Lecture 05 – Pointer Analyses & CSSV

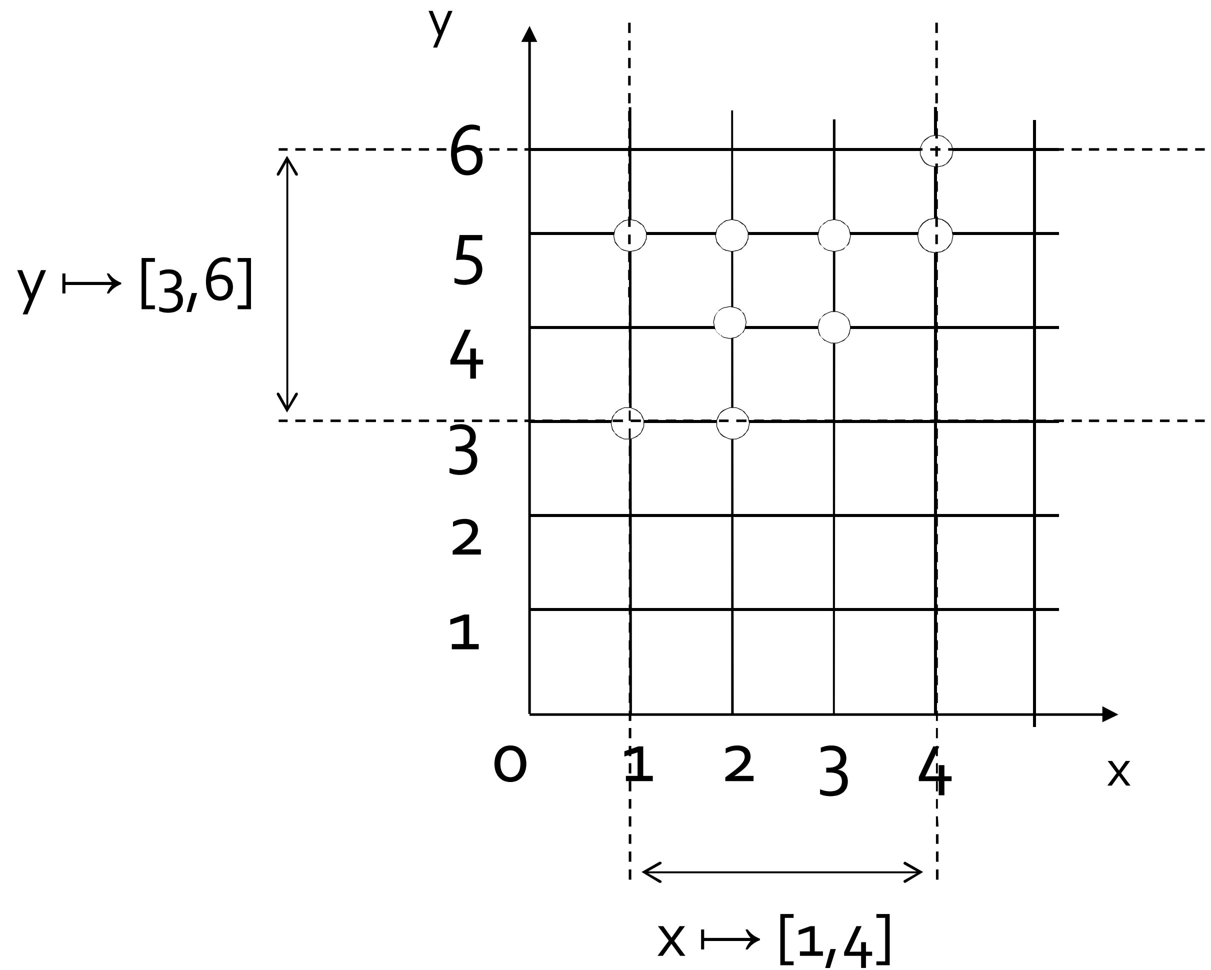
# PROGRAM ANALYSIS & SYNTHESIS

Eran Yahav

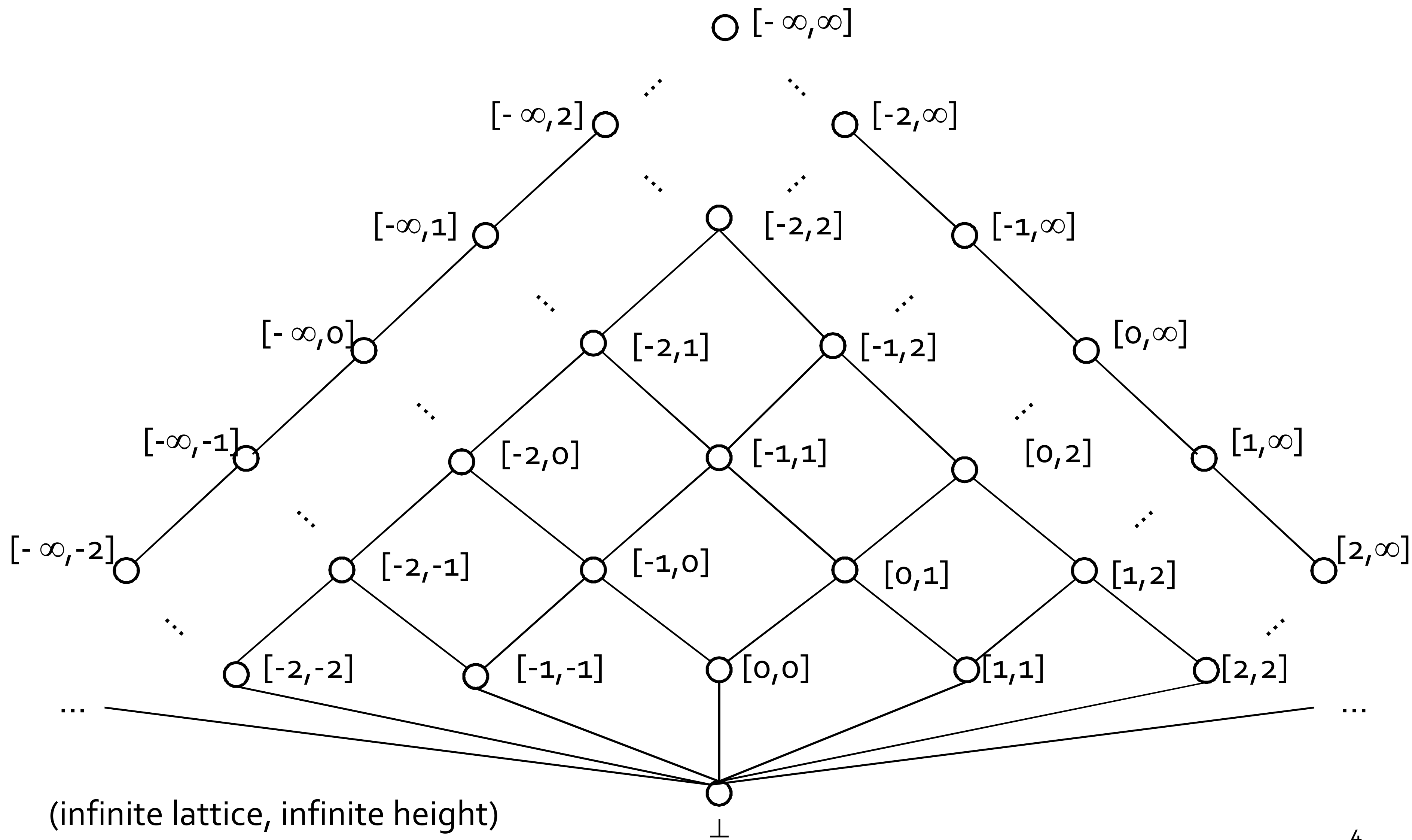
# Previously

- Numerical Abstractions
  - parity
  - signs
  - constant
  - interval
  - octagon
  - polyhedra

# Intervals Abstraction



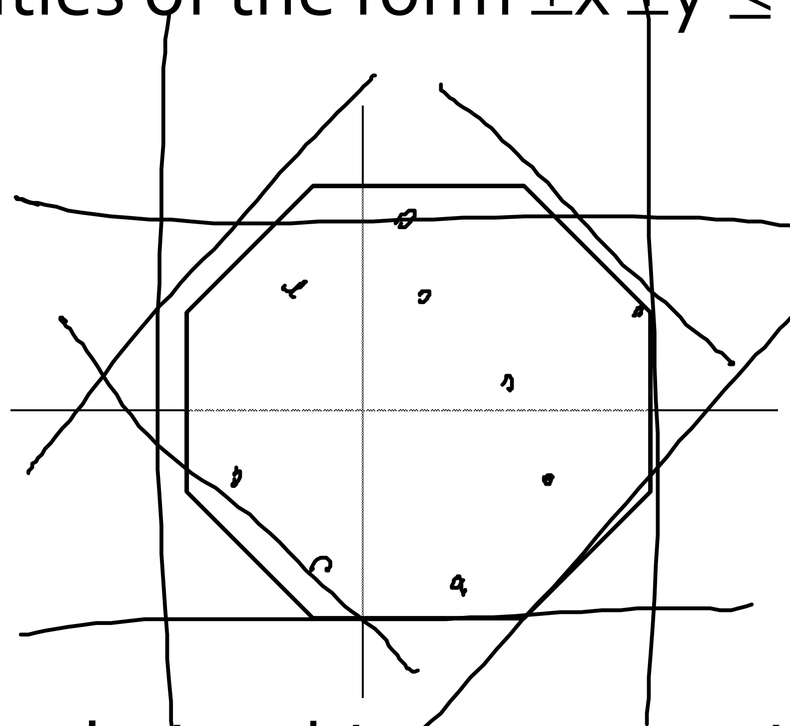
# Interval Lattice




(infinite lattice, infinite height)

# Octagon Abstraction

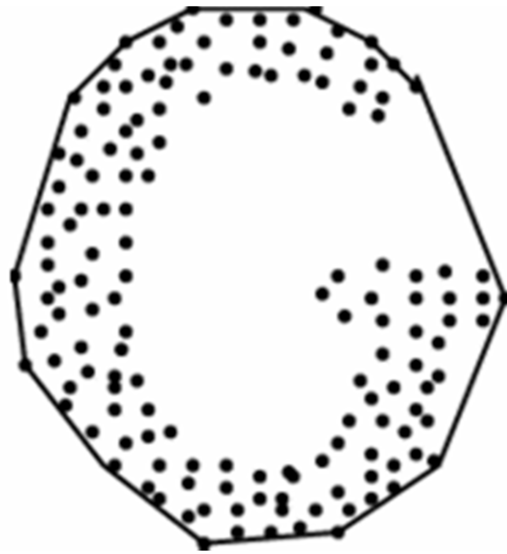
- abstract state is an intersection of linear inequalities of the form $\pm x \pm y \leq c$



- captures relationships common in programs (array access)

