

PREfix

Reading: *A Static Analyzer for Finding Dynamic Programming Errors*

17-654/17-765
Analysis of Software Artifacts
Jonathan Aldrich

Lecture Objectives

- Analyze Microsoft's PREfix as a practical example of effective static analysis
- Big Ideas
 - Symbolic execution
 - Path sensitivity
 - Interprocedural analysis

Find the Bugs!

```
char *f(int size) {
    char * result;
    if (size > 0)
        result = (char *)malloc(size);
    if (size == 1)
        return NULL;
    result[0] = 0;
    // result may be uninitialized

    return result;
}
```

Find the Bugs!

```
char *f(int size) {
    char * result;
    if (size > 0)
        result = (char *)malloc(size);
    if (size == 1)
        return NULL;
    result[0] = 0;
    // memory leak
    // result may be uninitialized
    // malloc may have failed

    return result;
}
```

Motivation

- Finding programming errors
 - invalid pointers
 - storage allocation errors
 - uninitialized memory
 - improper operations on resources

Can't we just test?

- 90% of errors involve interactions of multiple functions
 - Is this why the original developer didn't find them?
- Occur in unusual or error conditions
 - Often hard to exercise with testing

Challenges for Analysis

- **False Negatives**
 - Looking only in one function and miss errors across functions
- **False Positives**
 - Reporting errors that can't really occur
- **Engineering effort** (e.g. ESC/Java)
 - Requiring extensive program specifications
- **Execution overhead**
 - Monitoring program may be impractical
 - Only as good as your test suite

Goals of PREFIX

- **Handle hard aspects of C-like languages**
 - Pointers, arrays, unions, libraries, casts...
- **Don't require user annotations**
 - Build on language semantics
- **Avoid false positives**
 - Use path-sensitive analysis
- **Give the user good feedback**
 - Why might an error occur? Show the user an example execution

PREfix Analysis

- Explore paths through function
- For each path:
 - Symbolically execute path
 - Determine facts true along the path
 - Compute a guard
 - What must be true for the path to be taken
 - Compute constraints
 - Preconditions for successful execution of path
 - Compute result
 - What is true of the return value?

PREfix: Analysis Example

(syntax slightly de-LISP-ified)

```
char *f(int size) {                               f (param size)
  char * ptr;                                     alternate 0
  if (size > 0)
    ptr=(char*)malloc(size);
  if (size == 1)
    return NULL;
  ptr[0] = 0;
  return ptr;
}
```

PREfix: Analysis Example

(syntax slightly de-LISP-ified)

```
char *f(int size) {
    char * ptr;
    if (size > 0)
        ptr=(char*)malloc(size);
    if (size == 1)
        return NULL;
    ptr[0] = 0;
    return ptr;
}

f (param size)
alternate 0
guard size <= 0
constraint initialized(size)
```

PREfix: Analysis Example

(syntax slightly de-LISP-ified)

```
char *f(int size) {
    char * ptr;
    if (size > 0)
        ptr=(char*)malloc(size);
    if (size == 1)
        return NULL;
    ptr[0] = 0;
    return ptr;
}

f (param size)
alternate 0
guard size <= 0
constraint initialized(size)
```

PREfix: Analysis Example

(syntax slightly de-LISP-ified)

```
char *f(int size) {
  char * ptr;
  if (size > 0)
    ptr=(char*)malloc(size);
  if (size == 1)
    return NULL;
  ptr[0] = 0;
  return ptr;
}
```

f (param size)
alternate 0
guard size <= 0
constraint initialized(size)
ARRAY ACCESS ERROR: ptr not initialized

PREfix: Analysis Example

(syntax slightly de-LISP-ified)

```
char *f(int size) {
  char * ptr;
  if (size > 0)
    ptr=(char*)malloc(size);
  if (size == 1)
    return NULL;
  ptr[0] = 0;
  return ptr;
}
```

f (param size)
alternate 0
guard size <= 0
constraint initialized(size)
ARRAY ACCESS ERROR: ptr not initialized
alternate 1
guard size > 0
constraint initialized(size)

PREfix: Analysis Example

(syntax slightly de-LISP-ified)

```
char *f(int size) {
  char * ptr;
  if (size > 0)
    ptr=(char*)malloc(size);
  if (size == 1)
    return NULL;
  ptr[0] = 0;
  return ptr;
}
```

```
f (param size)
alternate 0
  guard size <= 0
  constraint initialized(size)
  ARRAY ACCESS ERROR: ptr not initialized
alternate 1
  guard size > 0
  constraint initialized(size)
  fact ptr==memory_new(size)
```

PREfix: Analysis Example

(syntax slightly de-LISP-ified)

```
char *f(int size) {
  char * ptr;
  if (size > 0)
    ptr=(char*)malloc(size);
  if (size == 1)
    return NULL;
  ptr[0] = 0;
  return ptr;
}
```

```
f (param size)
alternate 0
  guard size <= 0
  constraint initialized(size)
  ARRAY ACCESS ERROR: ptr not initialized
alternate 1
  guard size == 1
  constraint initialized(size)
  fact ptr==memory_new(size)
```

