Vulnerability Prevention - Checked C

Holistic Software Security

Aravind Machiry

Safe C Dialects

- Add new types/features to unsafe languages: C/C++.
- Developers can write/convert the existing code by using these new features.
- Small burden to developers but low performance penalty.

Checked C

- Safe-dialect of C:
  - Fast and backward compatible
  - `_Ptr<type>`
  - `_Array_ptr<type>:count(expr)`
  - `_Nt_array_ptr<type>:count(expr)`

Writing Checked C Code

```
unsigned dlen;
int *darr;
int sarr[10];
char *r = "hello";
```

unsigned dlen;

_Array_ptr*<**int**>** darr: ***count(dlen);

int sarr **_Checked[10]**;

***_Nt_array_ptr*<**char**>** r : **count(6)** = "hello";

Minimizes Runtime Checks

```
unsigned dlen;
_Array_ptr<int> darr: count(dlen);
int sarr[10] _Checked;
_Nt_array_ptr<char> r : count(6) = "hello";

if (dlen > 4) {
    s = dlen - 4;
    ...darr[s];
}
```

Runtime check `if (s < dlen)` is optimized out.

Minimizes Runtime Checks

```
unsigned dlen;
_Array_ptr<int> darr: count(dlen);
int sarr[10] _Checked;
_Nt_array_ptr<char> r : count(6) = "hello";

if (dlen > 4) {
    s = dlen - 4;
    ...darr[s];
}
If (q < dlen) {
    ….darr[q]...
} else { err(“dynamic check failed”);}
```

Can We Easily Convert C to Checked C?

Refactoring the FreeBSD Kernel with Checked C

Junhan Duan,* Yudi Yang,* Jie Zhou, and John Criswell
Department of Computer Science
University of Rochester

It took two undergraduate students approximately three months to refactor the system calls, the network packet (mbuf) utility routines, and parts of the IP and UDP processing code. Our experiments show that using Checked C incurred no performance or code size overheads.

***Index Terms*—memory safety, safe C, FreeBSD**

Automated Conversion of C to Checked C

- Infer Checked C types for pointers
  - Type Inference
- Infer bounds for array pointers
  - Relationships among program variables
- Rewrite the code with Checked C types

Inferring Checked C types

- Type Inference
  - Standard Type Inference does not work

Conversion Example

```
int main(int argc, char **argv) {
  int *b, *c;
  struct f ob[5];
  char *str = argv[0];
  b = &global;
  baz(b, str, ob);
  c = b;
  printf("%d", strlen(str));
  return 0;
}
```

```
int *efunc();
int global;
int baz(int *w, char *s, struct f *p) {
   …
   w = (int*)0xdeadbeef;
   *s = p->ch;
   return 0;
}
```

```
int bar() {
   int *a = efunc();
   char *s = a;
   baz(a, s, NULL);
   return 0;
}
```