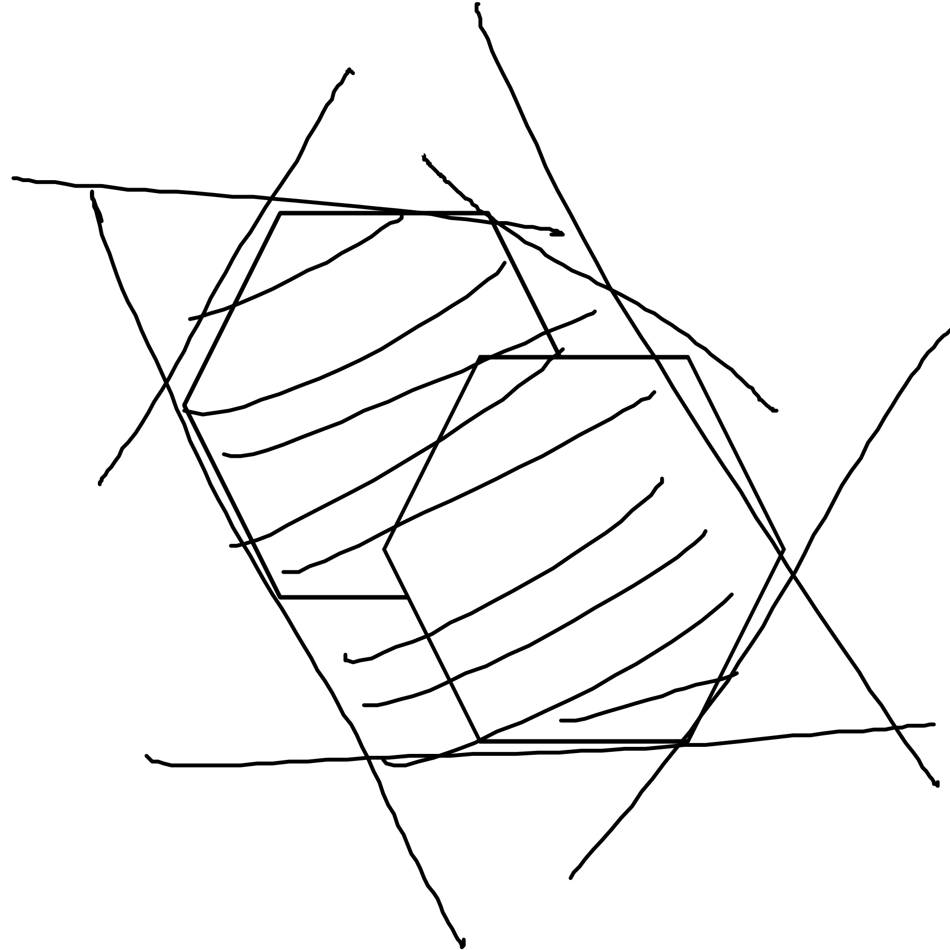
# Polyhedral Abstraction

- abstract state is an intersection of linear inequalities of the form $a_1x_2+a_2x_2+\ldots a_nx_n \leq c$
- represent a set of points by their convex hull



(image from http://www.cs.sunysb.edu/~algorith/files/convex-hull.shtml)

# Operations on Polyhedra



# Today

- Verifying absence of buffer overruns
- (requires) Heap abstractions
- (requires) Combining heap and numerical information

- Acknowledgement:
  - CSSV slides adapted from Nurit Dor (TAU, Panaya)

# Simple Motivating Example

```
void simple() {
   char        s[20];
   char        *p;
   char        t[10];

   strcpy(s,”Hello”);
   p = s + 5;
   strcpy(p,” world!”);
   strcpy(t,s);
}
```

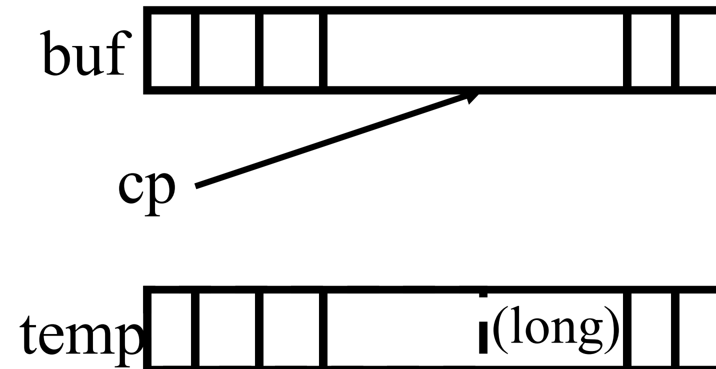
Unsafe call to strcpy

# String Manipulation Errors

- An important problem
  - Common errors
  - Cause security vulnerability
- A challenging problem
  - Use of pointers
  - Use of pointer arithmetic
  - Error point vs. failure point

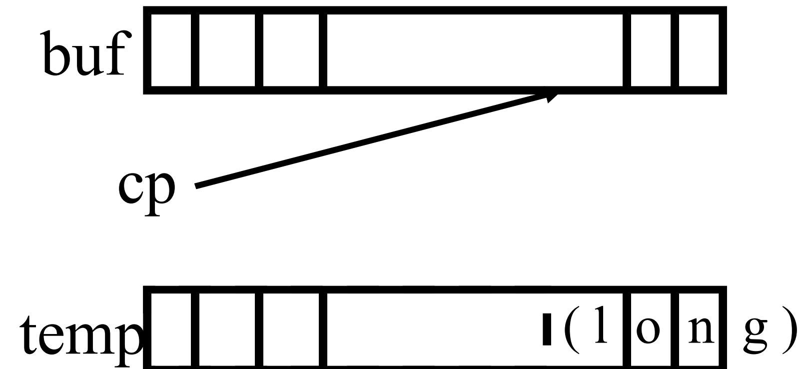
# Motivating Example 2

```
/* from web2c [fixwrites.c] */
#define BUFSIZ 1024
char       buf[BUFSIZ];

char insert_long(char *cp)
{
     char   temp[BUFSIZ];
     ...
     for (i = 0; &buf[i] < cp ; ++i)
          temp[i] = buf[i];

     strcpy(&temp[i],"(long)");
     strcpy(&temp[i+6],cp);
     ...
```



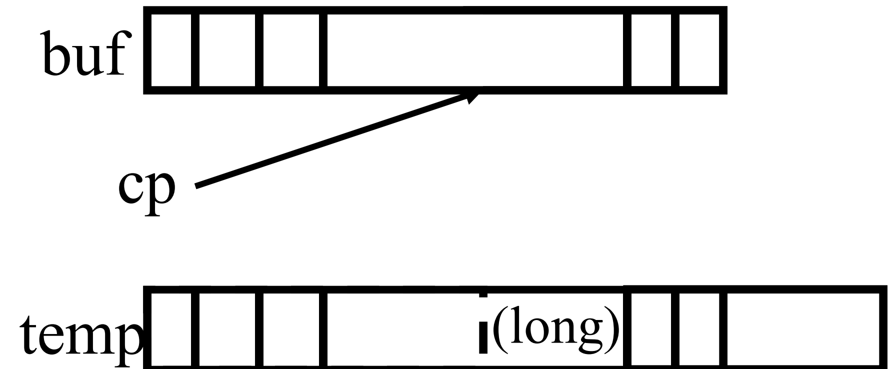
# Motivating Example 2

```
/* from web2c [fixwrites.c] */
#define BUFSIZ 1024
char      buf[BUFSIZ];

char insert_long(char *cp)
{
    char   temp[BUFSIZ];
    ...
    for (i = 0; &buf[i] < cp ; ++i)
        temp[i] = buf[i];

    strcpy(&temp[i],"(long)");
    strcpy(&temp[i+6],cp);
    ...
```



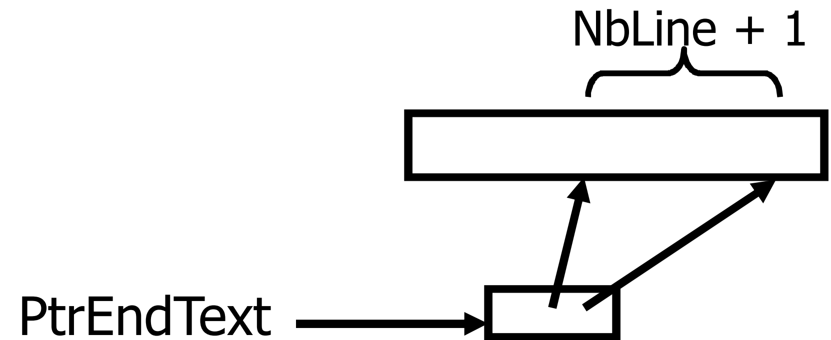
# Motivating Example 2

```
/* from web2c [fixwrites.c] */
#define BUFSIZ 1024
char       buf[BUFSIZ];

char insert_long(char *cp)
{
    char   temp[BUFSIZ];
    ...
    for (i = 0; &buf[i] < cp ; ++i)
        temp[i] = buf[i];

    strcpy(&temp[i],"(long)");
    strcpy(&temp[i+6],cp);
    ...
```



# Real Example

```
void SkipLine(const INT32   NbLine, char ** const  PtrEndText)
{
        INT32 indice;

        for (indice=0; indice<NbLine; indice++)
        {
                **PtrEndText = '\n';
                (*PtrEndText)++;
        }
        **PtrEndText = '\0';
        return;
}
```



# Real Example

```
void main() {
    char buf[SIZE];
    char *r, *s;
    r = buf;
    SkipLine(1,&r);
    fgets(r,SIZE-1,stdin);
    s = r + strlen(r);
    SkipLine(1,&s);
}
```

```
void SkipLine(const INT32   NbLine, char ** const
    PtrEndText) {
    INT32 indice;
    for (indice=0; indice<NbLine; indice++) {
        **PtrEndText = '\n';
        (*PtrEndText)++;
    }
    **PtrEndText = '\0';
    return;
}
```

# Vulnerable String Manipulation

- Pointers to buffers

```
char *p= buffer;
…
while(?) p++;
```

- Standard string manipulation functions

```
strcpy(), strcat(), …
```

- NULL termination

```
strncpy(), …
```

# String Violations are Common

- FUZZ study (1995)
  - Random test programs on various systems
  - 9 different UNIX systems
  - 18% – 23% hang or crash
  - 80% are string related errors
- CERT advisory
  - 50% of attacks are abuses of buffer overflows
  - (at least back in the day)

# Goals

- Static detection of string errors
  - References beyond array limit
  - Unsafe pointer arithmetic
  - Missing null terminator
  - Additional properties
    - References beyond null
    - Specified using preconditions

- Sound: never miss an error
- Precise: few false alarms

# Challenges in Static Analysis

- Soundness
- Precision
  - Combine integer and pointer analysis

    ```
    (p+i) = '\o';
    strcpy(q, p);
    ```
- Scale to handle real applications
  - Size: complexity of Chaotic iterations
  - Features: handle full C

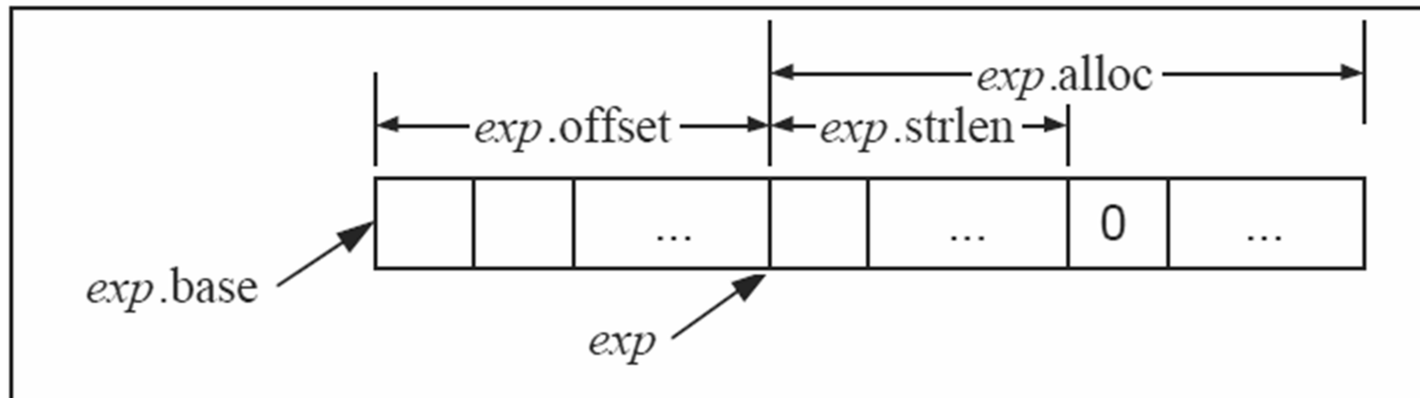
# But wait…

- pointer arithmetic
- casting from integer to pointer

- how do you even know that a memory access is illegal?
- do you even know what is "a string"?