PREfix: Analysis Example

(syntax slightly de-LISP-ified)

```
char *f(int size) {
  char * ptr;
  if (size > 0)
    ptr=(char*)malloc(size);
  if (size == 1)
    return NULL;
  ptr[0] = 0;
  return ptr;
}
```

```
f (param size)
alternate 0
  guard size <= 0
  constraint initialized(size)
  ARRAY ACCESS ERROR: ptr not initialized
alternate 1
  guard size == 1
  constraint initialized(size)
  fact ptr==memory_new(size)
  result return==NULL
```

PREfix: Analysis Example

(syntax slightly de-LISP-ified)

```
char *f(int size) {
  char * ptr;
  if (size > 0)
    ptr=(char*)malloc(size);
  if (size == 1)
    return NULL;
  ptr[0] = 0;
  return ptr;
}
```

```
f (param size)
alternate 0
  guard size <= 0
  constraint initialized(size)
  ARRAY ACCESS ERROR: ptr not initialized
alternate 1
  guard size == 1
  constraint initialized(size)
  fact ptr==memory_new(size)
  result return==NULL
  MEMORY LEAK ERROR:
  memory pointed to by ptr is not reachable
  through externally visible state
```

PREfix: Analysis Example

(syntax slightly de-LISP-ified)

```
char *f(int size) {
  char * ptr;
  if (size > 0)
    ptr=(char*)malloc(size);
  if (size == 1)
    return NULL;
  ptr[0] = 0;
  return ptr;
}

f (param size)
alternate 0
  guard size <= 0
  constraint initialized(size)
  ARRAY ACCESS ERROR: ptr not initialized
alternate 1
  guard size == 1
  constraint initialized(size)
  fact ptr==memory_new(size)
  result return==NULL
  MEMORY LEAK ERROR:
  memory pointed to by ptr is not reachable
  through externally visible state
alternate 2
  guard size > 0
  constraint initialized(size)
```

PREfix: Analysis Example

(syntax slightly de-LISP-ified)

```
char *f(int size) {
  char * ptr;
  if (size > 0)
    ptr=(char*)malloc(size);
  if (size == 1)
    return NULL;
  ptr[0] = 0;
  return ptr;
}

f (param size)
alternate 0
  guard size <= 0
  constraint initialized(size)
  ARRAY ACCESS ERROR: ptr not initialized
alternate 1
  guard size == 1
  constraint initialized(size)
  fact ptr==memory_new(size)
  result return==NULL
  MEMORY LEAK ERROR:
  memory pointed to by ptr is not reachable
  through externally visible state
alternate 2
  guard size > 0
  constraint initialized(size)
  fact ptr==NULL
```

PREfix: Analysis Example

(syntax slightly de-LISP-ified)

```
char *f(int size) {
  char * ptr;
  if (size > 0)
    ptr=(char*)malloc(size);
  if (size == 1)
    return NULL;
  ptr[0] = 0;
  return ptr;
}

f (param size)
alternate 0
  guard size <= 0
  constraint initialized(size)
  ARRAY ACCESS ERROR: ptr not initialized
alternate 1
  guard size == 1
  constraint initialized(size)
  fact ptr==memory_new(size)
  result return==NULL
  MEMORY LEAK ERROR:
  memory pointed to by ptr is not reachable
  through externally visible state
alternate 2
  guard size > 1
  constraint initialized(size)
  fact ptr==NULL
```

PREfix: Analysis Example

(syntax slightly de-LISP-ified)

```
char *f(int size) {
  char * ptr;
  if (size > 0)
    ptr=(char*)malloc(size);
  if (size == 1)
    return NULL;
  ptr[0] = 0;
  return ptr;
}

f (param size)
alternate 0
  guard size <= 0
  constraint initialized(size)
  ARRAY ACCESS ERROR: ptr not initialized
alternate 1
  guard size == 1
  constraint initialized(size)
  fact ptr==memory_new(size)
  result return==NULL
  MEMORY LEAK ERROR:
  memory pointed to by ptr is not reachable
  through externally visible state
alternate 2
  guard size > 1
  constraint initialized(size)
  fact ptr==NULL
  ARRAY ACCESS ERROR: ptr is NULL
```

PREfix: Analysis Example

(syntax slightly de-LISP-ified)

```
char *f(int size) {
  char * ptr;
  if (size > 0)
    ptr=(char*)malloc(size);
  if (size == 1)
    return NULL;
  ptr[0] = 0;
  return ptr;
}

f (param size)
alternate 0
  guard size <= 0
  constraint initialized(size)
  ARRAY ACCESS ERROR: ptr not initialized
alternate 1
  guard size == 1
  constraint initialized(size)
  fact ptr==memory_new(size)
  result return==NULL
  MEMORY LEAK ERROR:
  memory pointed to by ptr is not reachable
  through externally visible state
alternate 2
  guard size > 1
  constraint initialized(size)
  fact ptr==NULL
  ARRAY ACCESS ERROR: ptr is NULL
alternate 3
  guard size > 1
  constraint initialized(size)
```