Conversion Example

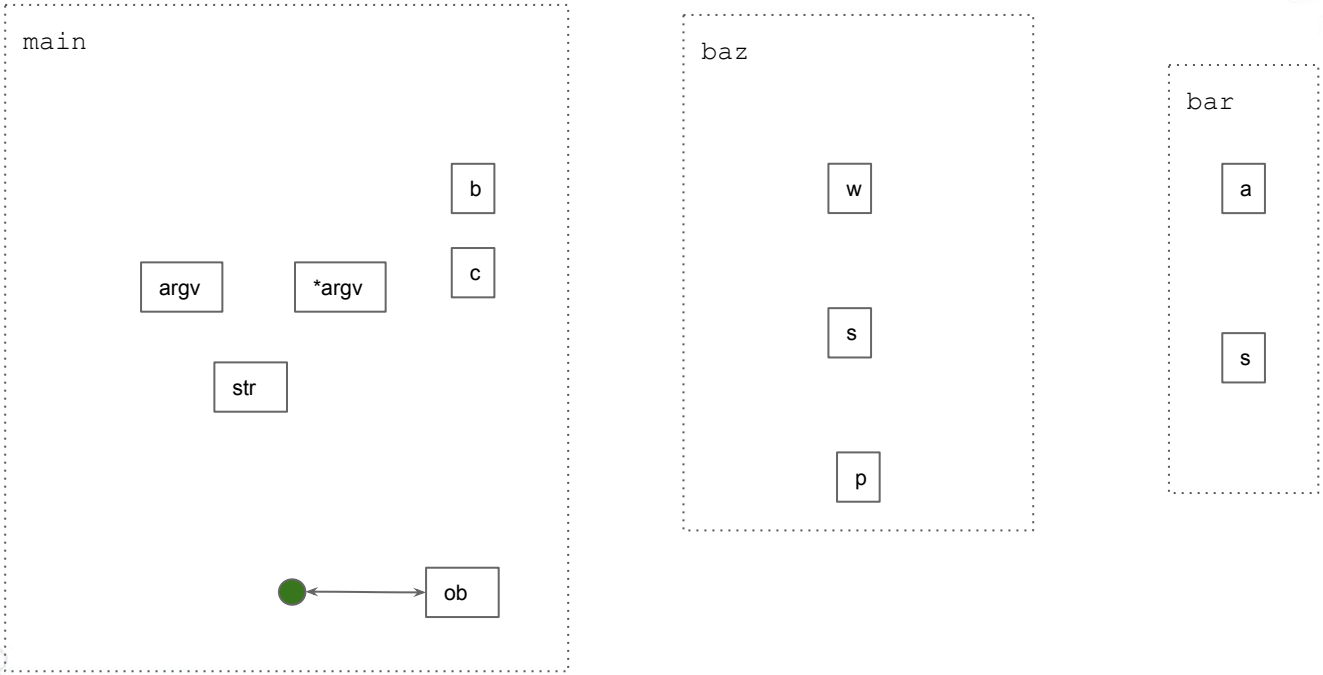
```
int main(int argc, char **argv) {
  int *b, *c;
  struct f ob[5];
  char *str = argv[0];
  b = &global;
  baz(b, str, ob);
  c = b;
  printf("%d", strlen(str));
  return 0;
}
```

```
int *efunc();
int global;
int baz(int *w, char *s, struct f *p) {
   …
   w = (int*)0xdeadbeef;
   *s = p->ch;
   return 0;
}
```

```
int bar() {
   int *a = efunc();
   char *s = a;
   baz(a, s, NULL);
   return 0;
}
```