# Instrumented Concrete Semantics

- explicitly represent
  - the base address of every memory location
  - allocated size starting from the base address



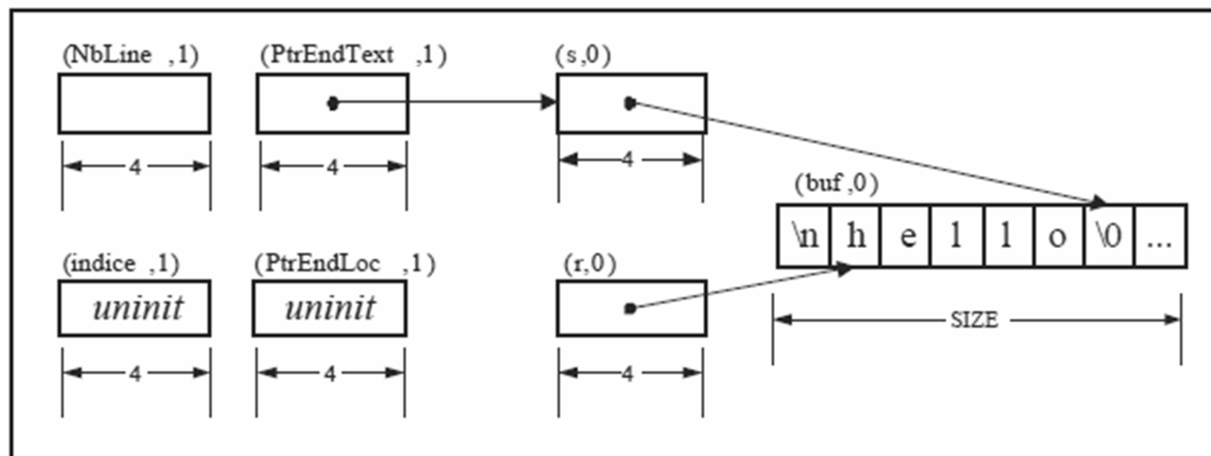
# Instrumented Concrete Semantics

| Shorthand | Meaning |
|---|---|
| $offset^\natural(\sigma^\natural)(l^\natural)$ | $l^\natural - base^\natural(\sigma^\natural)(l^\natural)$ |
| $alloc^\natural(\sigma^\natural)(l^\natural)$ | $size^\natural(\sigma^\natural)(base^\natural(\sigma^\natural)(l^\natural)) - offset^\natural(\sigma^\natural)(l^\natural)$ |
| $strlen^\natural(\sigma^\natural)(l^\natural)$ | $\begin{cases} i & content^\natural(\sigma^\natural)(l^\natural + i) = \text{'\textbackslash 0'} \wedge 0 \leq i < alloc^\natural(\sigma^\natural)(l^\natural) \wedge \\ & \neg\exists j : 0 \leq j < i \wedge content^\natural(\sigma^\natural)(l^\natural + j) = \text{'\textbackslash 0'} \\ undef & \text{otherwise} \end{cases}$ |
| $string^\natural(\sigma^\natural)(l^\natural)$ | $\begin{cases} true & strlen^\natural(\sigma^\natural)(l^\natural) \neq undef \\ false & \text{otherwise} \end{cases}$ |
| $is_overlap^\natural(\sigma^\natural)(l_1^\natural, l_2^\natural)$ | $\begin{cases} true & base^\natural(\sigma^\natural)(l_1^\natural) = base^\natural(\sigma^\natural)(l_2^\natural) \\ false & \text{otherwise} \end{cases}$ |
| $overlap^\natural(\sigma^\natural)(l_1^\natural, l_2^\natural)$ | $\begin{cases} offset^\natural(\sigma^\natural)(l_2^\natural) - offset^\natural(\sigma^\natural)(l_1^\natural) & is_overlap^\natural(\sigma^\natural)(l_1^\natural, l_2^\natural) \\ undef & \text{otherwise} \end{cases}$ |

# Real Example

```
void main() {
    char buf[SIZE];
    char *r, *s;
    r = buf;
    SkipLine(1,&r);
    fgets(r,SIZE-1,stdin);
    s = r + strlen(r);
    SkipLine(1,&s);
}
```

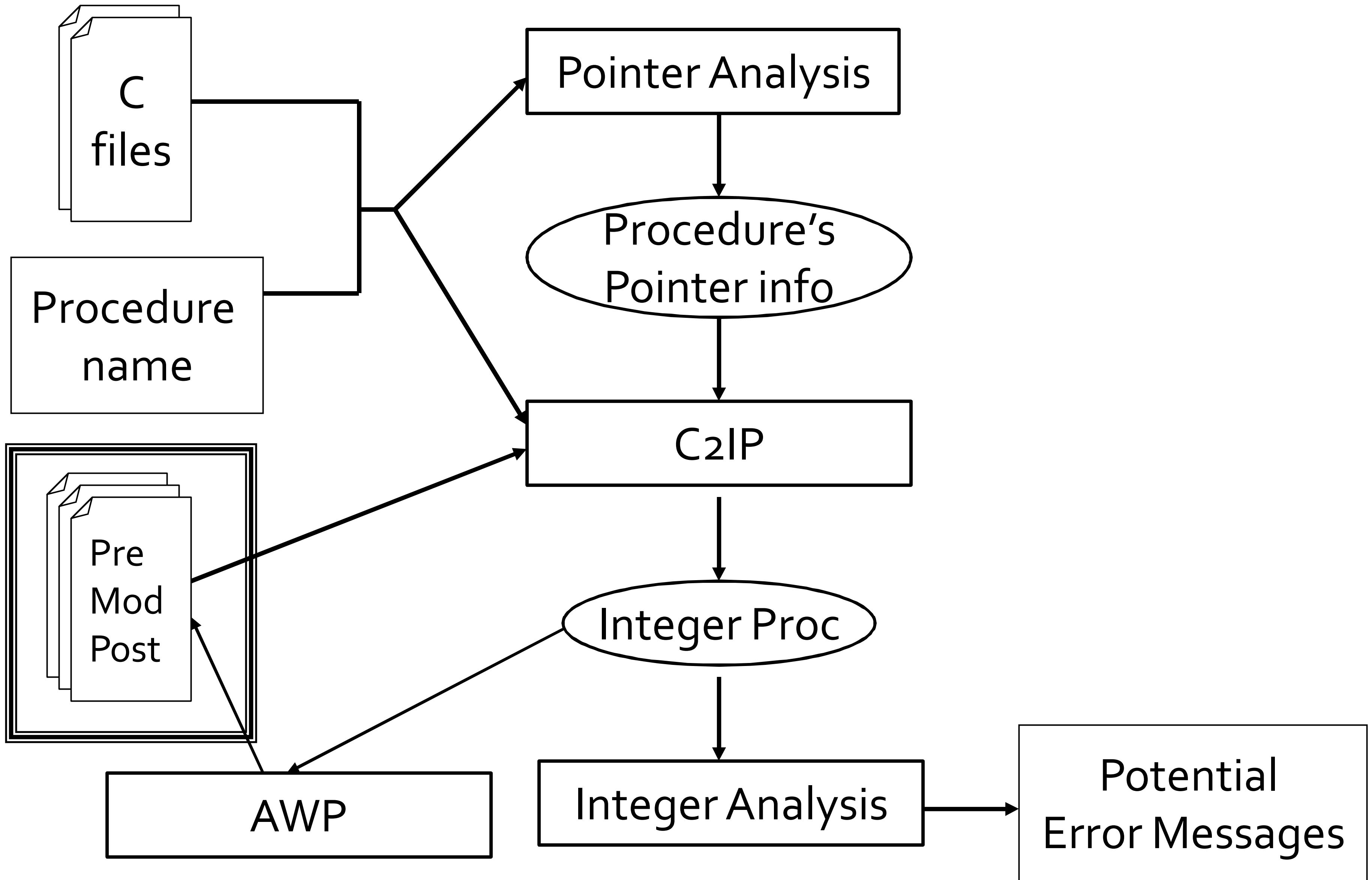
```
void SkipLine(const INT32   NbLine, char ** const
    PtrEndText) {
    INT32 indice;
    for (indice=0; indice<NbLine; indice++) {
        **PtrEndText = '\n';
        (*PtrEndText)++;
    }
    **PtrEndText = '\0';
    return;
}
```



# CSSV Approach

- Use powerful (and expensive) static domain
  - Exponential abstract interpretation
- Use pre- and post-conditions to specify procedure requirements on strings
  - No inter-procedural analysis
  - Modular analysis
- Automatic generation of procedure specification

# CSSV



# Advantages of Specifications

- Enable modular analysis
  - Not all the code is available
  - Enables more precise analyses
- Programmer controls verification
  - Detect errors at point of logical error
  - Improve analysis precision
  - Check additional properties
    - Beyond ANSI-C

# Specification and Soundness

- Preconditions are handled conservatively
- All errors are detected
  - Inside a procedure's body
    OR
  - At call statements to the procedure

# Specification – strcpy

char* strcpy(char* dst, char *src)

requires ( string(src) $\wedge$
alloc(dst) > len(src)
)

mod dst.strlen, dst.is_nullt

ensures ( len(dst) == pre@len(src) $\wedge$
return == pre@dst
)

# Specification – insert_long()

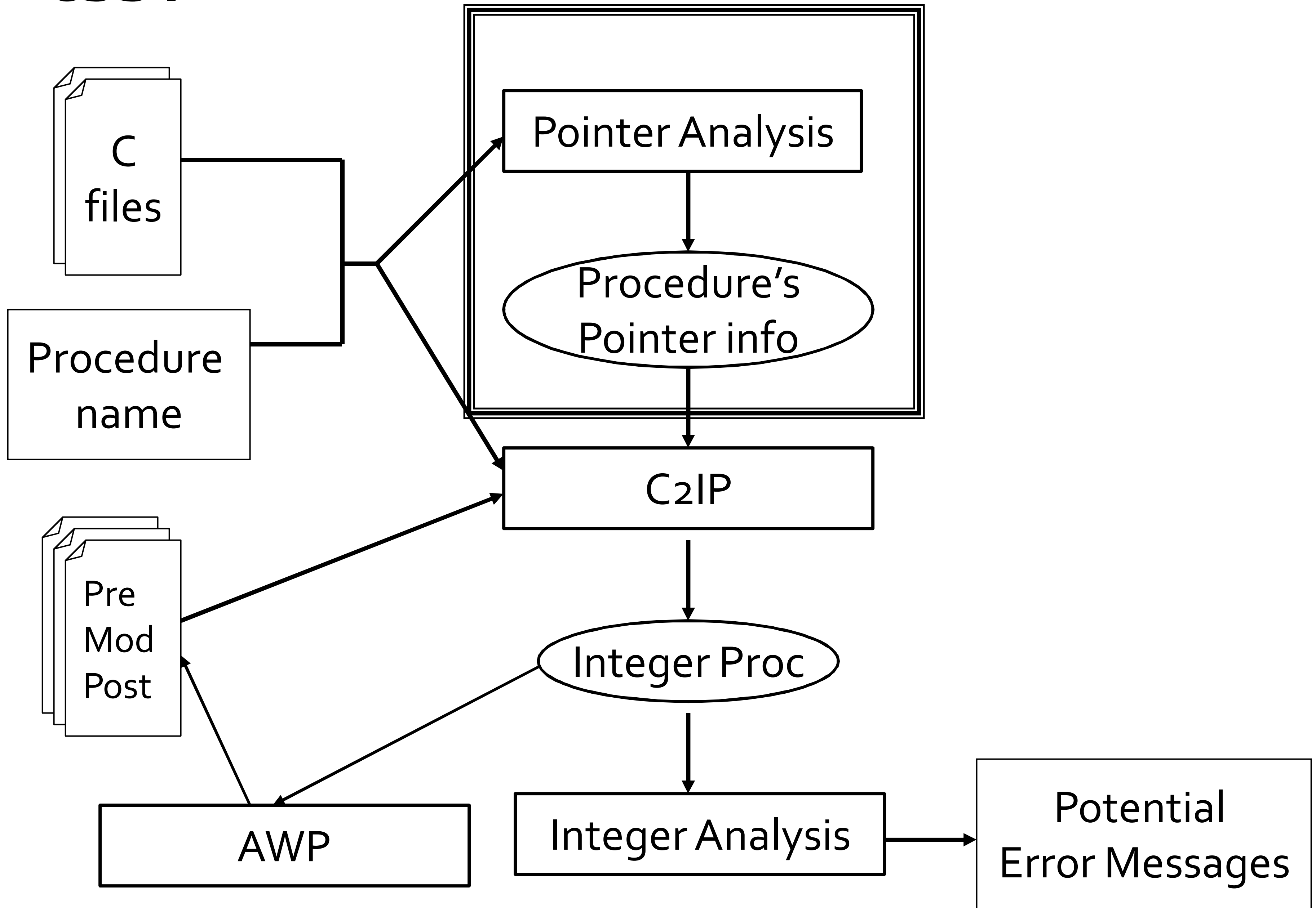
```
/* insert_long.c */
#include "insert_long.h"
char buf[BUFSIZ];
char* insert_long(char *cp) {
 char temp[BUFSIZ];
 int i;
 for (i=0; &buf[i] < cp; ++i){
  temp[i] = buf[i];
 }
 strcpy (&temp[i],"(long)");
 strcpy (&temp[i + 6], cp);
 strcpy (buf, temp);
 return cp + 6;
}
```

char * insert_long(char *cp)
  requires( string(cp) $\wedge$
    buf $\leq$ cp $<$ buf + BUFSIZ
    )
  mod cp.strlen
  ensures (
   cp.strlen = = pre[cp.strlen + 6]
    $\wedge$
   return_value = = cp + 6 ;
    )