PREfix: Analysis Example

(syntax slightly de-LISP-ified)

```
char *f(int size) {
  char * ptr;
  if (size > 0)
    ptr=(char*)malloc(size);
  if (size == 1)
    return NULL;
  ptr[0] = 0;
  return ptr;
}

f (param size)
alternate 0
  guard size <= 0
  constraint initialized(size)
  ARRAY ACCESS ERROR: ptr not initialized
alternate 1
  guard size == 1
  constraint initialized(size)
  fact ptr==memory_new(size)
  result return==NULL
  MEMORY LEAK ERROR:
  memory pointed to by ptr is not reachable
  through externally visible state
alternate 2
  guard size > 1
  constraint initialized(size)
  fact ptr==NULL
  ARRAY ACCESS ERROR: ptr is NULL
alternate 3
  guard size > 1
  constraint initialized(size)
  fact ptr==memory_new(size)
```

PREfix: Analysis Example

(syntax slightly de-LISP-ified)

```
char *f(int size) {
  char * ptr;
  if (size > 0)
    ptr=(char*)malloc(size);
  if (size == 1)
    return NULL;
  ptr[0] = 0;
  return ptr;
}

f (param size)
alternate 0
  guard size <= 0
  constraint initialized(size)
  ARRAY ACCESS ERROR: ptr not initialized
alternate 1
  guard size == 1
  constraint initialized(size)
  fact ptr==memory_new(size)
  result return==NULL
  MEMORY LEAK ERROR:
  memory pointed to by ptr is not reachable
  through externally visible state
alternate 2
  guard size > 1
  constraint initialized(size)
  fact ptr==NULL
  ARRAY ACCESS ERROR: ptr is NULL
alternate 3
  guard size > 1
  constraint initialized(size)
  fact ptr==memory_new(size)
```

PREfix: Analysis Example

(syntax slightly de-LISP-ified)

```
char *f(int size) {
  char * ptr;
  if (size > 0)
    ptr=(char*)malloc(size);
  if (size == 1)
    return NULL;
  ptr[0] = 0;
  return ptr;
}

f (param size)
alternate 0
  guard size <= 0
  constraint initialized(size)
  ARRAY ACCESS ERROR: ptr not initialized
alternate 1
  guard size == 1
  constraint initialized(size)
  fact ptr==memory_new(size)
  result return==NULL
  MEMORY LEAK ERROR:
  memory pointed to by ptr is not reachable
  through externally visible state
alternate 2
  guard size > 1
  constraint initialized(size)
  fact ptr==NULL
  ARRAY ACCESS ERROR: ptr is NULL
alternate 3
  guard size > 1
  constraint initialized(size)
  fact ptr==memory_new(size)
  fact ptr[0] == 0
```

PREfix: Analysis Example

(syntax slightly de-LISP-ified)

```
char *f(int size) {
  char * ptr;
  if (size > 0)
    ptr=(char*)malloc(size);
  if (size == 1)
    return NULL;
  ptr[0] = 0;
  return ptr;
}

f (param size)
alternate 0
  guard size <= 0
  constraint initialized(size)
  ARRAY ACCESS ERROR: ptr not initialized
alternate 1
  guard size == 1
  constraint initialized(size)
  fact ptr==memory_new(size)
  result return==NULL
  MEMORY LEAK ERROR:
  memory pointed to by ptr is not reachable
  through externally visible state
alternate 2
  guard size > 1
  constraint initialized(size)
  fact ptr==NULL
  ARRAY ACCESS ERROR: ptr is NULL
alternate 3
  guard size > 1
  constraint initialized(size)
  fact ptr==memory_new(size)
  fact ptr[0] == 0
  result return == memory_new(size) && return[0] == 0
```

PREfix: Analysis Example

(syntax slightly de-LISP-ified)

```
char *f(int size) {
  char * ptr;
  if (size > 0)
    ptr=(char*)malloc(size);
  if (size == 1)
    return NULL;
  ptr[0] = 0;
  return ptr;
}

f (param size)
alternate 0
  guard size <= 0
  constraint initialized(size)
  ARRAY ACCESS ERROR: ptr not initialized
alternate 1
  guard size == 1
  constraint initialized(size)
  fact ptr==memory_new(size)
  result return==NULL
  MEMORY LEAK ERROR:
  memory pointed to by ptr is not reachable
  through externally visible state
alternate 2
  guard size > 1
  constraint initialized(size)
  fact ptr==NULL
  ARRAY ACCESS ERROR: ptr is NULL
alternate 3
  guard size > 1
  constraint initialized(size)
  fact ptr==memory_new(size)
  fact ptr[0] == 0
  result return == memory_new(size) && return[0] == 0
alternate 4...
```

