Design Of A Flow Sensitive, Context Sensitive Points To Algorithm

Amira Mensi

Centre for Research In Computer
Department of Mathematics and Systems

October 25, 2010

Plan

Issue (1)

```
void foo()
{
  int **z,*v,*w,x=0,y=0,t=0;
  w = &t;
  *z = &y;
  z = &v;
```

```
  if(x>0) {
    z = &w;
  }else {
    *z = &x;
  }
}
```

Issue (2)

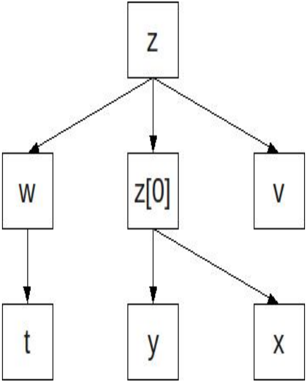


Figure: Memory state

Issue (3)

Many cases should be taken into account [Emami 93]

- p = &i;
- p = q;
- my_str->p = &i;
- my_str.p = &i;
- p = &tab[i];
- foo(x)->p = q;
- ...

Reduce complexity to 2 basic cases

- p = &i; //{(p,i,E)}
- p = q; //{(q,j,M),(p,j,M)}

pt=(source,sink,approximation)

Goals

- Design a flow-sensitive algorithm (statement's order is taken into account)
- Design a context-sensitive algorithm (store informations about the call site)
  [not adressed in this talk]
- Handle all C features

Statement Level

A statement can be a:

- sequence
- test
- loop
- whileloop
- goto
- forloop
- expression

Sequence Level

$$\mathcal{PT} : (i, \sigma) \mapsto (i', \sigma) \tag{1}$$

For $[\![S_1; S_2]\!]$:

$$\mathcal{PT}[\![S_1; S_2]\!](i, \sigma) = \mathcal{PT}[\![S_2]\!](\mathcal{PT}[\![S_1]\!](i, \sigma)) \tag{2}$$

Test Level (1)

Exact equation with condition evaluation's for $[\![if\ \ E\ \ then\ \ S_1\ \ else\ \ S_2]\!]$:

$$\mathcal{PT}[\![if\ \ E\ \ then\ \ S_1\ \ else\ \ S_2]\!](i,\sigma) = \mathcal{PT}[\![S_1]\!](\mathcal{PT}[\![E]\!](i,\sigma)) \wedge \mathcal{E}[\![E]\!]\sigma \\ \bigcup \mathcal{PT}[\![S_2]\!](\mathcal{PT}[\![\neg E]\!](i,\sigma)) \wedge \neg\mathcal{E}[\![E]\!]\sigma$$

Test Level (2)

Approximate equations without condition evaluation's for $[\![if\ E\ then\ S_1\ else\ S_2]\!]$:

$$\overline{\mathcal{PT}}[\![if\ E\ then\ S_1\ else\ S_2]\!](i,\sigma) = \mathcal{PT}[\![S_1]\!](\mathcal{PT}[\![E]\!](i,\sigma)) \bigcup \mathcal{PT}[\![S_2]\!](\mathcal{PT}[\![\neg E]\!](i,\sigma))$$

$$\underline{\mathcal{PT}}[\![if\ E\ then\ S_1\ else\ S_2]\!](i,\sigma) = \mathcal{PT}[\![S_1]\!](\mathcal{PT}[\![E]\!](i,\sigma)) \bigcap \mathcal{PT}[\![S_2]\!](\mathcal{PT}[\![\neg E]\!](i,\sigma))$$

with

$$\mathcal{PT}[\![\neg E]\!](i,\sigma) = \mathcal{PT}[\![E]\!](i,\sigma)$$

Loop Level (1)

The exact equation with condition evaluation's for $[\![while\ \ C\ \ do\ \ b]\!]$:

$$\begin{aligned}\mathcal{PT}[\![while\ \ C\ \ do\ \ b]\!](i,\sigma) &= \mathcal{PT}[\![if\ \ C\ \ then\ \ b; while\ \ C\ \ do\ \ b]\!](i,\sigma)\\ &= \mathcal{PT}[\![b; while\ \ C\ \ do\ \ b]\!](\mathcal{PT}[\![C]\!](i,\sigma)) \wedge \mathcal{E}[\![C]\!]\sigma\\ &\bigcup \mathcal{PT}[\![C]\!](i,\sigma) \wedge \neg\mathcal{E}[\![C]\!]\sigma\\ &= \mathcal{PT}[\![while\ \ C\ \ do\ \ b]\!](\mathcal{PT}[\![b]\!](\mathcal{PT}[\![C]\!](i,\sigma))) \wedge \mathcal{E}[\![C]\!]\sigma\\ &\bigcup \mathcal{PT}[\![C]\!](i,\sigma) \wedge \neg\mathcal{E}[\![C]\!]\sigma\end{aligned}$$