WILD <: _Nt_array_ptr <: _Array_ptr <: _Ptr

Constraint Graph



Conversion Example

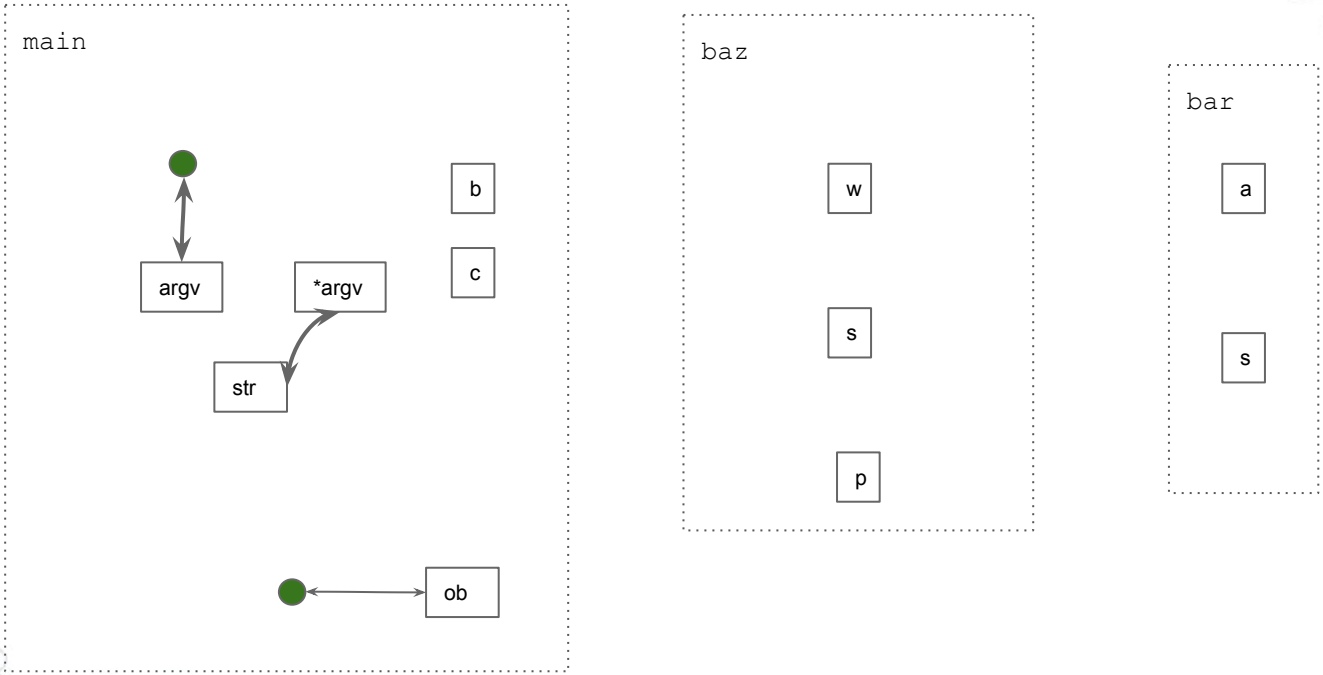
```
int main(int argc, char **argv) {
  int *b, *c;
  struct f ob[5];
  char *str = argv[0];
  b = &global;
  baz(b, str, ob);
  c = b;
  printf("%d", strlen(str));
  return 0;
}
```

```
int *efunc();
int global;
int baz(int *w, char *s, struct f *p) {
   …
   w = (int*)0xdeadbeef;
   *s = p->ch;
   return 0;
}
```

```
int bar() {
   int *a = efunc();
   char *s = a;
   baz(a, s, NULL);
   return 0;
}
```

WILD <: _Nt_array_ptr <: _Array_ptr <: _Ptr

Constraint Graph



Conversion Example

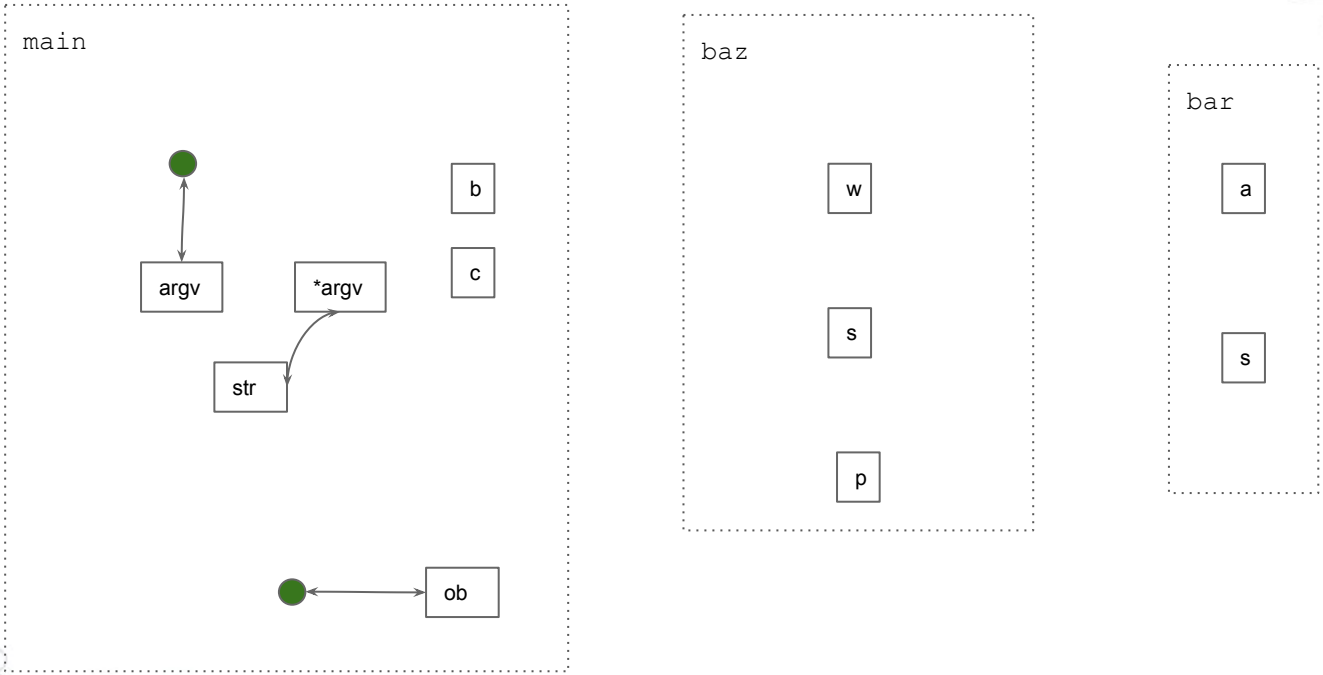
```
int main(int argc, char **argv) {
  int *b, *c;
  struct f ob[5];
  char *str = argv[0];
  b = &global;
  baz(b, str, ob);
  c = b;
  printf("%d", strlen(str));
  return 0;
}
```

```
int *efunc();
int global;
int baz(int *w, char *s, struct f *p) {
   …
   w = (int*)0xdeadbeef;
   *s = p->ch;
   return 0;
}
```

```
int bar() {
   int *a = efunc();
   char *s = a;
   baz(a, s, NULL);
   return 0;
}
```