# Challenges with Specifications

- Legacy code
- Complexity of software
- Need to know context

# CSSV



# CSSV – Pointer Analysis

- Models the string objects
- Pre compute points-to information
- Determines which objects may be updated through a pointer

```
char        s[20];
char        *p;
…
p = s + 5;
strcpy(p,” world!”);
```

# Pointer Analysis in General

- points-to analysis
- shape analysis

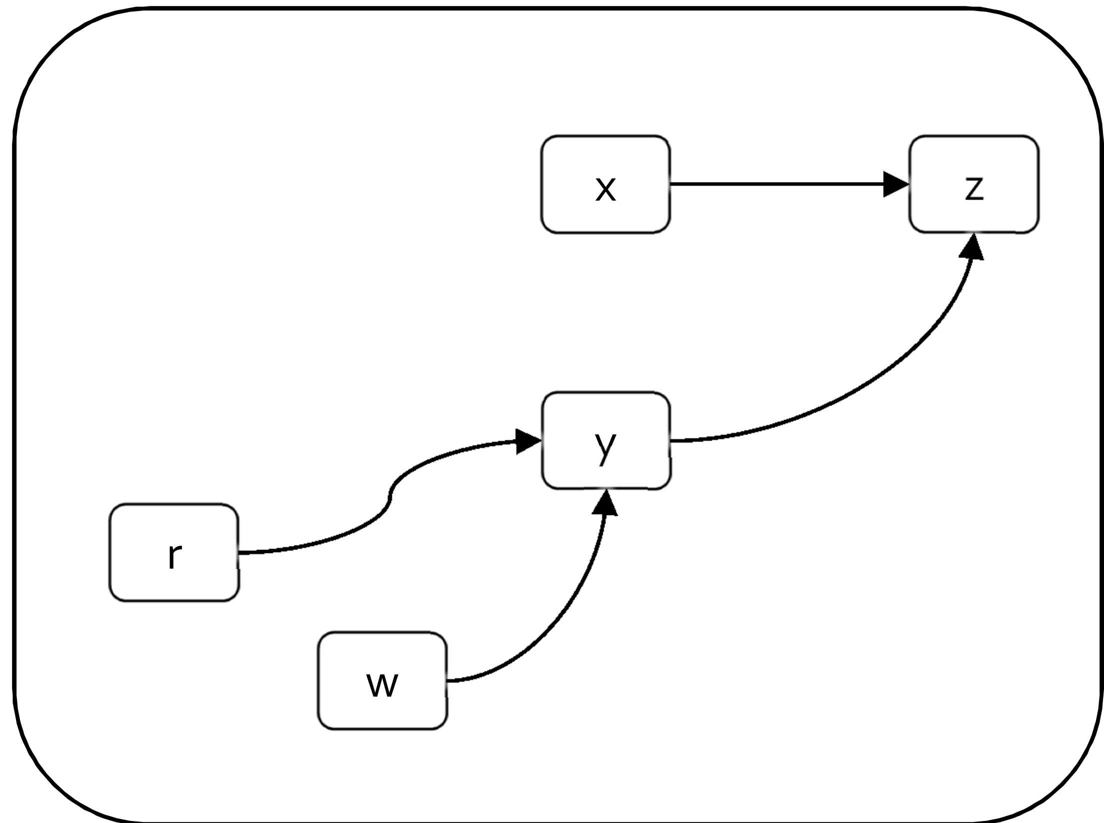
# Points-to Analysis

- Aliases
  - Two pointers p and q are aliases if they point to the same memory location
- Points-to pair
  - (p,q) means p holds the address of q
- Points-to pairs and aliases
  - (p,q) and (r,q) means that p and r are aliases

- Challenge: no a priori bound on the set of heap locations

# Terminology Example

[x := &z][1]
[y := &z][2]
[w := &y][3]
[r := w][4]




Points-to pairs: (x,z), (y,z), (w,y), (r,y)
Aliases: (x,y), (r,w)

# (May) Points-to Analysis

- Property Space
  - L = ( $\wp$(Var x Var),$\subseteq$,$\cup$,$\cap$,$\emptyset$, Var x Var )

- Transfer functions

| Statement | out(lab) |
|---|---|
| $[p = \&x]^{lab}$ | in(lab) $\cup$ { (p,x) } |
| $[p = q]^{lab}$ | in(lab) U { (p,x) \| (q,x) $\in$ in(lab) } |
| $[*p = q]^{lab}$ | in(lab) U { (r,x) \| (q,x) $\in$ in(lab) and (p,r) $\in$ in(lab) } |
| $[p = *q]^{lab}$ | in(lab) U { (p,r) \| (q,x) $\in$ in(lab) and (x,r) $\in$ in(lab) } |

# (May) Points-to Analysis

- What to do with malloc?
- Need some static naming scheme for dynamically allocated objects

- Single name for the entire heap
  - Aobj = (Var $\cup$ {H})
  - L = ( $\wp$(Var x Aobj), $\subseteq$, $\cup$, $\cap$, $\emptyset$, Var x Aobj )
  - $[\![$[p = malloc]$^{lab}]\!]$ (S) = S $\cup$ { (p,H) }
- Name based on static allocation site
  - Aobj = (Var $\cup$ { (x,l) | stmt(l) is x = malloc(…)})
  - $[\![$[p = malloc]$^{lab}]\!]$ (S) = S $\cup$ { (p,lab) }

# (May) Points-to Analysis

```
[x :=malloc]1;        ∅
                      { (x,H) }
[y:=malloc]2;
                      { (x,H), (y,H) }
(if [x==y]3 then
                      { (x,H), (y,H) }
   [z:=x]4
                      { (x,H), (y,H), (z,H) }
else
   [z:=y]5
                      { (x,H), (y,H), (z,H) }
);
                      { (x,H), (y,H), (z,H) }
```

Single name H for the entire heap

# Allocation Sites

- Divide the heap into a fixed partition based on allocation site
- All objects allocated at the same program point represented by a single "abstract object"

| | | | |
|---|---|---|---|
| AS1 | AS2 | AS3 | AS1 |
| AS2 | AS3 | AS3 | AS2 |

# (May) Points-to Analysis

```
[x :=malloc]1; // A1
[y:=malloc]2; // A2
(if [x==y]3 then
   [z:=x]4
else
  [z:=y]5
);
```

$\emptyset$

{ (x,A1) }

{ (x,A1), (y,A2) }

{ (x,A1), (y,A2) }

{ (x,A1), (y,A2), (z,A1) }

{ (x,A1), (y,A2), (z,A2) }

{ (x,A1), (y,A2), (z,A1), (z,A2) }

Allocation-site based naming (using $A_{lab}$ instead of just "lab" for clarity)

# Weak Updates

| Statement | out(lab) |
|---|---|
| $[p = \&x]^{lab}$ | $in(lab) \cup \{ (p,x) \}$ |
| $[p = q]^{lab}$ | in(lab) **U** { (p,x) \| (q,x) ∈ in(lab) } |
| $[*p = q]^{lab}$ | in(lab) **U** { (r,x) \| (q,x) ∈ in(lab) and (p,r) ∈ in(lab) } |
| $[p = *q]^{lab}$ | in(lab) **U** { (p,r) \| (q,x) ∈ in(lab) and (x,r) ∈ in(lab) } |

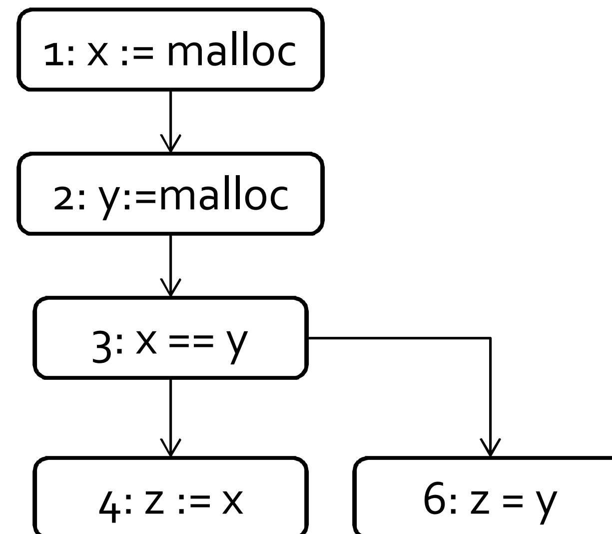
```
                              ∅
[x :=malloc]¹; // A1
                              { (x,A1) }
[y:=malloc]²; // A2
                              { (x,A1), (y,A2) }
[z:=x]³;
                              { (x,A1), (y,A2), (z,A1) }
[z:=y]⁴;
                              { (x,A1), (y,A2), (z,A1), (z,A2) }
```

# (May) Points-to Analysis

- Fixed partition of the (unbounded) heap to static names
  - Allocation sites
  - Types
  - Calling contexts
  - …
- Scalable points-to analysis typically flow-insensitive
  - Ignoring the structure of the flow in the program

# Flow-sensitive vs. Flow-insensitive Analyses

```
[x :=malloc]¹;
[y:=malloc]²;
(if [x==y]³ then
   [z:=x]⁴
else
  [z:=y]⁵
);
```




- Flow sensitive: respect program flow
  - a separate set of points-to pairs for every program point
  - the set at a point represents possible may-aliases on some path from entry to the program point
- Flow insensitive: assume all execution orders are possible, abstract away order between statements

# Flow Insensitive

| **Statement** | **pt sets** |
|---|---|
| $[p = malloc]^{lab}$ | $Alab \in pt(p)$ |
| $[p = \&x]^{lab}$ | $x \in pt(p)$ |
| $[p = q]^{lab}$ | $pt(q) \subseteq pt(p)$ |
| $[*p = q]^{lab}$ | $\forall a \in pt(p).\ pt(q) \subseteq pt(a)$ |
| $[p = *q]^{lab}$ | $\forall a \in pt(q)\ .\ pt(a) \subseteq pt(p)$ |

$[x := malloc]^1$; // A1 — $A1 \in pt(x)$

$[y := malloc]^2$; // A2 — $A2 \in pt(y)$

$[z := x]^3$; — $pt(x) \subseteq pt(z)$

$[z := y]^4$; — $pt(y) \subseteq pt(z)$

# Flow Insensitive Points-to

| | |
|---|---|
| $[x := \text{malloc}]^1$; // A1 | $A1 \in pt(x)$ |
| $[y := \text{malloc}]^2$; // A2 | $A2 \in pt(y)$ |
| $[z := x]^3$; | $pt(x) \subseteq pt(z)$ |
| $[z := y]^4$; | $pt(y) \subseteq pt(z)$ |

