```
float **p, **a;
...
if (...) p = &a[0]; else p = &a[1];
// points-to: {(p, a[0], may),(p, a[1], may)}

for(i=0; i<n; i++)
// loop body pointer effect:
//         <may be written >: a[i][0:n-1] a[i+1][0:n-1]


{
  float * q = p[i];
// points-to: {(q, a[i], may),(q, a[i+1], may)}
// pointer effect:
//         <must be written>: q[0:n-1]
  for (j=0; i<n; j++)   q[j] = 0.0;
}
```

Constant Path Points-to may not be precise enough

```
float **p, **a;
...
if (...) p = &a[0]; else p = &a[1];
// points-to: {(p, a[0], may),(p, a[1], may)}

for(i=0; i<n; i++)
// loop body pointer effect:
//         <may be written >: a[i][0:n-1] a[i+1][0:n-1]
// wished loop body pointer effect:
//         <must be written>: p[i][0:n-1]
{
  float * q = p[i];
// points-to: {(q, a[i], may),(q, a[i+1], may)}
// pointer effect:
//         <must be written>: q[0:n-1]
  for (j=0; i<n; j++)   q[j] = 0.0;
}
```

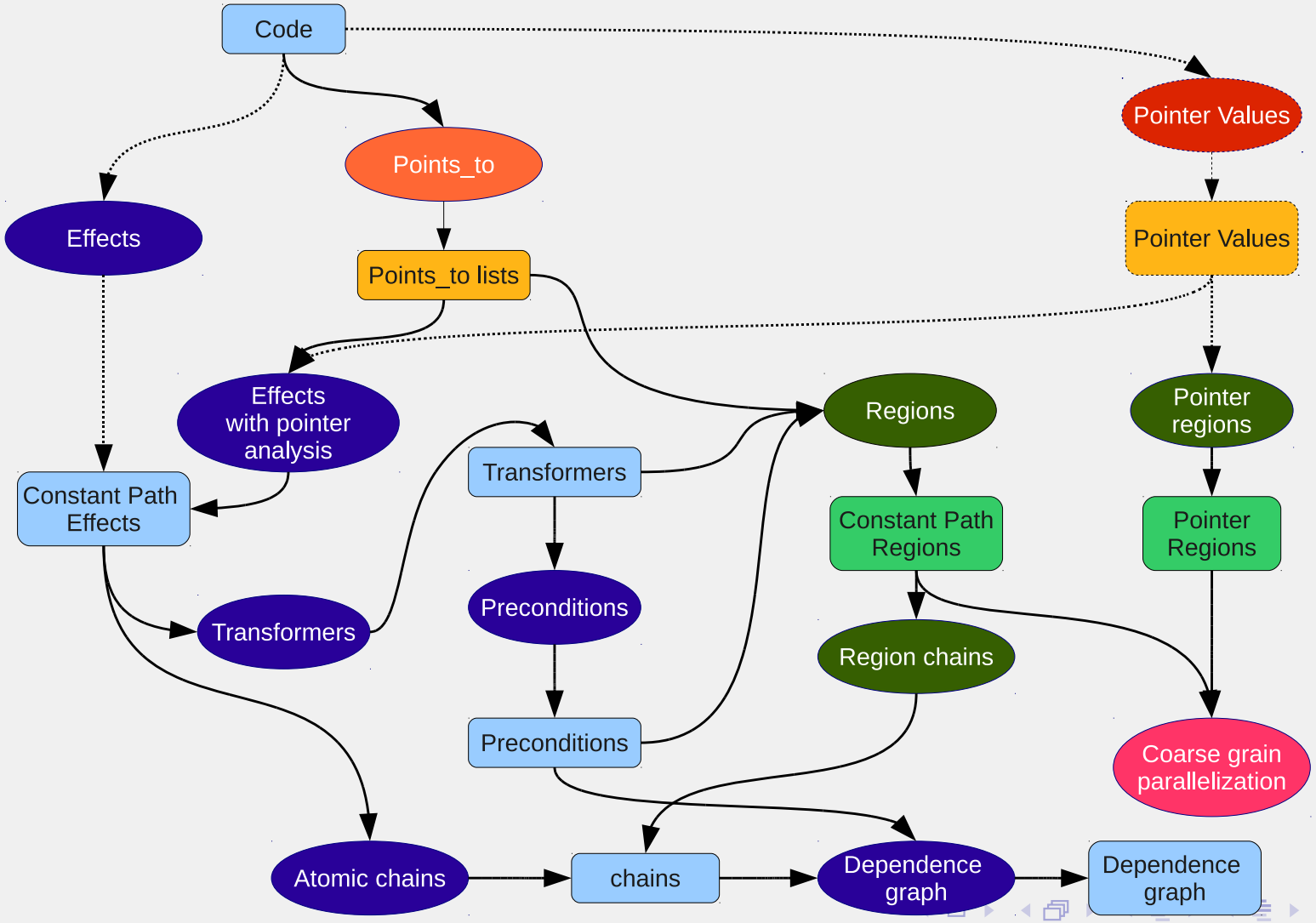
Pointer Values

- Keep pointer values relations given by programmer
  ex: `q = p[i]` $\Longrightarrow$ `p==q[i] (exact)`
- Preserve locally valid pointer values relations with dereferencements

$\Longrightarrow$ more precision ☺

- ☹ Client analyses must be adapted
- ☹ Maybe more costly than constant path points-to

PIPS phases scheduling with Pointer Values



Conclusion

- As many pointer analyses as C static analysers
- Keep an eye on actual needs/constraints!
- Pointer Values analysis under construction
- Missing features:
  - loops
  - intrinsics
  - interprocedural analysis
  - recursive types
- long term thinking needed on internal representation