Loop Level (2)

The approximate equations for $[\![while\ \ C\ \ do\ \ b]\!]$, no condition evaluation's is taken into account:

$$\begin{aligned}\overline{\mathcal{PT}}[\![while\ \ C\ \ do\ \ b]\!](i,\sigma) &= \overline{\mathcal{PT}}[\![if\ \ C\ \ then\ \ b; while\ \ C\ \ do\ \ b]\!](i,\sigma)\\ &= \overline{\mathcal{PT}}[\![b; while\ \ C\ \ do\ \ b]\!](\overline{\mathcal{PT}}[\![C]\!](i,\sigma))\bigcup\overline{\mathcal{PT}}[\![C]\!](i,\sigma)\\ &= \overline{\mathcal{PT}}[\![while\ \ C\ \ do\ \ b]\!](\overline{\mathcal{PT}}[\![b]\!](\overline{\mathcal{PT}}[\![C]\!](i,\sigma)))\\ &\bigcup\overline{\mathcal{PT}}[\![C]\!](i,\sigma)\end{aligned}$$

$$\begin{aligned}\underline{\mathcal{PT}}[\![while\ \ C\ \ do\ \ b]\!](i,\sigma) &= \underline{\mathcal{PT}}[\![if\ \ C\ \ then\ \ b; while\ \ C\ \ do\ \ b]\!](i,\sigma)\\ &= \underline{\mathcal{PT}}[\![b; while\ \ C\ \ do\ \ b]\!](\underline{\mathcal{PT}}[\![C]\!](i,\sigma))\bigcap\underline{\mathcal{PT}}[\![C](i,\sigma)\\ &= \underline{\mathcal{PT}}[\![while\ \ C\ \ do\ \ b]\!](\underline{\mathcal{PT}}[\![b]\!](\underline{\mathcal{PT}}[\![C]\!](i,\sigma)))\\ &\bigcap\underline{\mathcal{PT}}[\![C]\!](i,\sigma)\end{aligned}$$

Loop Level (3)

- $\mathcal{PT}[\![while\ \ C\ \ do\ \ b]\!]$ replaced by $\mathcal{W}$;
- $\mathcal{PT}[\![b]\!]$ replaced by $\mathcal{B}$;
- $\mathcal{PT}[\![C]\!]$replaced by $\mathcal{C}$;

To obtain a recurrence equation

$$\mathcal{W}(i,\sigma) = (\mathcal{W} \circ \mathcal{B} \circ \mathcal{C})(i,\sigma) \cup \mathcal{C}(i,\sigma)$$

$$\mathcal{W}(i,\sigma) = ((\mathcal{W} \circ \mathcal{B} \circ \mathcal{C}) \cup \mathcal{C}) \circ \mathcal{B} \circ \mathcal{C})(i,\sigma) \cup \mathcal{C}(i,\sigma)$$

$$= \bigcup_{k=1}^{\infty} (\mathcal{B} \circ \mathcal{C})^{k-1} \circ \mathcal{C}$$

more precisely:

$$Y_0 = \mathcal{C}$$

$$Y_i = Y_{i-1} \cup \mathcal{B} \circ \mathcal{C} \circ Y_{i-1}$$

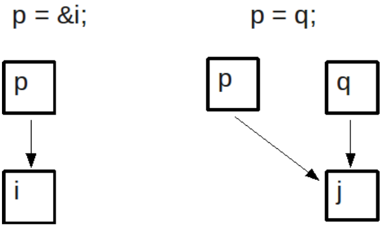
Expression Level

An expression could be a :

- reference
- range
- call
- cast
- sizeofexpression
- subscript
- application

- At the IR an assignment is a call so a sub case of expression
- The assignment operator generates *points to* relations
- For the other sub cases a recursive descent is performed

Assignment Algorithm (1)



Assignment Algorithm (2)

lhs = rhs; // in is the set of initial *points to*

1. Compute side effects of lhs and rhs
2. Extract may/must relations
3. Evaluate lhs using *points to* relations
4. switch rhs type's :
   - if rhs is an lhs => Evaluate rhs using *points to* relations
   - else change rhs into an lhs => Evaluate rhs using *points to* relations
   - for the ohter sub cases a recursive descent is performed
5. compute must/may kill set
6. compute must/may gen set
7. compute must/may out set
8. out = out_may $\sqcup$ out_must