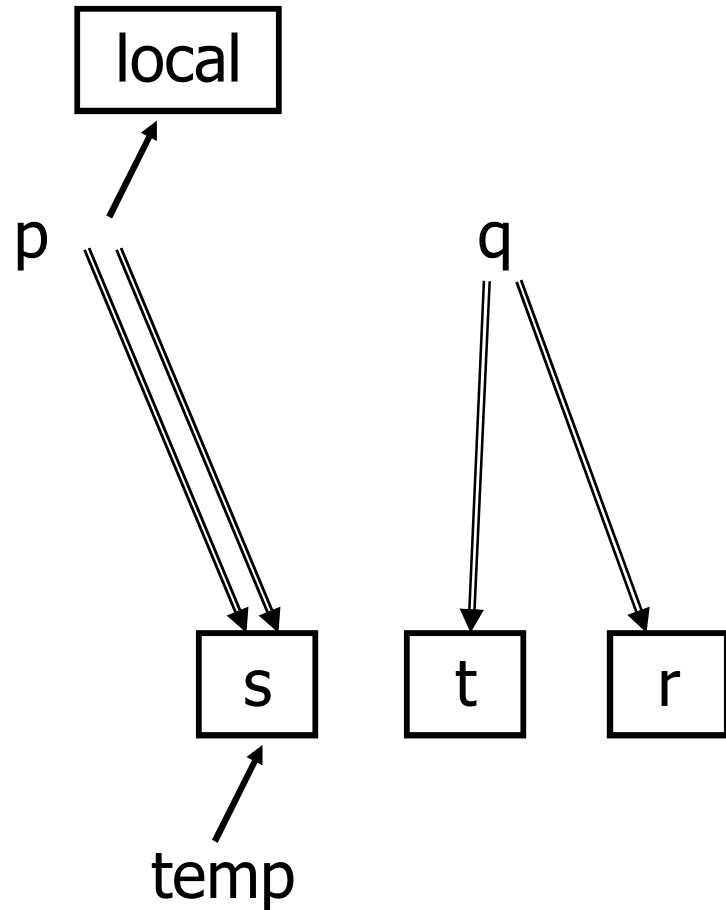
| | | | | | |
|---|---|---|---|---|---|
| pt(x) = ø | pt(x) = {A1} | pt(x) = {A1} | pt(x) = {A1} | pt(x) = {A1} | pt(x) = {A1} |
| pt(y) = ø | pt(y) = ø | pt(y) = ø | pt(y) = ø | pt(y) = {A2} | pt(y) = {A2} |
| pt(z) = ø | pt(z) = ø | pt(z) = {A1} | pt(z) = {A1} | pt(z) = {A1} | pt(z) = {A1,A2} |

# Back to CSSV

- We need a points-to analysis
- But we need it to be modular…

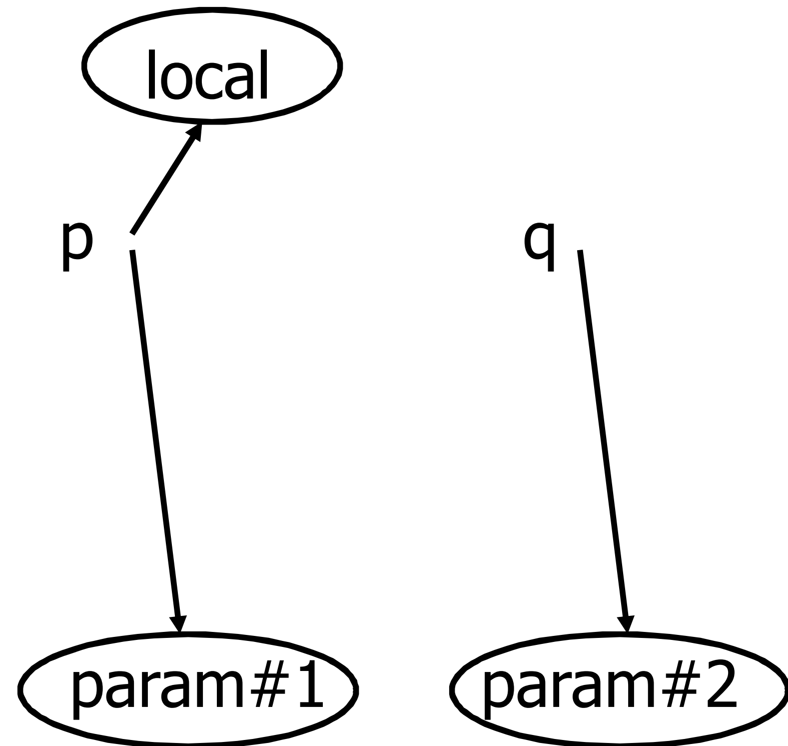
# Integrating Pointer Information?

```
foo(char *p, char *q)
{
        char local[100];
        …
        p = local;
        *q  = 0;
        …
}
main()
{
        char s[10], t[20], r[30];
        char *temp;
        foo(s,t);
        foo(s,r);
        …
        temp = s
        …
}
```

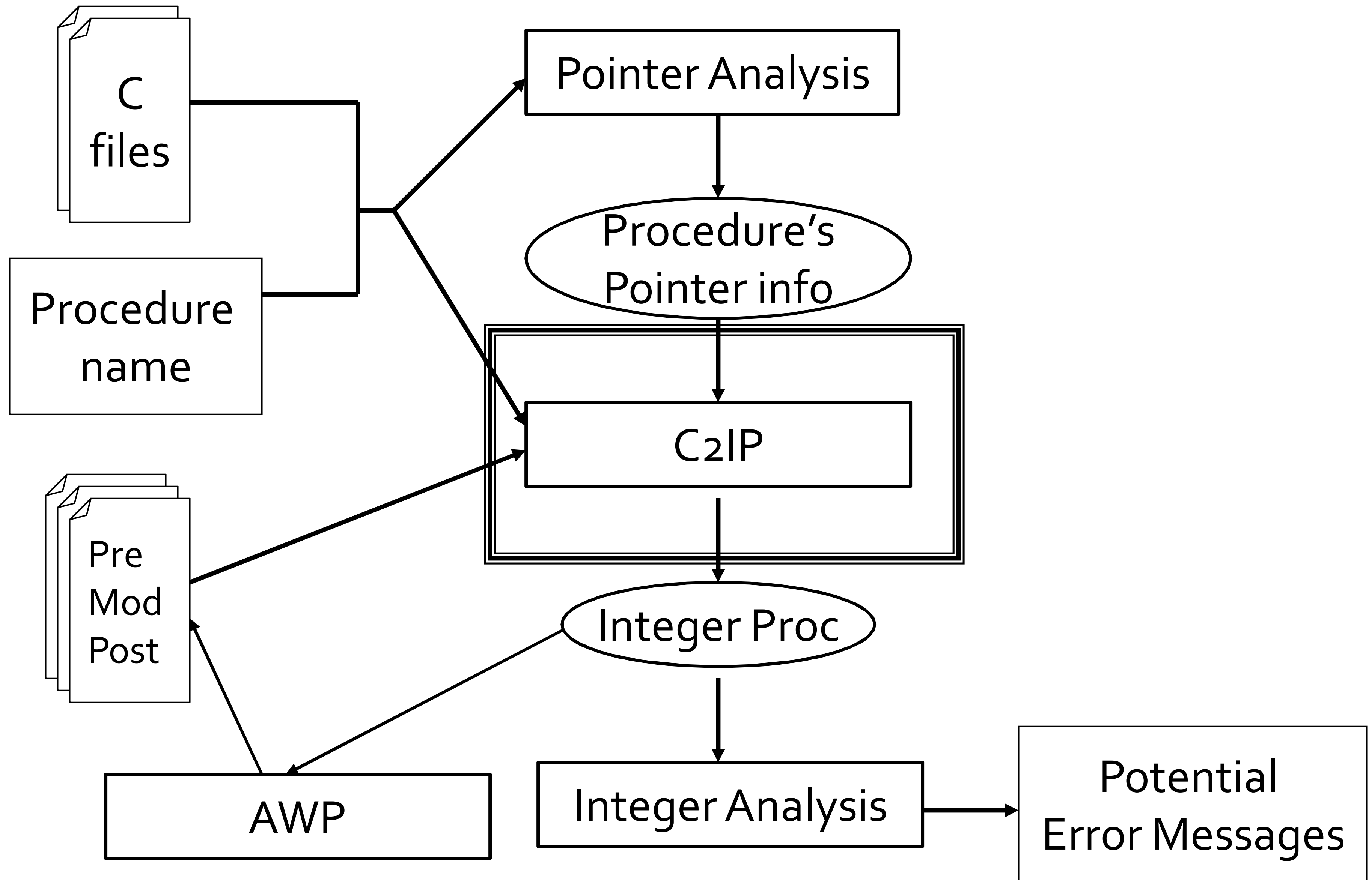


# Projection for foo()

```
foo(char *p, char *q)
{
        char local[100];
        …
        p = local;
        …
}
```



# CSSV

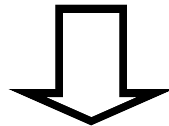


# C2IP – C to Integer Program

- Generate an *integer program*
  - Integer variables only
  - No function calls
  - Non deterministic

- Goal

String error in the C program



Assert violated in the IP

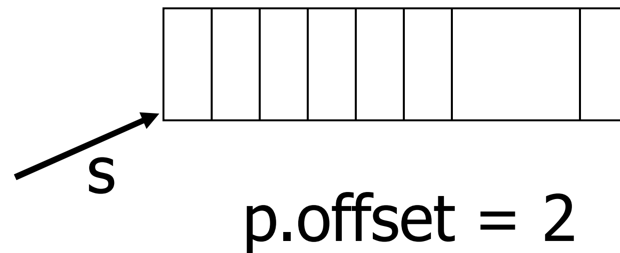
# C2IP – C to Integer Program

- Inline specification
- Based on points-to information
  - Generate *constraint variables*
  - Generate assert statements
  - Generate update statements

# C2IP - Constraint Variable

- For every pointer

  *p*.offset



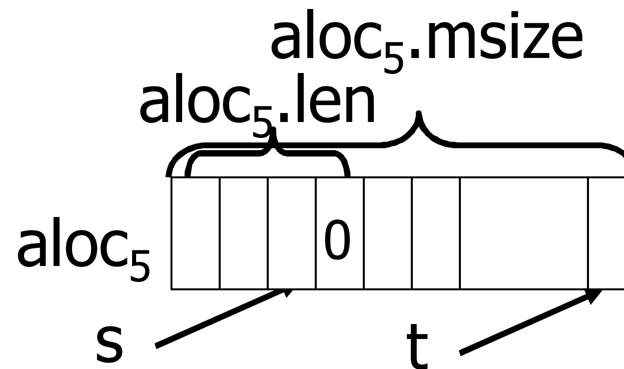
# C2IP - Constraint Variable

- For every abstract location

  *aloc.*is_nullt

  *aloc*.len

  *aloc.*msize



# C2IP

```
char buf[BUFSIZ];
```

```
int  buf.offset = 0;
int  s_buf.msize = BUFSIZ;
int  s_buf.len;
int  s_buf.is_nullt;
```

```
char * insert_long (char *cp) {
char temp[BUFSIZ]
```

```
int  cp.offset;
int  temp.offset = 0;
int  s_temp.msize = BUFSIZ;
int  s_temp.len ;
int  s_temp.is_nullt;
```

```
int i
```

```
int i
```

```
require string(cp);
```

```
assume(s_buf.is_nullt  ∧ 0 ≤ cp.offset ≤ s_buf.len ≤ s_buf.alloc );
```

```
for(i=0; &buf[i] < cp; ++i) {
    temp[i]=cp[i];
}
```

```
for (i=0; i< cp.offset ; ++i )  {
   assert(0 ≤ i ≤ s_temp.msize ∧
     (s_temp.is_nullt→ i≤ s_temp.len));
   assert(-i ≤cp.offset< -i +s_buf.len);
   if (s_buf.is_nullt ∧ s_buf.len == i )  {
       s_temp.len = i;  s_temp.is_nullt = true;   }
   else ...
```

# C2IP

```
char * insert_long (char *cp) {
char temp[BUFSIZ]

int i

require string(cp);

for(i=0; &buf[i] < cp; ++i) {
    temp[i]=cp[i];
}

strcpy(&temp[i],"(long)");
```

int cp.offset;
int temp.offset = 0;
int $s_{temp}$.msize = BUFSIZ;
int $s_{temp}$.len ;
int $s_{temp}$.is_nullt;

int i

assume($s_{buf}$.is_nullt $\wedge$ $0 \leq$ cp.offset $\leq s_{buf}$.len $\leq s_{buf}$.alloc );

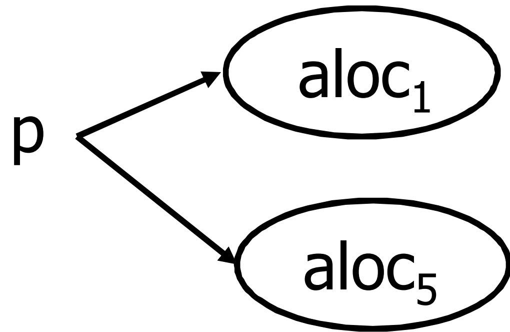
for (i=0; i< cp.offset ; ++i ) {
  assert($0 \leq i \leq s_{temp}$.msize $\wedge$
    ($s_{temp}$.is_nullt$\rightarrow$ i$\leq s_{temp}$.len));
  assert(-i $\leq$cp.offset< -i +$s_{buf}$.len);
  if ($s_{buf}$.is_nullt $\wedge$ $s_{buf}$.len == i ) {
    $s_{temp}$.len = i; $s_{temp}$.is_nullt = true; }
  else ...

assert($0 \leq i <$ 6 - $s_{temp}$.msize );
assume($s_{temp}$.len == i + 6);...