Big Ideas

- Symbolic execution
 - Explore a *subset* of possible program executions
 - May not find all errors, but still useful
 - Carefully constructed to cover more functionality than most testing strategies can
- Path sensitivity
 - Avoids reporting errors that occur on control-flow paths that can't really be taken
- Interprocedural analysis
 - Looks at how the behavior of a callee affects the caller

Motivation: Path Sensitivity

[z := 0]₁

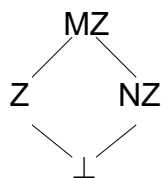
if [b]₂

 [z := 10]₃;

[x := 100]₄;

if [b]₅

 [x := x / z]₆;

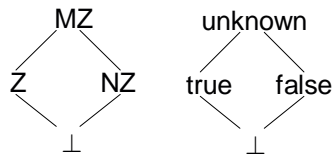


	<u>after pp</u>	<u>z</u>
0		MZ
1		Z
2		Z
3		NZ
4		MZ
5		MZ

- Does this code have a bug?
- What would zero analysis say?

Warning: possible divide by zero

Path Sensitive Analysis



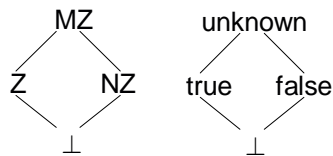
```

[z := 0]1
if [b]2
  [z := 10]3;
[x := 100]4;
if [b]5
  [x := x / z]6;
  
```

Analysis value after statement

0: [z→MZ, b→unknown]
 1: [z→Z, b→unknown]
 2: Split path into p and q on b
 2p: [z→Z, b→true], take branch
 3p: [z→NZ, b→true]
 4p: [z→NZ, x→NZ, b→true]
 5p: [z→NZ, x→NZ, b→true], take branch
 6p: [z→NZ, x→NZ, b→true]
No error

Path Sensitive Analysis



```

[z := 0]1
if [b]2
  [z := 10]3;
[x := 100]4;
if [b]5
  [x := x / z]6;
  
```

Analysis value after statement

0: [z→MZ, b→unknown]
 1: [z→Z, b→unknown]
 2: Split path into p and q on b
 2q: [z→Z, b→false], skip branch
 4q: [z→Z, x→NZ, b→false]
 5q: [z→Z, x→NZ, b→false], skip branch
No error

Path Sensitive Analysis

Analyzes each feasible program path separately

- Benefit
 - Increased precision from eliminating infeasible paths
- Cost
 - Exponential number of paths
- Loops
 - Infinite number of paths—cannot explore them all

Path Sensitivity: Addressing the Cost

- How does PREFIX deal with
 - Exponential path blowup?
 - Explore up to a fixed number of paths
 - Merge paths with identical results
 - Loops?
 - Explore up to a fixed number of iterations

What if you miss a path?

```
char *f(int size) {
    char * ptr;
    if (size > 0)
        ptr=(char*)malloc(size);
    if (size == 1)
        return NULL;
    ptr[0] = 0;
    return ptr;
}

f (param size)
alternate 0
  guard size <= 0
  constraint initialized(size)
  ARRAY ACCESS ERROR: ptr not initialized
alternate 1
  guard size == 1
  constraint initialized(size)
  fact ptr==memory_new(size)
  result return==NULL
  MEMORY LEAK ERROR:
  memory pointed to by ptr is not reachable
  through externally visible state
alternate 2
  guard size > 1
  constraint initialized(size)
  fact ptr==NULL
  ARRAY ACCESS ERROR: ptr is NULL
alternate 3
  guard size > 1
  constraint initialized(size)
  fact ptr==memory_new(size)
  fact ptr[0] == 0
  result return == memory_new(size) && return[0] == 0
alternate 4...
```

Soundness for PREFIX

- Exploring only some paths is unsound
 - Might miss bugs on paths not explored
- Sound alternatives
 - Explore a fixed set of paths/iterations
 - Merge all other paths together using dataflow analysis to reach a fixed point
 - Cost
 - May yield too many false positive error reports
 - PREFIX chooses unsoundness to avoid false positives

Motivation: Interprocedural Analysis

```
void exercise_deref() {  
    int v = 5;  
    int x = deref(&v);  
    int y = deref(NULL);  
    int z = deref((int *) 5);  
}
```

- Are there errors in this code?

Motivation: Interprocedural Analysis

```
void exercise_deref() {  
    int v = 5;  
    int x = deref(&v);  
    int y = deref(NULL);  
    int z = deref((int *) 5);  
}
```

- Are there errors in this code?
 - Depends on what the function does
 - Second call: error if dereference w/o NULL check
 - Third call: error if any dereference

Interprocedural Analysis

- ***Any analysis where the analysis results for a caller depend on the results for a callee, or vice versa***

Summaries

- Summarize what a function does
 - Maps arguments to results
 - May case-analyze on argument information
 - Simulateable
 - Given information about arguments, will yield:
 - Any errors
 - Information about results

PREfix: Building a Summary

(syntax slightly de-LISP-ified)

```
int deref(int *p) {  
    if (p == NULL)  
        return NULL;  
    return *p;  
}
```

