Semantic analysis for arrays, structures and pointers

Nelson Lossing

Centre de recherche en informatique
Mines ParisTech

Septièmes rencontres de la communauté française de compilation

December 04, 2013

Framework PIPS

- Fortran code
- C code

```
int main() {
  int i=10, j=1;
  int k = 2*(2*i+j);

  return k;
}
```

- Static analyses
- Instrumentation/ Dynamic analyses
- Transformations
- Source code generation
- Code modelling
- Prettyprint

- Fortran code
- C code

```
//PRECONDITIONS
int main() {
// P() {}
   int i = 10, j = 1;

// P(i,j) {i==10, j==1}
   int k = 2*(2*i+j);

// P(i,j,k) {i==10,
      j==1, k==42}
   return k;
}
```

Motivation

- Analyze C applications
  - Signal Processing
  - Scientific Computing
  - Programs with pointers
- Use points-to graph to compute transformer
- Make a relational analysis for pointers
- Make better code optimization

Related work

- *Which pointer analysis should I use?*, Hind (2000)
- *Pointer analysis: haven't we solved this problem yet?*, Hind (2001)
- *Field-sensitive Value Analysis of Embedded C Programs with Union Types and Pointer Arithmetics*, Miné (2006)
- *Static analysis by abstract interpretation of concurrent programs*, Miné (2013)
- *Analyse des pointeurs pour le langage C*, Mensi (2013)

Outline

Static analyses (1)

- Memory Effect $\sim$ *Gen* and *Kill* sets of the Dragon Book

$E \in Statement \rightarrow (id \times \{READ, WRITE\} \times \{MAY, EXACT\})^*$

$E(x = x + 1) = \{(x, R, E), (x, W, E)\}$

Static analyses (1)

- Memory Effect $\sim$ *Gen* and *Kill* sets of the Dragon Book

$$E \in Statement \rightarrow (id \times \{READ, WRITE\} \times \{MAY, EXACT\})^*$$
$$E(x = x + 1) = \{(x, R, E), (x, W, E)\}$$

- Transformer $\sim$ approximation of transfer function

$$T \in Statement \rightarrow (id)^* \times (affine\,(in)equality)^*$$
$$T(x = x + 1) = (\{x\}, \{x = x\sharp init + 1\})$$

Static analyses (2)

- Points-to graph = relation from pointers to variables

$$PT = (id \times id \times \{MAY, EXACT\})^*$$
$$p = \&i$$
$$PT = \{p \rightarrow i, EXACT\}$$

Static analyses (2)

- Points-to graph = relation from pointers to variables

$$PT = (id \times id \times \{MAY, EXACT\})^*$$
$$p = \&i$$
$$PT = \{p \rightarrow i, EXACT\}$$

- Constant Path = extension of traditional identifiers

$$CP = (Name \times V_{ref} \times Type)$$
$$a[3][2]$$
$$(a, \{3, 2\}, integer)$$

New analyses

2 new independent analyses :

1. generalized transformers using points-to graph

Memory effect	Effect with points-to	Effect with points-to + points-to	Effect with points-to + points-to + constant path
$lhs = expr$ $\downarrow$ $anywhere = anything$	$lhs = expr$ $\downarrow$ Effect with $\mathcal{PT}^\sharp$ $id = anything$	$lhs = expr$ $\downarrow \mathcal{PT}^\sharp$ $id = expr$	$lhs = expr$ $\downarrow \mathcal{PT}^\sharp + \mathcal{CP}^\sharp$ $\mathcal{CP}^\sharp = expr$

New analyses

2 new independent analyses :

1. generalized transformers using points-to graph

Memory effect	Effect with points-to	Effect with points-to + points-to	Effect with points-to + points-to + constant path
$lhs = expr$ $\downarrow$ $anywhere = anything$	$lhs = expr$ $\downarrow$ Effect with $\mathcal{PT}^\sharp$ $id = anything$	$lhs = expr$ $\downarrow \mathcal{PT}^\sharp$ $id = expr$	$lhs = expr$ $\downarrow \mathcal{PT}^\sharp + \mathcal{CP}^\sharp$ $\mathcal{CP}^\sharp = expr$

2. Pointer arithmetic
   $Value = \{\text{integer, float, boolean, pointer}\}$

Simple case

```
int main() {
  int i=1, j=1;
  int *p;

  p=&i;
  *p=0;
  return *p;
}
```