# C2IP - Update statements

```
p  = s + 5;
p.offset = s.offset + 5;
```

# C2IP - Use points-to information




```
*p  = 0;
```

```
if (…) {
        aloc1.len = p.offset;
        aloc1.is_nullt = true; }
else {
        alloc5.len = p.offset;
        alloc5.is_nullt = true; }
```

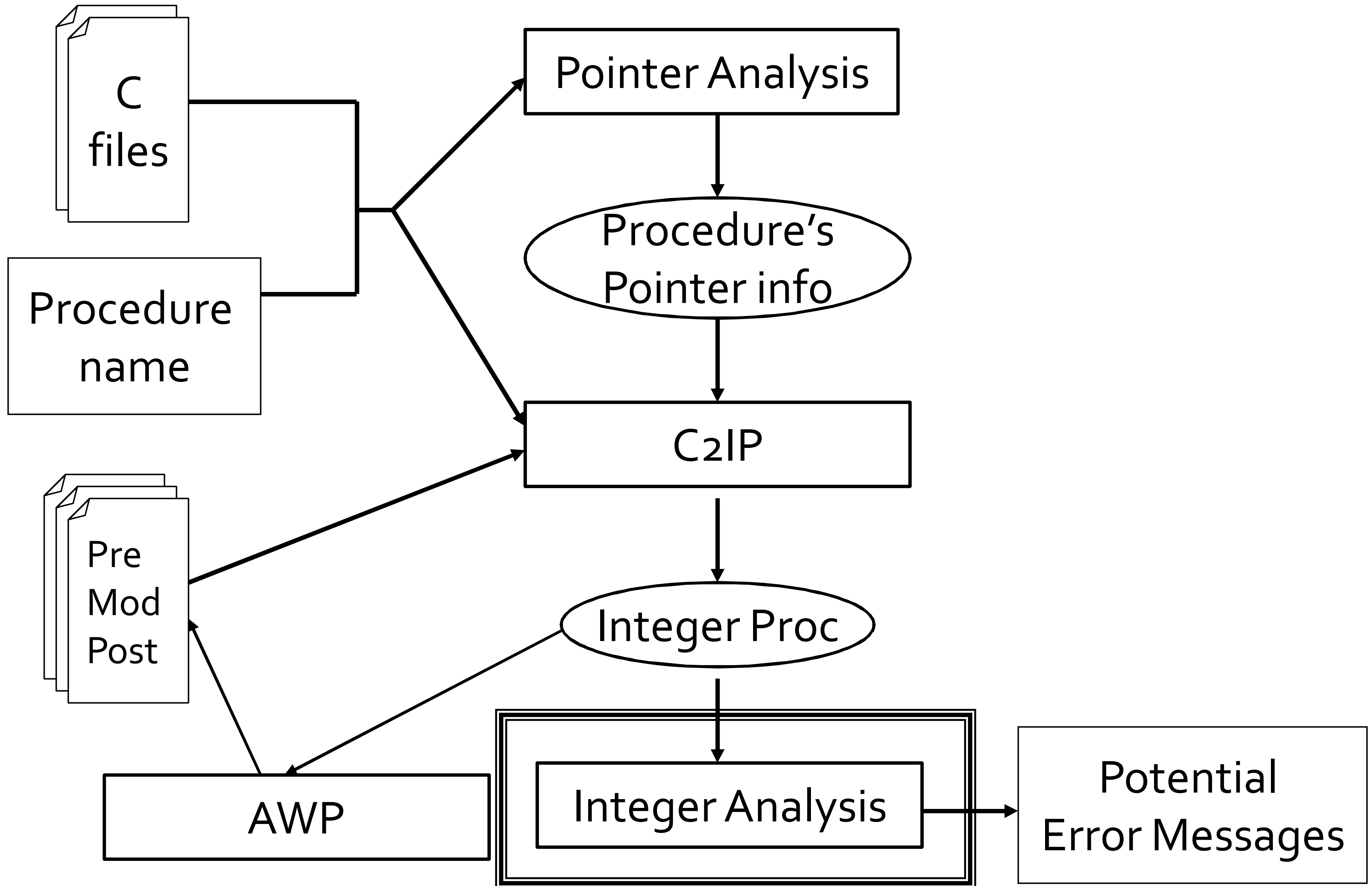
# C2IP

| Construct | IP Statements |
|---|---|
| p = Alloc(i) | lvp.offset = 0<br>rvp.aSize = lvi.val<br>rvp.isnullt = false |
| p = q + i | lvp.offset = lvq.offset + lvi.val |
| *p = c | if c = 0 {<br>  rvp.len = lvp.offset<br>  rvp.isnullt = true<br>} else {<br>  if rvp.isnullt and lvp.offset = rvp.len<br>     lvp.isnullt = unknown<br>} |
| c = *p | if rvp.isnullt and lvp.offset = rvp.len<br>   lvc.val = 0<br>else lvc.val = unknown |
| g(a1,…,am) | mod[g](a1…am) |

# C2IP

| Construct | IP Statements |
|---|---|
| *p == 0 | rvp.isnullt and rvp.len == lvp.offset |
| p > q | lvp.offset > lvq.offset |
| p.alloc | rvp.aSize – lvp.offset |
| p.offset | lvp.offset |
| p.isnullt | rvp.isnullt |
| p.strlen | rvp.lep – lvp.offset |

# CSSV



# Integer Analysis

- Interval analysis is not enough

assert($-i \le cp.offset < -i + s_{buf}.len$);

- Use a powerful abstract domain
- Polyhedra (Cousot Halbwachs, 78)

◆ Statically analyzes program variable relations and detects constraints:

$$a1 * var1 + a2 * var2 + \ldots + a_n * var_n \le b$$

# Linear Relation Analysis

- Statically analyzes program variable relations and detects constraints:

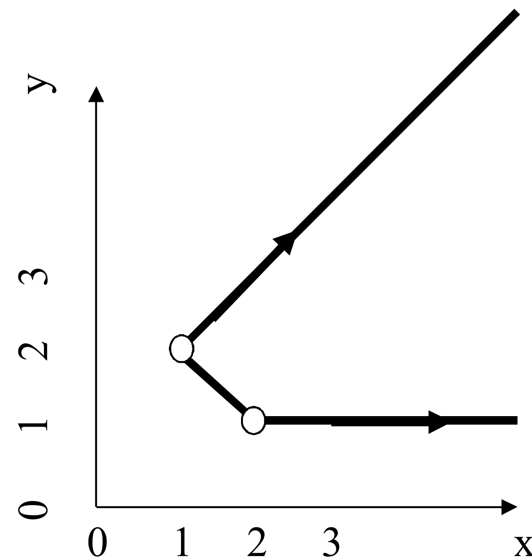
$$a_1 * var_1 + a_2 * var_2 + \ldots + a_n * var_n \leq b$$

- Polyhedron

$$y \geq 1$$
$$x + y \geq 3$$
$$-x + y \leq 1$$

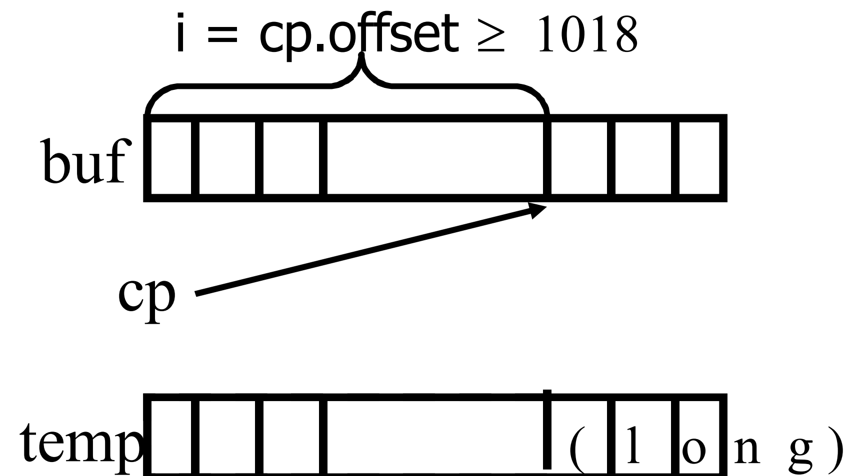
V = <(1,2) (2,1) >

R = <(1,0) (1,1)>



# Integer Analysis – insert_long()

buf.offset = 0
temp.offest = 0
$0 \le cp.offset = i$
$i \le s_{buf}.len < s_{buf}.msize$
$s_{buf}.msize = 1024$
$s_{temp}.msize = 1024$

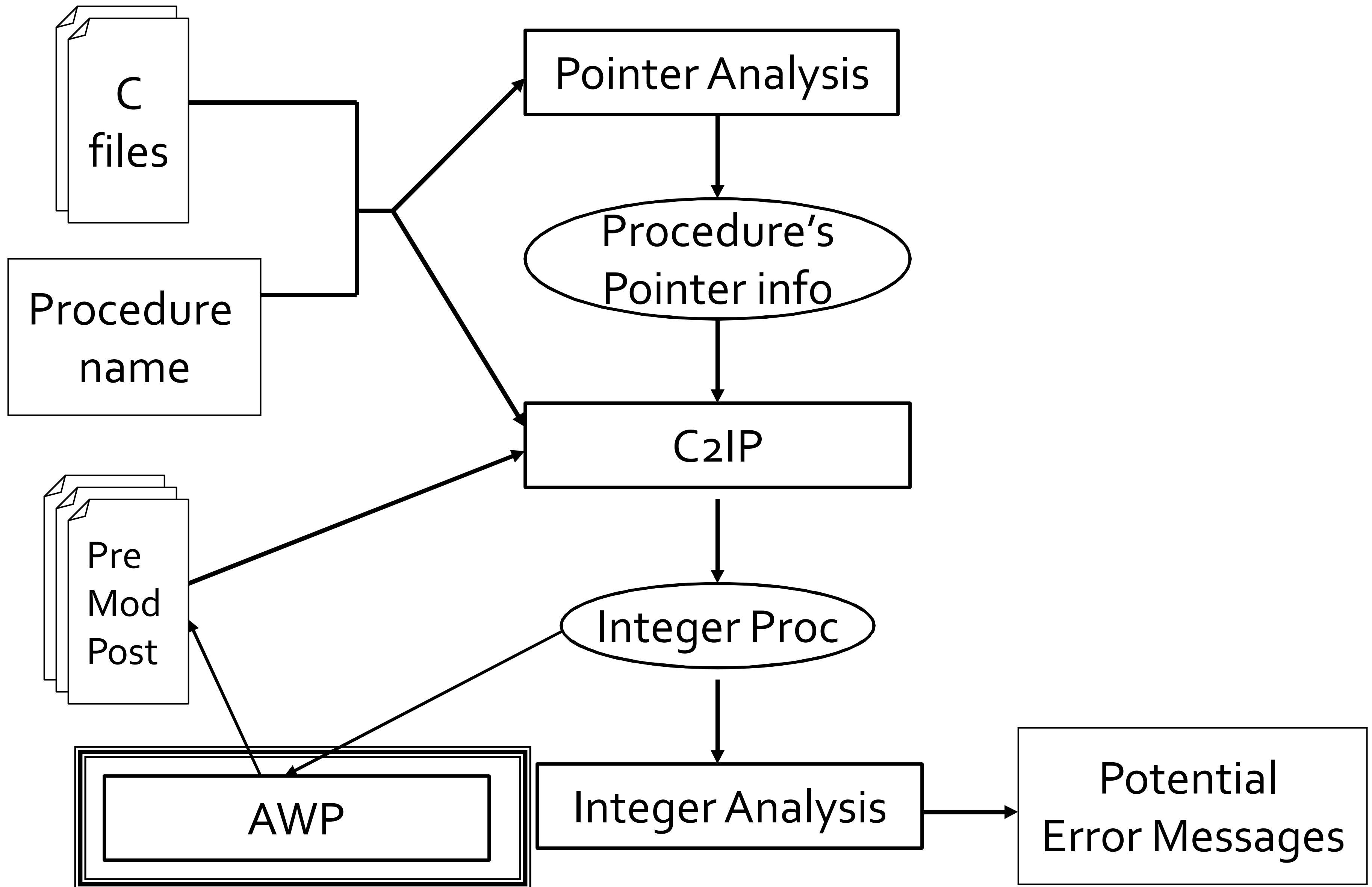



assert($0 \le i < 6 - s_{temp}.msize$ );   // strcpy(&temp[i],"(long)");