- **Begin**
deref (param p)

PREfix: Building a Summary

(syntax slightly de-LISP-ified)

```
int deref(int *p) {  
    if (p == NULL)  
        return NULL;  
    return *p;  
}
```

- **Use of p**
deref (param p)
constraint initialized(p)

PREfix: Building a Summary

(syntax slightly de-LISP-ified)

```
int deref(int *p) {  
    if (p == NULL)  
        return NULL;  
    return *p;  
}
```

- Split path on value of p
deref (param p)
alternate return_0
guard p==NULL
constraint initialized(p)

PREfix: Building a Summary

(syntax slightly de-LISP-ified)

```
int deref(int *p) {  
    if (p == NULL)  
        return NULL;  
    return *p;  
}
```

- Return statement
deref (param p)
alternate return_0
guard p==NULL
constraint initialized(p)
result return==NULL

PREfix: Building a Summary

(syntax slightly de-LISP-ified)

```
int deref(int *p) {  
    if (p == NULL)  
        return NULL;  
    return *p;  
}
```

- Consider other path
- ```
deref (param p)
 alternate return_0
 guard p==NULL
 constraint initialized(p)
 result return==NULL
 alternate return_X
 guard p != NULL
 constraint initialized(p)
```

## PREfix: Building a Summary

*(syntax slightly de-LISP-ified)*

```
int deref(int *p) {
 if (p == NULL)
 return NULL;
 return *p;
}
```

- Dereference of p
- ```
deref (param p)  
  alternate return_0  
    guard p==NULL  
    constraint initialized(p)  
    result return==NULL  
  alternate return_X  
    guard p != NULL  
    constraint initialized(p)  
    constraint valid_ptr(p)
```

PREfix: Building a Summary

(syntax slightly de-LISP-ified)

```
int deref(int *p) {  
    if (p == NULL)  
        return NULL;  
    return *p;  
}
```

- Use of *p

```
deref (param p)  
  alternate return_0  
    guard p==NULL  
    constraint initialized(p)  
    result return==NULL  
  alternate return_X  
    guard p != NULL  
    constraint initialized(p)  
    constraint valid_ptr(p)  
    constraint initialized(*p)
```

PREfix: Building a Summary

(syntax slightly de-LISP-ified)

```
int deref(int *p) {  
    if (p == NULL)  
        return NULL;  
    return *p;  
}
```

- Return statement

```
deref (param p)  
  alternate return_0  
    guard p==NULL  
    constraint initialized(p)  
    result return==NULL  
  alternate return_X  
    guard p != NULL  
    constraint initialized(p)  
    constraint valid_ptr(p)  
    constraint initialized(*p)  
    result return==*p
```

PREfix: Using a Summary

(syntax slightly de-LISP-ified)

```
void exercise_deref(int v) {
  int v = 5;
  int x = deref(&v);
  int y = deref(NULL);
  int z = deref((int *) 5);
}
```

- **Begin**
exercise_deref

```
deref (param p)
  alternate return_0
    guard p==NULL
    constraint initialized(p)
    result return==NULL
  alternate return_X
    guard p != NULL
    constraint initialized(p)
    constraint valid_ptr(p)
    constraint initialized(*p)
    result return==*p
```

PREfix: Using a Summary

(syntax slightly de-LISP-ified)

```
void exercise_deref(int v) {
  int v = 5;
  int x = deref(&v);
  int y = deref(NULL);
  int z = deref((int *) 5);
}
```

- **Evaluate v = 5**
exercise_deref
fact initialized(v), v==5

```
deref (param p)
  alternate return_0
    guard p==NULL
    constraint initialized(p)
    result return==NULL
  alternate return_X
    guard p != NULL
    constraint initialized(p)
    constraint valid_ptr(p)
    constraint initialized(*p)
    result return==*p
```

PREfix: Using a Summary

(syntax slightly de-LISP-ified)

```
void exercise_deref(int v) {
  int v = 5;
  int x = deref(&v);
  int y = deref(NULL);
  int z = deref((int *) 5);
}
```

- Evaluate &v
- ```
exercise_deref
 fact initialized(v), v==5
 fact initialized(&v), valid_ptr(&v)
```

```
deref (param p)
 alternate return_0
 guard p==NULL
 constraint initialized(p)
 result return==NULL
 alternate return_X
 guard p != NULL
 constraint initialized(p)
 constraint valid_ptr(p)
 constraint initialized(*p)
 result return==*p
```

## PREfix: Using a Summary

*(syntax slightly de-LISP-ified)*

```
void exercise_deref(int v) {
 int v = 5;
 int x = deref(&v);
 int y = deref(NULL);
 int z = deref((int *) 5);
}
```

- Apply summary
- ```
exercise_deref
  fact initialized(v), v==5
  fact initialized(&v), valid_ptr(&v)
```

```
deref (param p)
  alternate return_0
    guard p==NULL
    constraint initialized(p)
    result return==NULL
  alternate return_X
    guard p != NULL
    constraint initialized(p)
    constraint valid_ptr(p)
    constraint initialized(*p)
    result return==*p
```