Analyzing without points-to information

```
// PROPER EFFECTS
int main() {
//  <    is written>: i j
   int i = 1, j = 1;
   int *p;

//  <    is written>: p
   p = &i;
//  <may be written>:
      *ANY_MODULE*:*ANYWHERE*
//  <    is read   >: p
   *p = 0;

//  <may be read  >:
      *ANY_MODULE*:*ANYWHERE*
   return *p;
}
```

Analyzing without points-to information

```
// PROPER EFFECTS
int main() {
// <    is written>: i j
   int i = 1, j = 1;
   int *p;

// <    is written>: p
   p = &i;
// <may be written>:
     *ANY_MODULE*:*ANYWHERE*
// <    is read   >: p
   *p = 0;

// <may be read  >:
     *ANY_MODULE*:*ANYWHERE*
   return *p;
}
```

```
// TRANSFORMERS
// T(main) {}
int main() {
// T(i,j) {i==1, j==1}
   int i = 1, j = 1;
// T() {i==1, j==1}
   int *p;

// T() {i==1, j==1}
   p = &i;

// T(i,j) {i#init==1, j#init==1}
   *p = 0;

// T(main) {}
   return *p;
}
```

Analyzing with indirect points-to information

```
// PROPER EFFECTS with points-to
int main() {
//  <    is written>: i j
   int i = 1, j = 1;
   int *p;

//  <    is written>: p
   p = &i;
//  <    is read   >: p
//  <    is written>: i
   *p = 0;

//  <    is read   >: i p
   return *p;
}
```

Analyzing with indirect points-to information

```
// PROPER EFFECTS with points-to
int main() {
//  <    is written>: i j
   int i = 1, j = 1;
   int *p;

//  <    is written>: p
   p = &i;
//  <    is read   >: p
//  <    is written>: i
   *p = 0;

//  <    is read   >: i p
   return *p;
}
```

```
// TRANSFORMERS with
      PROPER EFFECTS with points-to
//  T(main) {}
int main() {
//  T(i,j) {i==1, j==1}
   int i = 1, j = 1;
//  T() {i==1, j==1}
   int *p;

//  T() {i==1, j==1}
   p = &i;
//  T(i) {i#init==1, j==1}
   *p = 0;

//  T(main) {j==1}
   return *p;
}
```

Analyzing with direct points-to information

```
// POINTS-TO
int main() {
// Points To: none
   int i = 1, j = 1;
// Points To: none
   int *p;

// p->undefined, EXACT
   p = &i;
// p->i, EXACT
   *p = 0;

// p->i, EXACT
   return *p;
}
```

Analyzing with direct points-to information

```
// POINTS-TO
int main() {
// Points To: none
   int i = 1, j = 1;
// Points To: none
   int *p;

// p->undefined, EXACT
   p = &i;
// p->i, EXACT
   *p = 0;

// p->i, EXACT
   return *p;
}
```

```
// TRANSFORMERS with points-to
//  T(main) {main==0}
int main() {
//  T(i,j) {i==1, j==1}
   int i = 1, j = 1;
//  T() {i==1, j==1}
   int *p;

//  T() {i==1, j==1}
   p = &i;
//  T(i) {i==0, i#init==1, j==1}
   *p = 0;

//  T(main) {i==0, j==1, main==0}
   return *p;
}
```

Analyzing translation of a sparse matrix

Representation of the sparse matrix

0	1	2	...
$index_0$; re_0 ; im_0	$index_1$; re_1 ; im_1	$index_2$; re_2 ; im_2	...

	c0	c1	c2	c3	c4	c5	...
0							
1							
2							

⋮

Analyzing translation of a sparse matrix

Representation of the sparse matrix

0	1	2	...
$index_0$; re_0 ; im_0	$index_1$; re_1 ; im_1	$index_2$; re_2 ; im_2	...

	c0=$index_0$	c1	c2	c3	c4	c5	...
0	re_0 ; im_0						
1							
2							

⋮

Analyzing translation of a sparse matrix

Representation of the sparse matrix

0	1	2	...
$index_0$; re_0 ; im_0	$index_1$; re_1 ; im_1	$index_2$; re_2 ; im_2	... ...

	c0	c1	c2=$index_1$	c3	c4	c5	...
0	re_0 ; im_0						
1			re_1 ; im_1				
2							

⋮

Analyzing translation of a sparse matrix

Representation of the sparse matrix

0	1	2	...
$index_0$; re_0 ; im_0	$index_1$; re_1 ; im_1	$index_2$; re_2 ; im_2	...