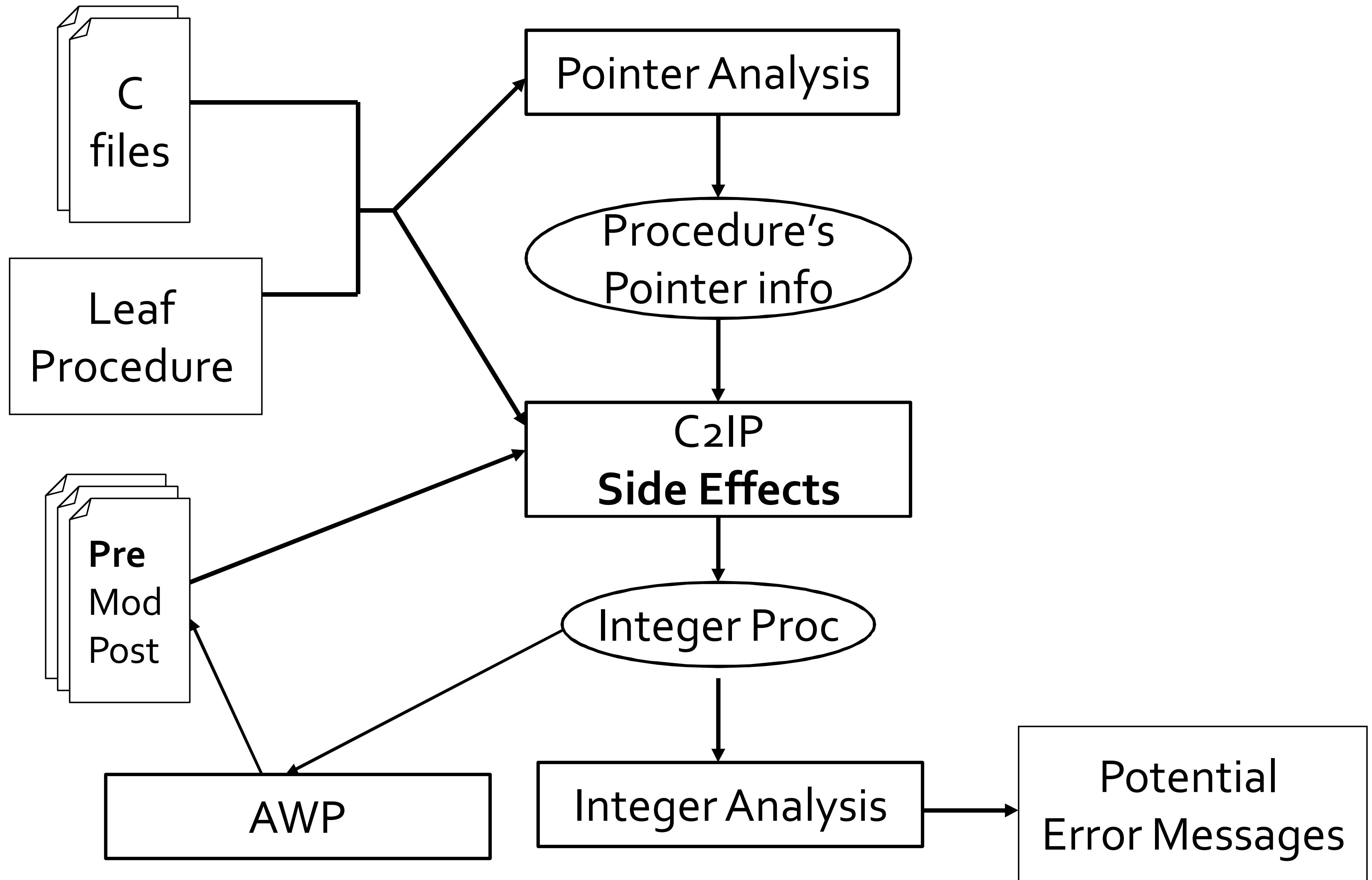
Potential violation when

cp.offset ≥ 1018

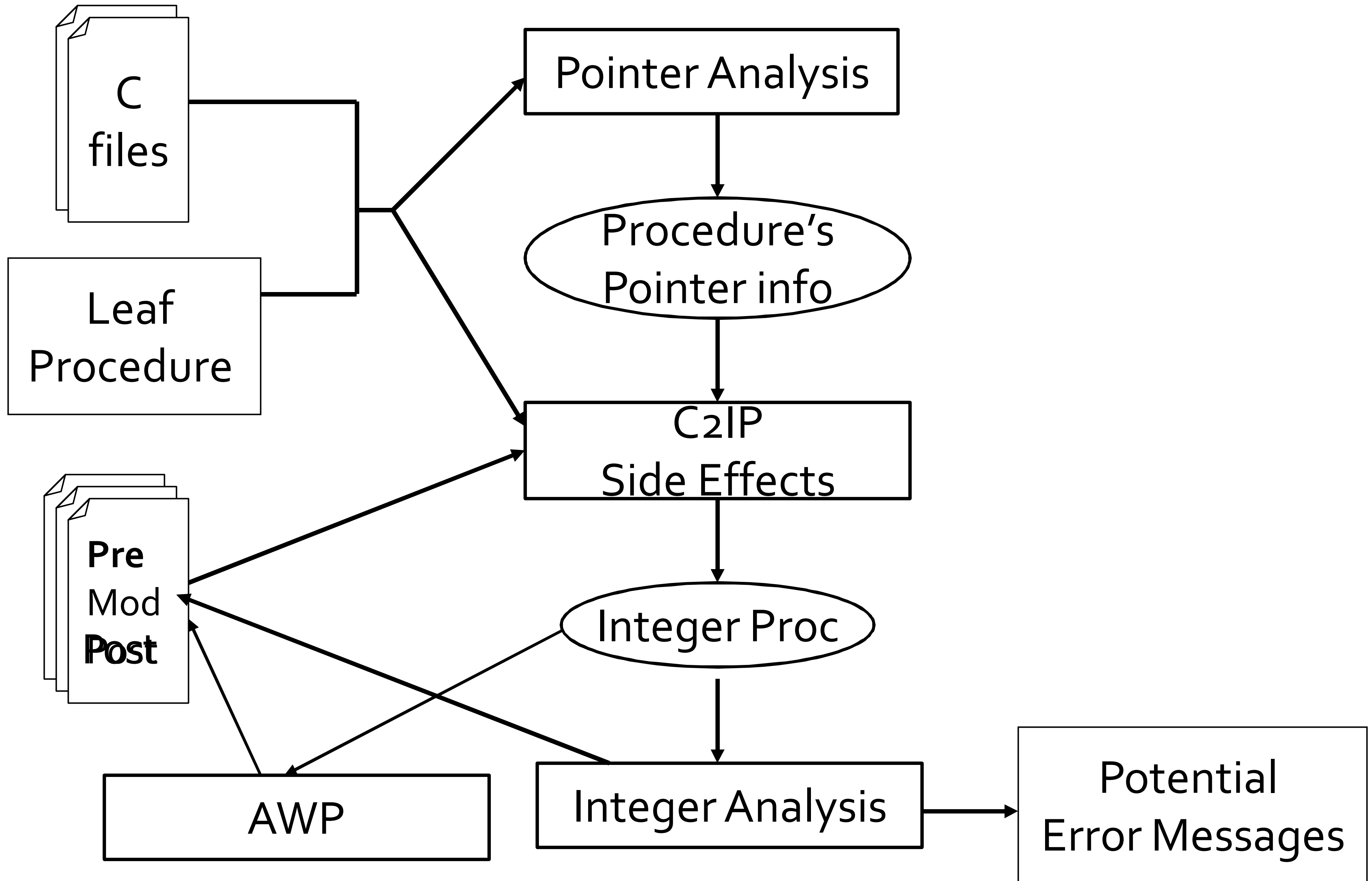
# CSSV



# CSSV



# CSSV



# AWP

- Approximate the Weakest Precondition
- Backward integer analysis
- Generates a precondition

# AWP – insert_long()

- Generate the following precondition:

  $s_{buf}$.is_nullt $\wedge$

  $s_{buf}$.len < $s_{buf}$.alloc $\wedge$

  $o \leq$ cp.offset $\leq s_{buf}$.len $\wedge$

  …

# AWP – insert_long()

- Generate the following precondition:

string(cp) $\wedge$

$s_{buf}.len \leq cp.offset + 1017$

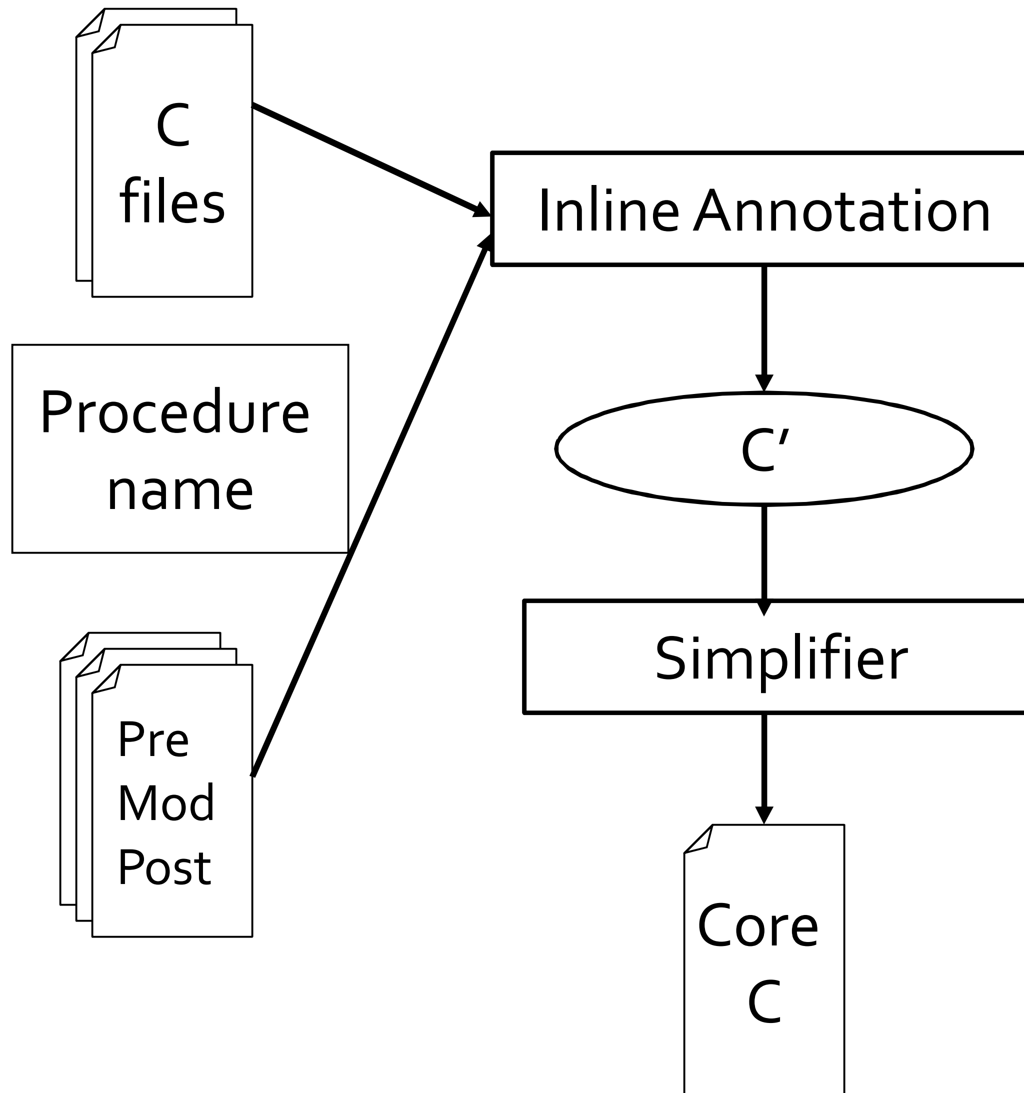
◆ Not the weakest precondition:

string(cp) $\wedge$ $s_{buf}.len \leq 1017$

# Implementation

- ASToolKit [Microsoft]
- GOLF [Microsoft – Das Manuvir]
- New Polka [IMAG - Bertrand Jeannet]
- Main steps
  - Simplifier
  - Pointer analysis
  - C2IP
  - Integer Analysis