Normalization of lhs with memory access paths

1. translate **p into p[0][0], my_str.p into my_str[.q], my_str→q en my_str[0][.q]
2. evaluate constant path using *points-to* p[0] replaced by i if (p,i,E)

The kill operator

- $constant - path : CP = Module \times Name \times Type \times Vref$
- $points - to : PT = CP \times CP \times A$
- $L = \{(m_l, n_l, t_l, vref_l)\}$

$$Kill_{MUST}(L, In) = \{(s, d, a) \in In \mid \forall l \in L \wedge opkillmust_{cp}(s, l) \wedge |L \models 1\}$$

$$Kill_{MAY}(L, In) = \{(s, d, a) \in In \mid \exists\ l \in L \wedge opkillmay_{cp}(s, l)\}$$

Examples (1): Program

```
typedef struct LinkedList{
  int *val;
  struct LinkedList *next;

}list;

list* initialize()
{
  int *pi;
```

```
  list *l=NULL, *nl;

  pi = malloc(sizeof(int));
  nl = malloc(sizeof(list*));
  nl->val = pi;

  return l;
}
```

Examples (1): Result

```
list * initialize()
{
// points to = {}
   int *pi;
// points to = {}
   list *l = (void *) 0, *nl;
// points to = {}

   pi = malloc(sizeof(int));
// {(pi,*HEAP*_l_15,-Exact-)}
```

```
   nl = malloc(sizeof(list *));
// {(nl,*HEAP*_l_16,-Exact-);(pi,*HEAP*
      _l_15,-Exact-)}
   nl->val = pi;
// {(*HEAP*_l_16[val],*HEAP*_l_15,-Exact-)
      ;(nl,*HEAP*_l_16,
//  -Exact-);(pi,*HEAP*_l_15,-Exact-)}

   return l;
}
```

- heap objects designated by *heap*NumberStatement
- dereferencing pointers evaluated using computed *points-to*

Examples (2): Program

```
typedef struct ListeChainee{
  int val;
  struct ListeChainee * next;
}liste;
int count(liste* p)
{
  liste* q = p;
```

```
  int i = 0;
  while(p != NULL){
    i++;
    p = p->next;
  }
  return i;
}
```

- while loop: traverse a linked list
- while loop changed into consecutive if (experimentally)
- dereferencing pointers evaluated using computed *points-to*

Examples (2): Result

```
typedef struct ListeChainee{
  int val;
  struct ListeChainee * next;
}liste;
// points-to {(p,p_formal,Exact)}
int count(liste* p)
{// points-to {(p,p_formal,Exact)}
  liste* q = p;
 // points-to {(p,p_formal,Exact);(q,
       p_formal,Exact)}
  int i = 0;
 // points-to {(p,p_formal,Exact);(q,
       p_formal,Exact)}
  if(p != NULL){
 // points-to {(p,p_formal,Exact);(q,
       p_formal,Exact)}
    i++;
 // points-to {(p,p_formal,Exact);(q,
       p_formal,Exact)}
    p = p->next;
 // points-to {(p,p_formal[next],Exact);(q,
       p_formal,Exact)}
 }
 if(p != NULL){
 // points-to {(p,p_formal[next],May);(q,
       p_formal,Exact)}
    i++;
 // points-to {(p,p_formal[next],May);(q,
       p_formal,Exact)}
    p = p->next;
 // points-to {(p,p_formal[next],May);(p,
       p_formal[next][next],May);(q,
       p_formal,Exact)}
 }

 return i;
}
```

Examples (2): Result

- sink increases over iterations
- appearence of a common prefix

Examples (3): Program

```
typedef struct ListeChainee{
  int* val;
  struct ListeChainee * next;
}liste;
int count(liste* p)
{
  liste* q = p;
  int i = 0, j;
  int tab[20];
  for(j=0; j<20; j++){
    tab[j] = j
  }
  while(p != NULL){
    p->val = &tab[i];
    p = p->next;
    i++;
  }
  return i;
}
```