WILD <: _Nt_array_ptr <: _Array_ptr <: _Ptr

Constraint Graph



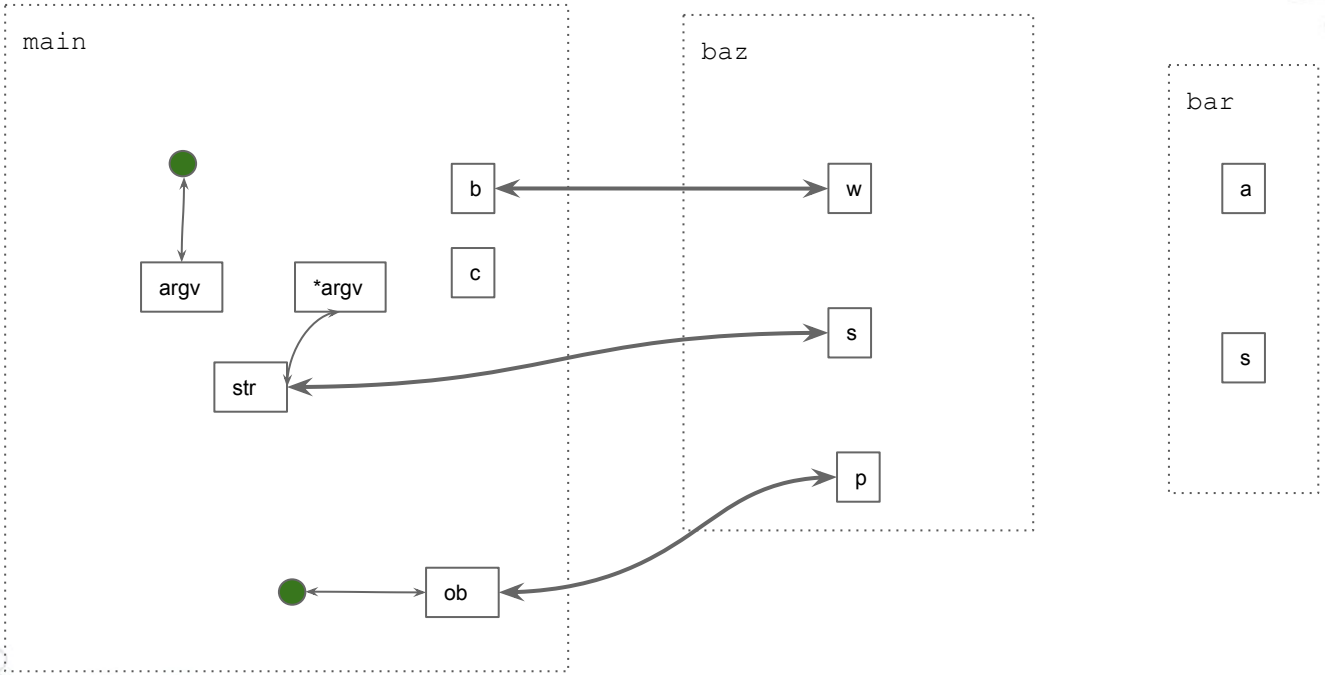
Conversion Example

```
int main(int argc, char **argv) {
  int *b, *c;
  struct f ob[5];
  char *str = argv[0];
  b = &global;
  baz(b, str, ob);
  c = b;
  printf("%d", strlen(str));
  return 0;
}
```

```
int *efunc();
int global;
int baz(int *w, char *s, struct f *p) {
   …
   w = (int*)0xdeadbeef;
   *s = p->ch;
   return 0;
}
```

```
int bar() {
   int *a = efunc();
   char *s = a;
   baz(a, s, NULL);
   return 0;
}
```