	c0	c1	c2	c3	c4=$index_2$	c5	...
0	re_0 ; im_0						
1			re_1 ; im_1				
2					re_2 ; im_2		

⋮

Code

```
#define DIM 3
#define DIM1 DIM
#define DIM2 2*DIM

typedef struct{
  int re;
  int im;
} COMPLEX;

typedef struct{
  int re;
  int im;
  int index;
} TCOMPLEX;
```

```
int main() {
  int i, k;
  TCOMPLEX sparse[DIM];
  COMPLEX matrix[DIM1][DIM2];
  TCOMPLEX *temp;
  temp = sparse;

loop1: for (i=0; i<DIM; i++) {
    temp[i].index = 2*i;
    temp[i].re = (i+1);
    temp[i].im = (i+1)*(i+1);
  }
loop2: for(i=0; i<DIM; i++) {
    k = sparse[i].index;
    if ((k>=0) && (k<DIM2)) {
      matrix[i][k].re = sparse[i].re;
      matrix[i][k].im = sparse[i].im;
    }
  }

  int result = -1;
  result = matrix[0][0].re + matrix[0][0].im
       + matrix[1][2].re + matrix[1][2].im +
       matrix[2][4].re + matrix[2][4].im;
  return result;
}
```

Code

```
#define DIM 3
#define DIM1 DIM
#define DIM2 2*DIM

typedef struct{
  int re;
  int im;
} COMPLEX;

typedef struct{
  int re;
  int im;
  int index;
} TCOMPLEX;
```

```
int main() {
  int i, k;
  TCOMPLEX sparse[DIM];
  COMPLEX matrix[DIM1][DIM2];
  TCOMPLEX *temp;
  temp = sparse;

loop1: for (i=0; i<DIM; i++) {
    temp[i].index = 2*i;
    temp[i].re = (i+1);
    temp[i].im = (i+1)*(i+1);
  }
loop2: for(i=0; i<DIM; i++) {
    k = sparse[i].index;
    if ((k>=0) && (k<DIM2)) {
      matrix[i][k].re = sparse[i].re;
      matrix[i][k].im = sparse[i].im;
    }
  }

  int result = -1;
  result = matrix[0][0].re + matrix[0][0].im
       + matrix[1][2].re + matrix[1][2].im +
       matrix[2][4].re + matrix[2][4].im;
  return result;
}
```

Code

```
#define DIM 3
#define DIM1 DIM
#define DIM2 2*DIM

typedef struct{
  int re;
  int im;
} COMPLEX;

typedef struct{
  int re;
  int im;
  int index;
} TCOMPLEX;
```

```
int main() {
  int i, k;
  TCOMPLEX sparse[DIM];
  COMPLEX matrix[DIM1][DIM2];
  TCOMPLEX *temp;
  temp = sparse;

loop1: for (i=0; i<DIM; i++) {
    temp[i].index = 2*i;
    temp[i].re = (i+1);
    temp[i].im = (i+1)*(i+1);
  }
loop2: for(i=0; i<DIM; i++) {
    k = sparse[i].index;
    if ((k>=0) && (k<DIM2)) {
      matrix[i][k].re = sparse[i].re;
      matrix[i][k].im = sparse[i].im;
    }
  }

  int result = -1;
  result = matrix[0][0].re + matrix[0][0].im
       + matrix[1][2].re + matrix[1][2].im +
       matrix[2][4].re + matrix[2][4].im;
  return result;
}
```

Code

```
#define DIM 3
#define DIM1 DIM
#define DIM2 2*DIM

typedef struct{
  int re;
  int im;
} COMPLEX;

typedef struct{
  int re;
  int im;
  int index;
} TCOMPLEX;
```

```
int main() {
  int i, k;
  TCOMPLEX sparse[DIM];
  COMPLEX matrix[DIM1][DIM2];
  TCOMPLEX *temp;
  temp = sparse;

loop1: for (i=0; i<DIM; i++) {
    temp[i].index = 2*i;
    temp[i].re = (i+1);
    temp[i].im = (i+1)*(i+1);
  }
loop2: for(i=0; i<DIM; i++) {
    k = sparse[i].index;
    if ((k>=0) && (k<DIM2)) {
      matrix[i][k].re = sparse[i].re;
      matrix[i][k].im = sparse[i].im;
    }
  }

  int result = -1;
  result = matrix[0][0].re + matrix[0][0].im
       + matrix[1][2].re + matrix[1][2].im +
       matrix[2][4].re + matrix[2][4].im;
  return result;
}
```