PREfix: Using a Summary

(syntax slightly de-LISP-ified)

```
void exercise_deref(int v) {
  int v = 5;
  int x = deref(&v);
  int y = deref(NULL);
  int z = deref((int *) 5);
}
```

- Apply summary

```
exercise_deref
  fact initialized(v), v==5
  fact initialized(&v), valid_ptr(&v)
```

```
deref (param p)
  alternate return_0
    guard p==NULL
    constraint initialized(p)
    result return==NULL
  alternate return_X
    guard p != NULL
    constraint initialized(p)
    constraint valid_ptr(p)
    constraint initialized(*p)
    result return==*p
```

- only return_X applies

PREfix: Using a Summary

(syntax slightly de-LISP-ified)

```
void exercise_deref(int v) {
  int v = 5;
  int x = deref(&v);
  int y = deref(NULL);
  int z = deref((int *) 5);
}
```

- Apply summary

```
exercise_deref
  fact initialized(v), v==5
  fact initialized(&v), valid_ptr(&v)
```

```
deref (param p)
  alternate return_0
    guard p==NULL
    constraint initialized(p)
    result return==NULL
  alternate return_X
    guard p != NULL
    constraint initialized(p)
    constraint valid_ptr(p)
    constraint initialized(*p)
    result return==*p
```

- only return_X applies
 - constraint initialized(&v) -- PASS

PREfix: Using a Summary

(syntax slightly de-LISP-ified)

```
void exercise_deref(int v) {
  int v = 5;
  int x = deref(&v);
  int y = deref(NULL);
  int z = deref((int *) 5);
}
```

- Apply summary

```
exercise_deref
  fact initialized(v), v==5
  fact initialized(&v), valid_ptr(&v)
```

```
deref (param p)
  alternate return_0
    guard p==NULL
    constraint initialized(p)
    result return==NULL
  alternate return_X
    guard p != NULL
    constraint initialized(p)
    constraint valid_ptr(p)
    constraint initialized(*p)
    result return==*p
```

- only return_X applies
- **constraint initialized(&v) – PASS**
- **constraint valid_ptr(&v) – PASS**

PREfix: Using a Summary

(syntax slightly de-LISP-ified)

```
void exercise_deref(int v) {
  int v = 5;
  int x = deref(&v);
  int y = deref(NULL);
  int z = deref((int *) 5);
}
```

- Apply summary

```
exercise_deref
  fact initialized(v), v==5
  fact initialized(&v), valid_ptr(&v)
```

```
deref (param p)
  alternate return_0
    guard p==NULL
    constraint initialized(p)
    result return==NULL
  alternate return_X
    guard p != NULL
    constraint initialized(p)
    constraint valid_ptr(p)
    constraint initialized(*p)
    result return==*p
```

- only return_X applies
- **constraint initialized(&v) – PASS**
- **constraint valid_ptr(&v) – PASS**
- **constraint initialized(*&v) – PASS**

PREfix: Using a Summary

(syntax slightly de-LISP-ified)

```
void exercise_deref(int v) {
  int v = 5;
  int x = deref(&v);
  int y = deref(NULL);
  int z = deref((int *) 5);
}
```

```
deref (param p)
  alternate return_0
    guard p==NULL
    constraint initialized(p)
    result return==NULL
  alternate return_X
    guard p != NULL
    constraint initialized(p)
    constraint valid_ptr(p)
    constraint initialized(*p)
    result return==*p
```

- Apply summary
exercise_deref
fact initialized(v), v==5
fact initialized(&v), valid_ptr(&v)
fact x==5
- only return_X applies
 - constraint initialized(&v) – PASS
 - constraint valid_ptr(&v) – PASS
 - constraint initialized(*&v) – PASS
 - apply result

PREfix: Using a Summary

(syntax slightly de-LISP-ified)

```
void exercise_deref(int v) {
  int v = 5;
  int x = deref(&v);
  int y = deref(NULL);
  int z = deref((int *) 5);
}
```

```
deref (param p)
  alternate return_0
    guard p==NULL
    constraint initialized(p)
    result return==NULL
  alternate return_X
    guard p != NULL
    constraint initialized(p)
    constraint valid_ptr(p)
    constraint initialized(*p)
    result return==*p
```

- Apply summary
exercise_deref
fact initialized(v), v==5
fact initialized(&v), valid_ptr(&v)
fact x==5

PREfix: Using a Summary

(syntax slightly de-LISP-ified)

```
void exercise_deref(int v) {
  int v = 5;
  int x = deref(&v);
  int y = deref(NULL);
  int z = deref((int *) 5);
}
```

- **Apply summary**

```
exercise_deref
  fact initialized(v), v==5
  fact initialized(&v), valid_ptr(&v)
  fact x==5
```

```
deref (param p)
  alternate return_0
  guard p==NULL
  constraint initialized(p)
  result return==NULL
  alternate return_X
  guard p != NULL
  constraint initialized(p)
  constraint valid_ptr(p)
  constraint initialized(*p)
  result return==*p
```