Constraint Graph



Conversion Example

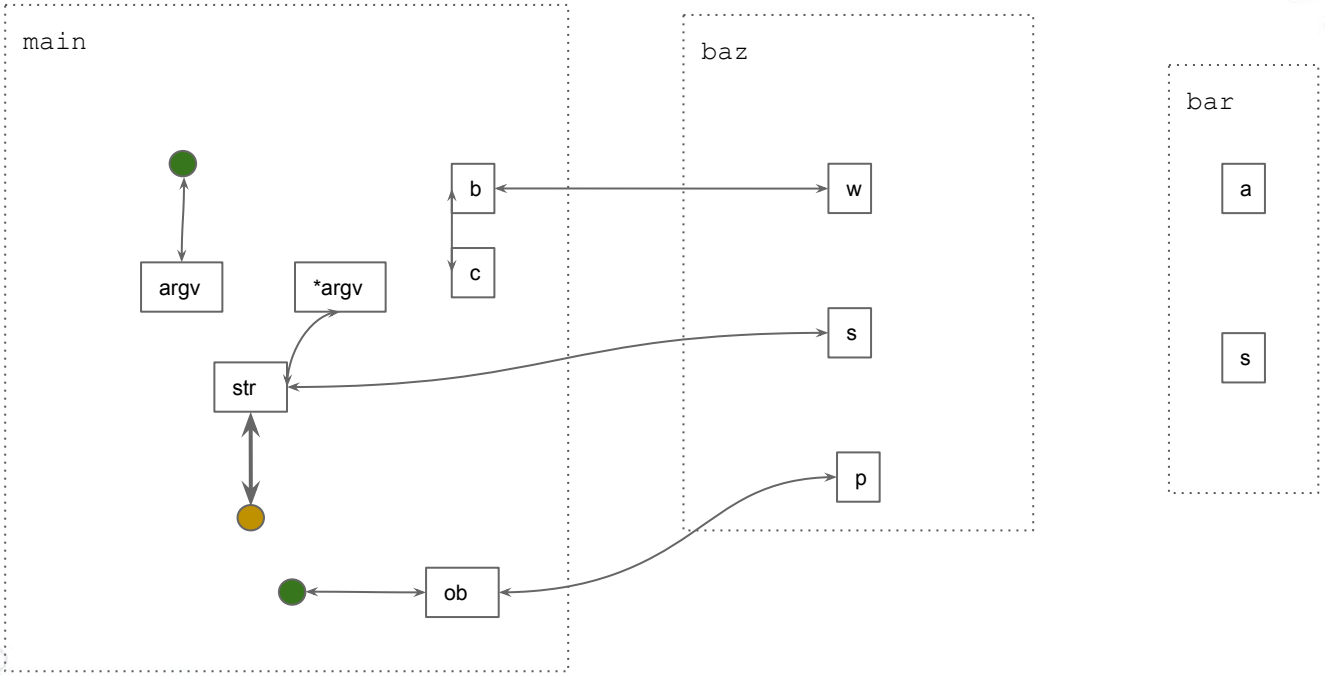
```
int main(int argc, char **argv) {
  int *b, *c;
  struct f ob[5];
  char *str = argv[0];
  b = &global;
  baz(b, str, ob);
  c = b;
  printf("%d", strlen(str));
  return 0;
}
```

```
int *efunc();
int global;
int baz(int *w, char *s, struct f *p) {
   …
   w = (int*)0xdeadbeef;
   *s = p->ch;
   return 0;
}
```

```
int bar() {
   int *a = efunc();
   char *s = a;
   baz(a, s, NULL);
   return 0;
}
```