Unrolling code without points-to

```
int main() {
   int k;
   TCOMPLEX sparse[3];
   COMPLEX matrix[3][6];
   TCOMPLEX *temp;
   temp = sparse;

   temp[0].index = 0;
   temp[0].re = 1;
   temp[0].im = 1;
   temp[1].index = 2;
   temp[1].re = 2;
   temp[1].im = 4;
   temp[2].index = 4;
   temp[2].re = 3;
   temp[2].im = 9;

   k = sparse[0].index;
   if (k>=0&&k<6) {
      matrix[0][k].re = sparse[0].re;
      matrix[0][k].im = sparse[0].im;
   }
   k = sparse[1].index;
   if (k>=0&&k<6) {
      matrix[1][k].re = sparse[1].re;
      matrix[1][k].im = sparse[1].im;
   }
   k = sparse[2].index;
   if (k>=0&&k<6) {
      matrix[2][k].re = sparse[2].re;
      matrix[2][k].im = sparse[2].im;
   }

   int result = -1;
   result = matrix[0][0].re +
        matrix[0][0].im + matrix[1][2].re +
         matrix[1][2].im + matrix[2][4].re
        + matrix[2][4].im;

   return result;
}
```

Unrolling code without points-to

```
int main() {
   int k;
   TCOMPLEX sparse[3];
   COMPLEX matrix[3][6];
   TCOMPLEX *temp;
   temp = sparse;

   temp[0].index = 0;
   temp[0].re = 1;
   temp[0].im = 1;
   temp[1].index = 2;
   temp[1].re = 2;
   temp[1].im = 4;
   temp[2].index = 4;
   temp[2].re = 3;
   temp[2].im = 9;

   k = sparse[0].index;
   if (k>=0&&k<6) {
      matrix[0][k].re = sparse[0].re;
      matrix[0][k].im = sparse[0].im;
   }
   k = sparse[1].index;
   if (k>=0&&k<6) {
      matrix[1][k].re = sparse[1].re;
      matrix[1][k].im = sparse[1].im;
   }
   k = sparse[2].index;
   if (k>=0&&k<6) {
      matrix[2][k].re = sparse[2].re;
      matrix[2][k].im = sparse[2].im;
   }

   int result = -1;
   result = matrix[0][0].re +
        matrix[0][0].im + matrix[1][2].re +
         matrix[1][2].im + matrix[2][4].re
        + matrix[2][4].im;
//  T(main) {main==result}
   return result;
}
```

Unrolling code with points-to

```
int main() {
   int k;
   TCOMPLEX sparse[3];
   COMPLEX matrix[3][6];
   TCOMPLEX *temp;
   temp = sparse;

   temp[0].index = 0;
   temp[0].re = 1;
   temp[0].im = 1;
   temp[1].index = 2;
   temp[1].re = 2;
   temp[1].im = 4;
   temp[2].index = 4;
   temp[2].re = 3;
   temp[2].im = 9;

   k = sparse[0].index;
   matrix[0][0].re = sparse[0].re;
   matrix[0][0].im = sparse[0].im;
   k = sparse[1].index;
   matrix[1][2].re = sparse[1].re;
   matrix[1][2].im = sparse[1].im;
   k = sparse[2].index;
   matrix[2][4].re = sparse[2].re;
   matrix[2][4].im = sparse[2].im;

   int result = -1;
   result = matrix[0][0].re +
        matrix[0][0].im + matrix[1][2].re +
         matrix[1][2].im + matrix[2][4].re
        + matrix[2][4].im;

   return result;
}
```

Analyzing with points-to information and constant path

```
int main() {
   int k;
   TCOMPLEX sparse[3];
   COMPLEX matrix[3][6];
   TCOMPLEX *temp;
   temp = sparse;

   temp[0].index = 0;
   temp[0].re = 1;
   temp[0].im = 1;
   temp[1].index = 2;
   temp[1].re = 2;
   temp[1].im = 4;
   temp[2].index = 4;
   temp[2].re = 3;
   temp[2].im = 9;

   k = sparse[0].index;
   matrix[0][0].re = sparse[0].re;
   matrix[0][0].im = sparse[0].im;
   k = sparse[1].index;
   matrix[1][2].re = sparse[1].re;
   matrix[1][2].im = sparse[1].im;
   k = sparse[2].index;
   matrix[2][4].re = sparse[2].re;
   matrix[2][4].im = sparse[2].im;

   int result = -1;
   result = matrix[0][0].re +
        matrix[0][0].im + matrix[1][2].re +
         matrix[1][2].im + matrix[2][4].re
        + matrix[2][4].im;

   return result;
}
```