- **only return_0 applies**

PREfix: Using a Summary

(syntax slightly de-LISP-ified)

```
void exercise_deref(int v) {
  int v = 5;
  int x = deref(&v);
  int y = deref(NULL);
  int z = deref((int *) 5);
}
```

- **Apply summary**

```
exercise_deref
  fact initialized(v), v==5
  fact initialized(&v), valid_ptr(&v)
  fact x==5
```

```
deref (param p)
  alternate return_0
  guard p==NULL
  constraint initialized(p)
  result return==NULL
  alternate return_X
  guard p != NULL
  constraint initialized(p)
  constraint valid_ptr(p)
  constraint initialized(*p)
  result return==*p
```

- **only return_0 applies**
 - **constraint initialized(p) -- PASS**

PREfix: Using a Summary

(syntax slightly de-LISP-ified)

```
void exercise_deref(int v) {
  int v = 5;
  int x = deref(&v);
  int y = deref(NULL);
  int z = deref((int *) 5);
}
```

```
deref (param p)
  alternate return_0
    guard p==NULL
    constraint initialized(p)
    result return==NULL
  alternate return_X
    guard p != NULL
    constraint initialized(p)
    constraint valid_ptr(p)
    constraint initialized(*p)
    result return==*p
```

- Apply summary
- ```
exercise_deref
 fact initialized(v), v==5
 fact initialized(&v), valid_ptr(&v)
 fact x==5
 fact y==NULL
```
- only return\_0 applies
  - **constraint initialized(p) – PASS**
  - **apply result**

## PREfix: Using a Summary

(syntax slightly de-LISP-ified)

```
void exercise_deref(int v) {
 int v = 5;
 int x = deref(&v);
 int y = deref(NULL);
 int z = deref((int *) 5);
}
```

```
deref (param p)
 alternate return_0
 guard p==NULL
 constraint initialized(p)
 result return==NULL
 alternate return_X
 guard p != NULL
 constraint initialized(p)
 constraint valid_ptr(p)
 constraint initialized(*p)
 result return==*p
```

- Evaluate (int \*) 5
- ```
exercise_deref
  fact initialized(v), v==5
  fact initialized(&v), valid_ptr(&v)
  fact x==5
  fact y==NULL
  fact !valid_ptr((int *) 5), (int *) 5 !=
  NULL
```

PREfix: Using a Summary

(syntax slightly de-LISP-ified)

```
void exercise_deref(int v) {
  int v = 5;
  int x = deref(&v);
  int y = deref(NULL);
  int z = deref((int *) 5);
}
```

```
deref (param p)
  alternate return_0
    guard p==NULL
    constraint initialized(p)
    result return==NULL
  alternate return_X
    guard p != NULL
    constraint initialized(p)
    constraint valid_ptr(p)
    constraint initialized(*p)
    result return==*p
```

- Apply summary

```
exercise_deref
  fact initialized(v), v==5
  fact initialized(&v), valid_ptr(&v)
  fact x==5
  fact y==NULL
  fact !valid_ptr((int *) 5), (int *) 5 !=
  NULL
```

PREfix: Using a Summary

(syntax slightly de-LISP-ified)

```
void exercise_deref(int v) {
  int v = 5;
  int x = deref(&v);
  int y = deref(NULL);
  int z = deref((int *) 5);
}
```

```
deref (param p)
  alternate return_0
    guard p==NULL
    constraint initialized(p)
    result return==NULL
  alternate return_X
    guard p != NULL
    constraint initialized(p)
    constraint valid_ptr(p)
    constraint initialized(*p)
    result return==*p
```

- Apply summary

```
exercise_deref
  fact initialized(v), v==5
  fact initialized(&v), valid_ptr(&v)
  fact x==5
  fact y==NULL
  fact !valid_ptr((int *) 5), (int *) 5 !=
  NULL
```

- return_0 does not apply

PREfix: Using a Summary

(syntax slightly de-LISP-ified)

```
void exercise_deref(int v) {
  int v = 5;
  int x = deref(&v);
  int y = deref(NULL);
  int z = deref((int *) 5);
}
```

```
deref (param p)
  alternate return_0
    guard p==NULL
    constraint initialized(p)
    result return==NULL
  alternate return_X
    guard p != NULL
    constraint initialized(p)
    constraint valid_ptr(p)
    constraint initialized(*p)
    result return==*p
```

- Apply summary

```
exercise_deref
  fact initialized(v), v==5
  fact initialized(&v), valid_ptr(&v)
  fact x==5
  fact y==NULL
  fact !valid_ptr((int *) 5), (int *) 5 !=
  NULL
```