# Implementation – step 1



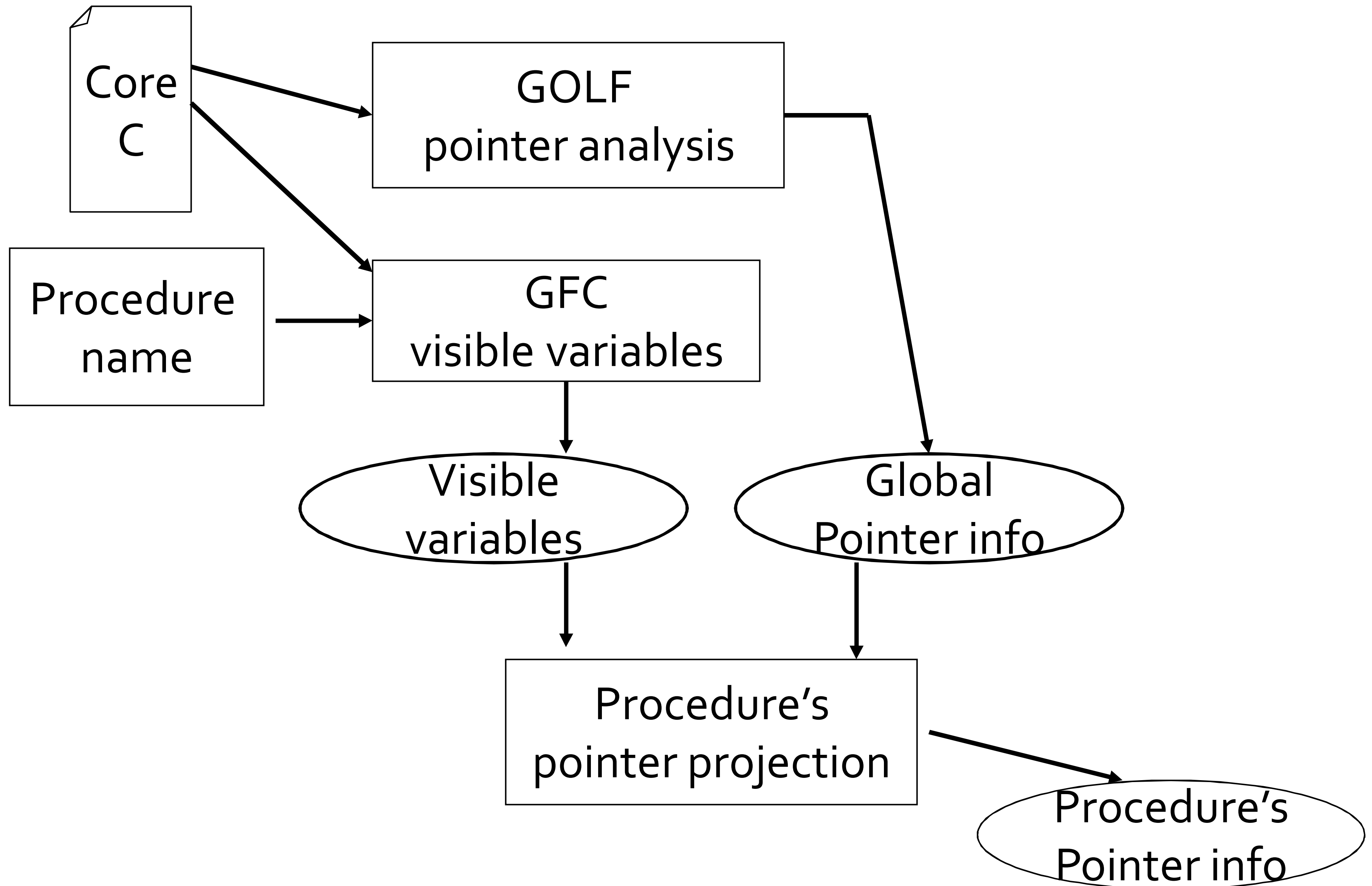
# `Core C`

- Simplify the analysis implementation
- A limited form of C expressions
  - Adds temporaries
  - At most one operator per statement
  - Convert value into location computation
  - No loss of precision

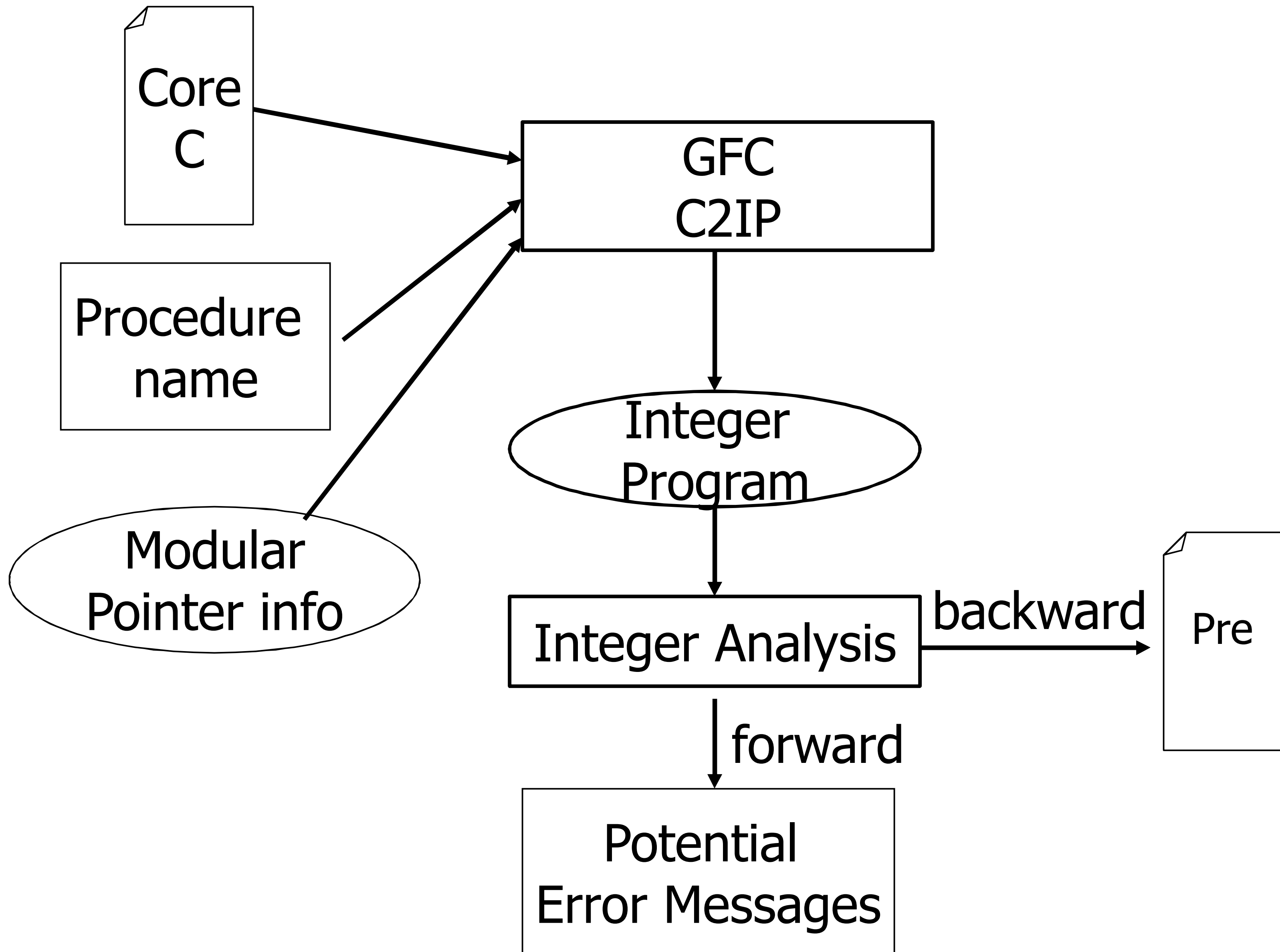
```
void SkipLine(const INT32 NbLine, char ** const PtrEndText){
  INT32 indice;
  for (indice=0; indice<NbLine; indice++) {
      **PtrEndText = '\n';
      (*PtrEndText)++;
  }
  **PtrEndText = '\0';
  return;
}
```

```
void SkipLine(int NbLine, char** PtrEndText) {
  int indice;
  char* PtrEndLoc;
  indice=0;
  begin_loop:
  if (indice>=NbLine) goto end_loop;
  PtrEndLoc = *PtrEndText;
  *PtrEndLoc = '\n';
  *PtrEndText = PtrEndLoc + 1;
  indice = indice + 1;
  goto begin_loop;
  end_loop:
  PtrEndLoc = *PtrEndText
  *PtrEndLoc = '\0';
}
```

# Implementation – step 2



# Implementation – step 3 , 4



# Results (web2C)

| Proc | line | coreC line | time (sec) | space (Mb) | errors | FA |
|---|---|---|---|---|---|---|
| insert_long | 14 | 64 | 2.0 | 13 | 2 | 0 |
| fprintf_pascal_string | 10 | 25 | 0.1 | 0.3 | 2 | 0 |
| space_terminate | 9 | 23 | 0.1 | 0.2 | 0 | 0 |
| external_file_name | 14 | 28 | 0.2 | 1.7 | 2 | 0 |
| join | 15 | 53 | 0.6 | 5.2 | 2 | 1 |
| remove_newline | 25 | 105 | 0.6 | 4.6 | 0 | 0 |
| null_terminate | 9 | 23 | 0.1 | 0.2 | 2 | 0 |

# Results (EADS/RTC_Si)

| Proc | line | coreC line | time (sec) | space (Mb) | errors | FA |
|---|---|---|---|---|---|---|
| FiltrerCarNonImp | 19 | 34 | 1.6 | 0.5 | 0 | 0 |
| SkipLine | 12 | 42 | 0.8 | 1.9 | 0 | 0 |
| StoreIntInBuffer | 37 | 134 | 7.9 | 21 | 0 | 0 |

# Conclusions

- Static checking for string errors is feasible!
  - Can show the absence of string errors in complicated string manipulation procedures
- Identified rare bugs
- Techniques used
  - Modular analysis (assume/guarantee)
  - Pointer analysis
  - Integer analysis
- Open questions
  - Can this be fully automated?

# Related Work

- Runtime
  - Safe-C [PLDI'94]
  - Purify
  - Bound-checking

  …
- Static+ Runtime
  - CCured [POPL'02]

# Related Work

- Static
  - Wagner et. al. [NDSS’00]
  - LCLint’s extension [USENIX’01]
  - Dor, Rodeh and Sagiv [SAS’01]

# Recap

- Verifying absence of buffer overflows
- requires
  - relatively precise numerical domain
  - combined with points-to information
  - sometimes user annotations
- points-to analysis
- reduction to an integer program