Analyzing with points-to information and constant path

```
int main() {
   int k;
   TCOMPLEX sparse[3];
   COMPLEX matrix[3][6];
   TCOMPLEX *temp;
   temp = sparse;

//  T(sparse[0][index], sparse[0][re],
     sparse[0][im])
    {sparse[0][index]==0,
     sparse[0][re]==1, sparse[0][im]==1}
   temp[0].index = 0;
   temp[0].re = 1;
   temp[0].im = 1;
   temp[1].index = 2;
   temp[1].re = 2;
   temp[1].im = 4;
   temp[2].index = 4;
   temp[2].re = 3;
   temp[2].im = 9;
   k = sparse[0].index;
   matrix[0][0].re = sparse[0].re;
   matrix[0][0].im = sparse[0].im;
   k = sparse[1].index;
   matrix[1][2].re = sparse[1].re;
   matrix[1][2].im = sparse[1].im;
   k = sparse[2].index;
   matrix[2][4].re = sparse[2].re;
   matrix[2][4].im = sparse[2].im;

   int result = -1;
   result = matrix[0][0].re +
        matrix[0][0].im + matrix[1][2].re +
         matrix[1][2].im + matrix[2][4].re
        + matrix[2][4].im;

   return result;
}
```

Analyzing with points-to information and constant path

```
int main() {
   int k;
   TCOMPLEX sparse[3];
   COMPLEX matrix[3][6];
   TCOMPLEX *temp;
   temp = sparse;

//  T(sparse[0][index], sparse[0][re],
     sparse[0][im])
    {sparse[0][index]==0,
     sparse[0][re]==1, sparse[0][im]==1}
   temp[0].index = 0;
   temp[0].re = 1;
   temp[0].im = 1;
   temp[1].index = 2;
   temp[1].re = 2;
   temp[1].im = 4;
   temp[2].index = 4;
   temp[2].re = 3;
   temp[2].im = 9;

//  T(k, matrix[0][0][re],
     matrix[0][0][im])
    {..., k==0, matrix[0][0][re]==1,
     matrix[0][0][im]==1}
   k = sparse[0].index;
   matrix[0][0].re = sparse[0].re;
   matrix[0][0].im = sparse[0].im;
   k = sparse[1].index;
   matrix[1][2].re = sparse[1].re;
   matrix[1][2].im = sparse[1].im;
   k = sparse[2].index;
   matrix[2][4].re = sparse[2].re;
   matrix[2][4].im = sparse[2].im;

   int result = -1;
   result = matrix[0][0].re +
        matrix[0][0].im + matrix[1][2].re +
         matrix[1][2].im + matrix[2][4].re
        + matrix[2][4].im;

   return result;
}
```

Analyzing with points-to information and constant path

```
int main() {
   int k;
   TCOMPLEX sparse[3];
   COMPLEX matrix[3][6];
   TCOMPLEX *temp;
   temp = sparse;

   temp[0].index = 0;
   temp[0].re = 1;
   temp[0].im = 1;
   temp[1].index = 2;
   temp[1].re = 2;
   temp[1].im = 4;
   temp[2].index = 4;
   temp[2].re = 3;
   temp[2].im = 9;

   k = sparse[0].index;
   matrix[0][0].re = sparse[0].re;
   matrix[0][0].im = sparse[0].im;
   k = sparse[1].index;
   matrix[1][2].re = sparse[1].re;
   matrix[1][2].im = sparse[1].im;
   k = sparse[2].index;
   matrix[2][4].re = sparse[2].re;
   matrix[2][4].im = sparse[2].im;

   int result = -1;
   result = matrix[0][0].re +
        matrix[0][0].im + matrix[1][2].re +
         matrix[1][2].im + matrix[2][4].re
        + matrix[2][4].im;
//  T(main) {..., main==20}
   return result;
}
```

Different possible analyses

1. Constrain pointer arithmetic only on same pointer types

```
int *p1, *q1;
float *p2, *q2;
p1 = q1 + 1;
p2 = q2 + 1;
```

2. Normalize pointer arithmetic, for instance with *sizeof*
3. Constrain pointer arithmetic only for arrays

Different possible analyses

1. Constrain pointer arithmetic only on same pointer types
2. Normalize pointer arithmetic, for instance with *sizeof*

```
int *p1, *q1;
p1 = q1 + 1;
// p1 = q1 + sizeof(int)
```