- return_0 does not apply
- return_X does apply

PREfix: Using a Summary

(syntax slightly de-LISP-ified)

```
void exercise_deref(int v) {
  int v = 5;
  int x = deref(&v);
  int y = deref(NULL);
  int z = deref((int *) 5);
}
```

```
deref (param p)
  alternate return_0
    guard p==NULL
    constraint initialized(p)
    result return==NULL
  alternate return_X
    guard p != NULL
    constraint initialized(p)
    constraint valid_ptr(p)
    constraint initialized(*p)
    result return==*p
```

- Apply summary

```
exercise_deref
  fact initialized(v), v==5
  fact initialized(&v), valid_ptr(&v)
  fact x==5
  fact y==NULL
  fact !valid_ptr((int *) 5), (int *) 5 !=
  NULL
```

- return_0 does not apply
- return_X does apply
 - constraint initialized((int *) 5) – PASS

PREfix: Using a Summary

(syntax slightly de-LISP-ified)

```
void exercise_deref(int v) {
  int v = 5;
  int x = deref(&v);
  int y = deref(NULL);
  int z = deref((int *) 5);
}
```

```
deref (param p)
  alternate return_0
    guard p==NULL
    constraint initialized(p)
    result return==NULL
  alternate return_X
    guard p != NULL
    constraint initialized(p)
    constraint valid_ptr(p)
    constraint initialized(*p)
    result return==*p
```

- Apply summary
- ```
exercise_deref
 fact initialized(v), v==5
 fact initialized(&v), valid_ptr(&v)
 fact x==5
 fact y==NULL
 fact !valid_ptr((int *) 5), (int *) 5 !=
 NULL
```
- return\_0 does not apply
  - return\_X does apply
    - constraint initialized((int \*) 5) – PASS
    - **constraint valid\_ptr((int \*) 5) – FAIL**
      - Generate error

## PREfix Scaleability

| Program  | Language | number of files | number of lines | PREfix parse time     | PREfix simulation time |
|----------|----------|-----------------|-----------------|-----------------------|------------------------|
| Mozilla  | C++      | 603             | 540613          | 2 hours<br>28 minutes | 8 hours<br>27 minutes  |
| Apache   | C        | 69              | 48393           | 6 minutes             | 9 minutes              |
| GDI Demo | C        | 9               | 2655            | 1 second              | 15 seconds             |

Table I: Performance on Sample Public Domain Software

- Analysis cost = 2x-5x build cost
  - Scales linearly
    - Probably due to fixed cutoff on number of paths

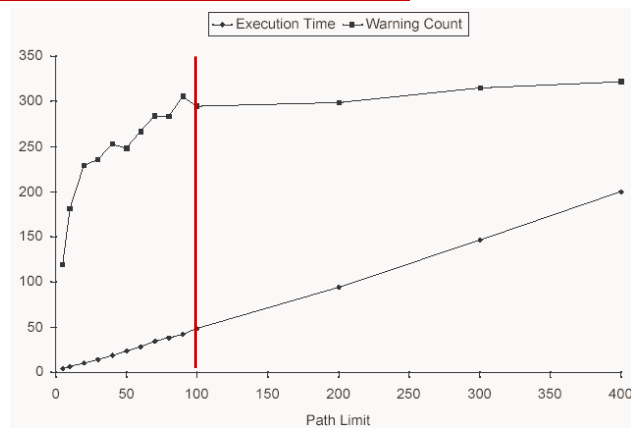
## Value of Interprocedural Analysis

| model set     | execution time (minutes) | statement coverage | branch coverage | predicate coverage | total warning count | using uninit memory | NULL pointer deref | memory leak |
|---------------|--------------------------|--------------------|-----------------|--------------------|---------------------|---------------------|--------------------|-------------|
| none          | 12                       | 90.1%              | 87.8%           | 83.9%              | 15                  | 2                   | 11                 | 0           |
| system        | 13                       | 88.9%              | 86.3%           | 82.1%              | 25                  | 6                   | 12                 | 7           |
| system & auto | 23                       | 73.1%              | 73.1%           | 68.6%              | 248                 | 110                 | 24                 | 124         |

Table III: Relationships between Available Models, Coverage, Execution Time, and Defects Reported

- 90% of errors require models (summaries)

## You don't need every path



- Get most of the warnings with 100 paths

## Empirical Observations

---

- PREFIX finds errors off the main code paths
  - Main-path errors caught by careful coding and testing
- UI is essential
  - Text output is hard to read
  - Need tool to visualize paths, sort defect reports
- Noise warnings
  - Real errors that users don't care about
    - E.g., memory leaks during catastrophic shutdown

## PREfix Summary

---

- PREFIX: Great tool to find errors
  - Can't guarantee that it finds them all
    - Role for other tools
  - Complements testing by analyzing uncommon paths
  - Focuses on low-level errors, not logic/functionality errors
    - Role for functional testing
- Huge impact
  - Used widely within Microsoft
  - Lightweight version is part of new Visual Studio

## Further Reading

---

- William R. Bush, Jonathan D. Pincus, and David J. Saelaff. **A Static Analyzer for Finding Dynamic Programming Errors.** *Software—Practice and Experience*, 30:775-802, 2000.