WILD <: _Nt_array_ptr <: _Array_ptr <: _Ptr

Constraint Graph



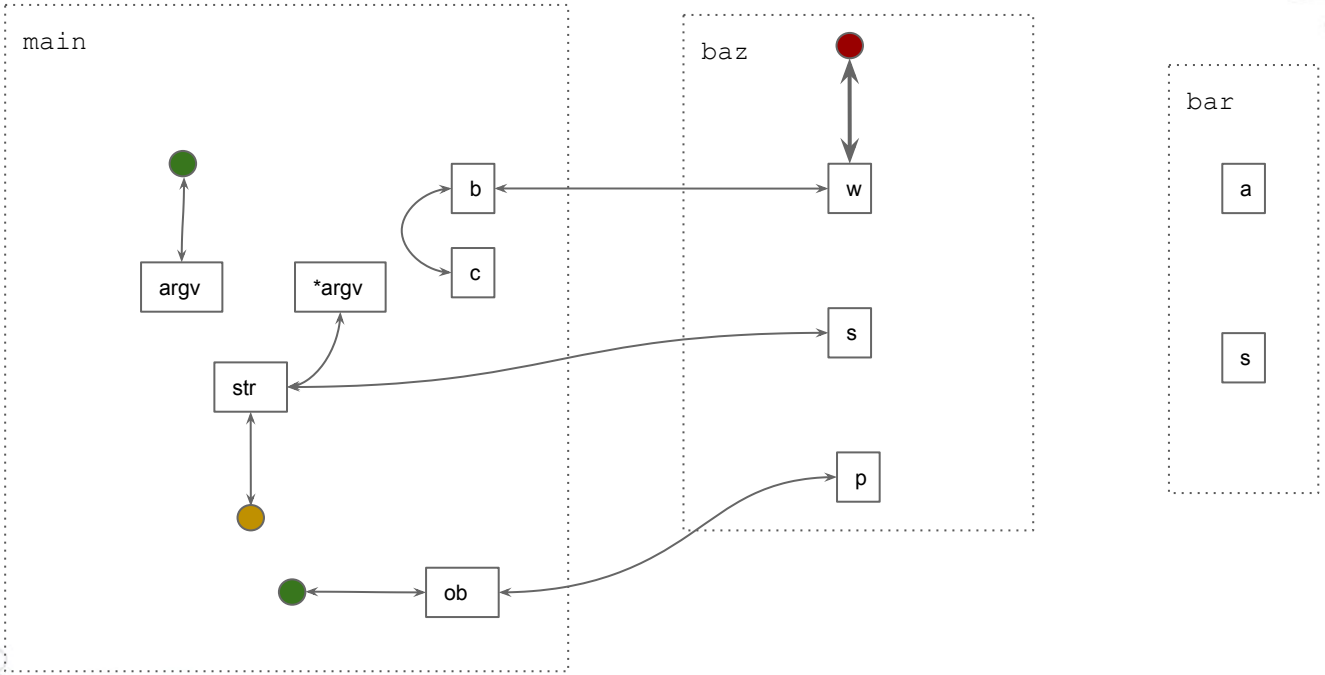
Conversion Example

```
int main(int argc, char **argv) {
  int *b, *c;
  struct f ob[5];
  char *str = argv[0];
  b = &global;
  baz(b, str, ob);
  c = b;
  printf("%d", strlen(str));
  return 0;
}
```

```
int *efunc();
int global;
int baz(int *w, char *s, struct f *p) {
   …
   w = (int*)0xdeadbeef;
   *s = p->ch;
   return 0;
}
```

```
int bar() {
   int *a = efunc();
   char *s = a;
   baz(a, s, NULL);
   return 0;
}
```