3. Constrain pointer arithmetic only for arrays

Different possible analyses

1. Constrain pointer arithmetic only on same pointer types
2. Normalize pointer arithmetic, for instance with *sizeof*
3. Constrain pointer arithmetic only for arrays

```
int *p, *q, a[10];
q = &a[0];
p = q + 1;
// q = &a[0], p=&a[1]
```

Example by normalization

```
int foo(int *p, int i) {
  int *r, *q;
  int error=-1, good=0;

  q = p+i;

  if(q==p && i>0)
    r = &error;
  else
    r = &good;

  return *r;
}
```

Analyzing with normalization

```
// T(foo) {}
int foo(int *p, int i) {
// T(q,r) {}
   int *r, *q;
// T(error,good) {error==-1, good==0}
   int error = -1, good = 0;
// T(q) {error==-1, good==0, 4i+p==q}
   q = p+i;

// T(r) {&good==r, error==-1, good==0, 4i+p==q}
   if (q==p&&i>0)
// T() {0==-1}
      r = &error;
   else
// T(r) {&good==r, error==-1, good==0, 4i+p==q}
      r = &good;

// T(foo) {&good==r, error==-1, good==0,
     4i+p==q}
   return *r;
}
```

Analyzing with points-to

```
// T(foo) {foo<=0, 0<=foo+1}
int foo(int *p, int i) {
// T(q,r) {}
   int *r, *q;
// T(error,good) {error==-1, good==0}
   int error = -1, good = 0;
// T(q) {error==-1, good==0, 4i+p==q}
   q = p+i;

// T(r) {&good==r, error==-1, good==0, 4i+p==q}
   if (q==p&&i>0)
// T() {0==-1}
      r = &error;
   else
// T(r) {&good==r, error==-1, good==0, 4i+p==q}
      r = &good;

// T(foo) {&good==r, error==-1, good==0,
     4i+p==q, foo<=0, 0<=foo+1}
   return *r;          (1)
}
```

```
// Points To (1):
// p -> _p_1[0] , EXACT
// q -> _p_1[*] , MAY
// r -> error , MAY
// r -> good , MAY
```

Analyzing after removing unreachable code

```
//  T(foo) {foo==0}
int foo(int *p, int i) {
//  T(q,r) {}
   int *r, *q;
//  T(error,good) {error==-1, good==0}
   int error = -1, good = 0;
//  T(q) {error==-1, good==0, 4i+p==q}
   q = p+i;
```

```
//  T(r) {&good==r, error==-1, good==0, 4i+p==q}
   r = &good;

//  T(foo) {&good==r, error==-1, good==0,
     4i+p==q, foo==0}
   return *r;          (1)
}
```

```
// Points To (1):
// p -> _p_1[0] , EXACT
// q -> _p_1[*] , MAY
// r -> good , EXACT
```

Contributions

- Extends PIPS to the pointer

Contributions

- Extends PIPS to the pointer
- Implementation of new analyses
  with a new phase and new properties
  - TRANSFORMERS_INTER_FULL_WITH_POINTS_TO
  - SEMANTICS_ANALYZE_CONSTANT_PATH
  - SEMANTICS_ANALYZE_SCALAR_POINTER_VARIABLES
  - POINTER_ARITHMETIC_WITH_SIZEOF
  - POINTER_ARITHMETIC_ONLY_FOR_ARRAY

Contributions

- Extends PIPS to the pointer
- Implementation of new analyses
  with a new phase and new properties
  - TRANSFORMERS_INTER_FULL_WITH_POINTS_TO
  - SEMANTICS_ANALYZE_CONSTANT_PATH
  - SEMANTICS_ANALYZE_SCALAR_POINTER_VARIABLES
  - POINTER_ARITHMETIC_WITH_SIZEOF
  - POINTER_ARITHMETIC_ONLY_FOR_ARRAY
- $\sim$1500 lines added and $\sim$500 modified
  in the $\sim$600k lines of PIPS
- Add more than 50 tests cases

Future work

- *cast* and *union*
- Update the points-to graph with preconditions
- Study pointer arithmetic constrained to arrays
- Improve formalisation

Semantic analysis for arrays, structures and pointers

Nelson Lossing

Centre de recherche en informatique
Mines ParisTech

Septièmes rencontres de la communauté française de compilation

December 04, 2013