Examples (3): Result

$$
\begin{aligned}
i = 0: \; Y_0 =& \{(q, p\_formal, E), (p, p\_formal, E)\} \\
i = 1: \; Y_1 =& \{(q, p\_formal, E), (p\_formal[val], tab[*], M), (p, p\_formal[next], M)\} \\
i = 2: \; Y_2 =& \{(q, p\_formal, E), (p\_formal[val], tab[*], M), (p, p\_formal[next], M)\} \\
& \cup \{(q, p\_formal, E), (p\_formal[next][val], tab[*], E), \\
& (p, p\_formal[next][next], E)\} \\
=& \{(q, p\_formal, E), (p\_formal[val], tab[*], M), (p, p\_formal[next], M); \\
& (p, p\_formal[next][next], M), (p\_formal[next][val], tab[*], M)\} \\
i = n: \; Y_n =& \{(q, p\_formal, E), (p\_formal[val], tab[*], M), (p, p\_formal[next]^n, M), \\
& (p\_formal[next]^n[val], tab[*], M), (p, p\_formal[next]^{n-1}, M), \\
& (p\_formal[next]^{n-1}[val], tab[*], M), (p, p\_formal[next]^{n-2}, M), ...\}
\end{aligned}
$$

Examples (3): Result

- sink increases over iterations
- source increases over iterations
- appearence of a common prefix
- appearence of a new prefix = common_prefix[val]

Examples (4): Program

```
typedef struct LinkedList{
  int *val;
  struct LinkedList *next;

}list;

list* initialize()
{
  int *pi, i;
  list *l=NULL, *nl;

  if(!feof(stdin)){
    scanf("%d",&i);
    pi = malloc(sizeof(int));
    *pi = i;
    nl = malloc(sizeof(list*));
    nl->val = pi;
    nl->next = l;
    l = nl;
  if(!feof(stdin)){
    scanf("%d",&i);
    pi = malloc(sizeof(int));
    *pi = i;
    nl = malloc(sizeof(list*));
    nl->val = pi;
    nl->next = l;
    l = nl;
   }
  }
  return l;
}
```

Examples (4): Result

```
if (!feof(stdin)) {
  pi = malloc(sizeof(int));
  //{(pi,*HEAP*_l_17,E)}
  nl = malloc(sizeof(list *));
  //{(nl,*HEAP*_l_19,E);(pi,*HEAP*_l_17,E)}
  nl->val = pi;
  //{(*HEAP*_l_19[val],*HEAP*_l_17,E);(nl,*
         HEAP*_l_19,E);(pi,*HEAP*_l_17,E)}
  nl->next = l;
  //{(*HEAP*_l_19[val],*HEAP*_l_17,E);(nl,*
         HEAP*_l_19,E);(pi,*HEAP*_l_17,E)}
  l = nl;
  //{(*HEAP*_l_19[val],*HEAP*_l_17,E);(l,*
         HEAP*_l_19,E);(nl,*HEAP*_l_19,E);
  //(pi,*HEAP*_l_17,E)}
  if (!feof(stdin)) {
    //{(*HEAP*_l_19[val],*HEAP*_l_17,M);(l,*
           HEAP*_l_19,M);(nl,*HEAP*_l_19,M);
    //(pi,*HEAP*_l_17,M)}
    pi = malloc(sizeof(int));
    //{(*HEAP*_l_19[val],*HEAP*_l_17,M);(l,*
           HEAP*_l_19,M);(nl,*HEAP*_l_19,M);
    //(pi,*HEAP*_l_25,E)}
    nl = malloc(sizeof(list *));
    //{(*HEAP*_l_19[val],*HEAP*_l_17,M);(l,*
           HEAP*_l_19,M);(nl,*HEAP*_l_27,E);
    //(pi,*HEAP*_l_25,E)}
    nl->val = pi;
    //{(*HEAP*_l_27[val],*HEAP*_l_25,E);
    //(l,*HEAP*_l_19,M);(nl,*HEAP*_l_27,E);(
           pi,*HEAP*_l_25,E)}
    nl->next = l;
    //{(*HEAP*_l_27[next],*HEAP*_l_19,E);
    //(*HEAP*_l_27[val],*HEAP*_l_25,E);(l,*
           HEAP*_l_19,M);(nl,*HEAP*_l_27,E);(
           pi,*HEAP*_l_25,E)}
    l = nl;
  }
}
```

Examples (4): Result

- source/sink increases over iterations
- more information about heap sites
- the number of *points to* relations increses
- need to find a fix point ?

1. k-limiting: define a length limit
2. use lattice order:

$$\{(q, pn^i, a)\} \cup in = \{(q, anywhere, a)\} \cup in \ \ si \ \ i >= k \quad (3)$$

Prospects

- handle more C features
- interprocedual analysis using *points to* summaries