Constraint Graph



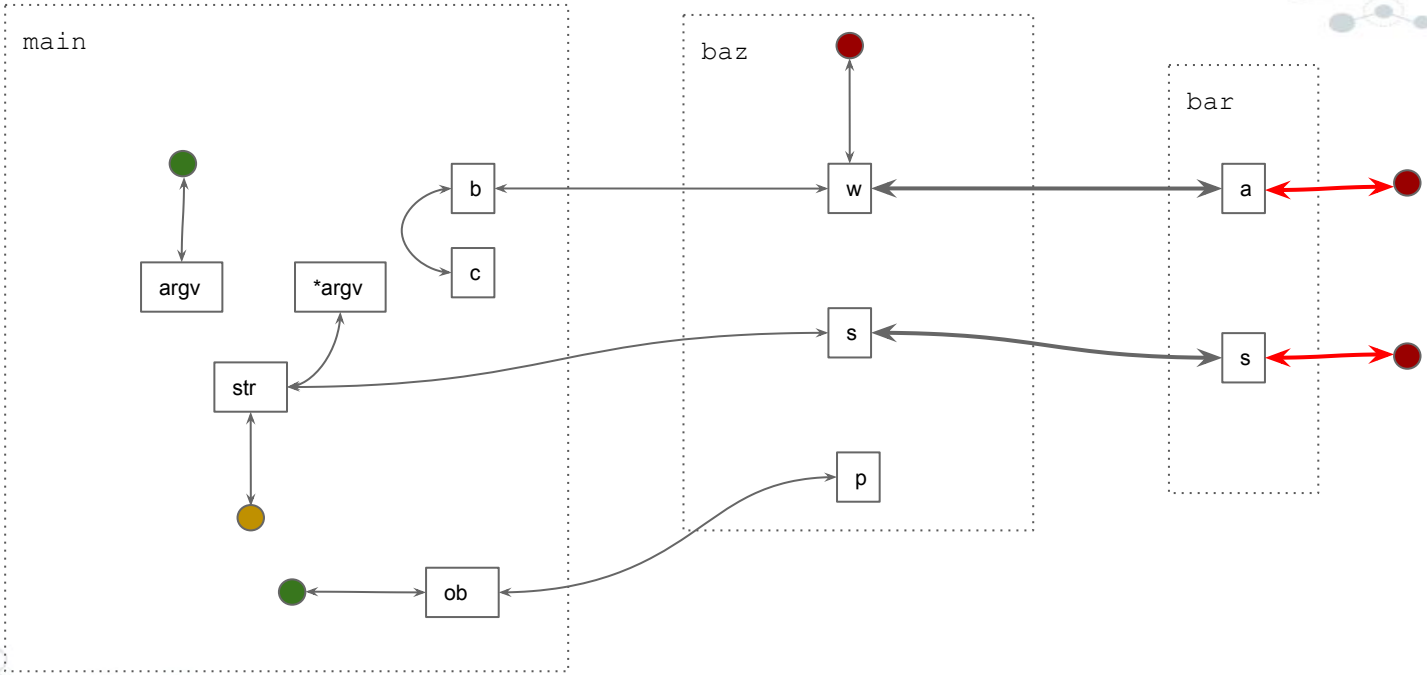
Conversion Example

```
int main(int argc, char **argv) {
  int *b, *c;
  struct f ob[5];
  char *str = argv[0];
  b = &global;
  baz(b, str, ob);
  c = b;
  printf("%d", strlen(str));
  return 0;
}
```

```
int *efunc();
int global;
int baz(int *w, char *s, struct f *p) {
   …
   w = (int*)0xdeadbeef;
   *s = p->ch;
   return 0;
}
```

```
int bar() {
   int *a = efunc();
   char *s = a;
   baz(a, s, NULL);
   return 0;
}
```

Constraint Graph

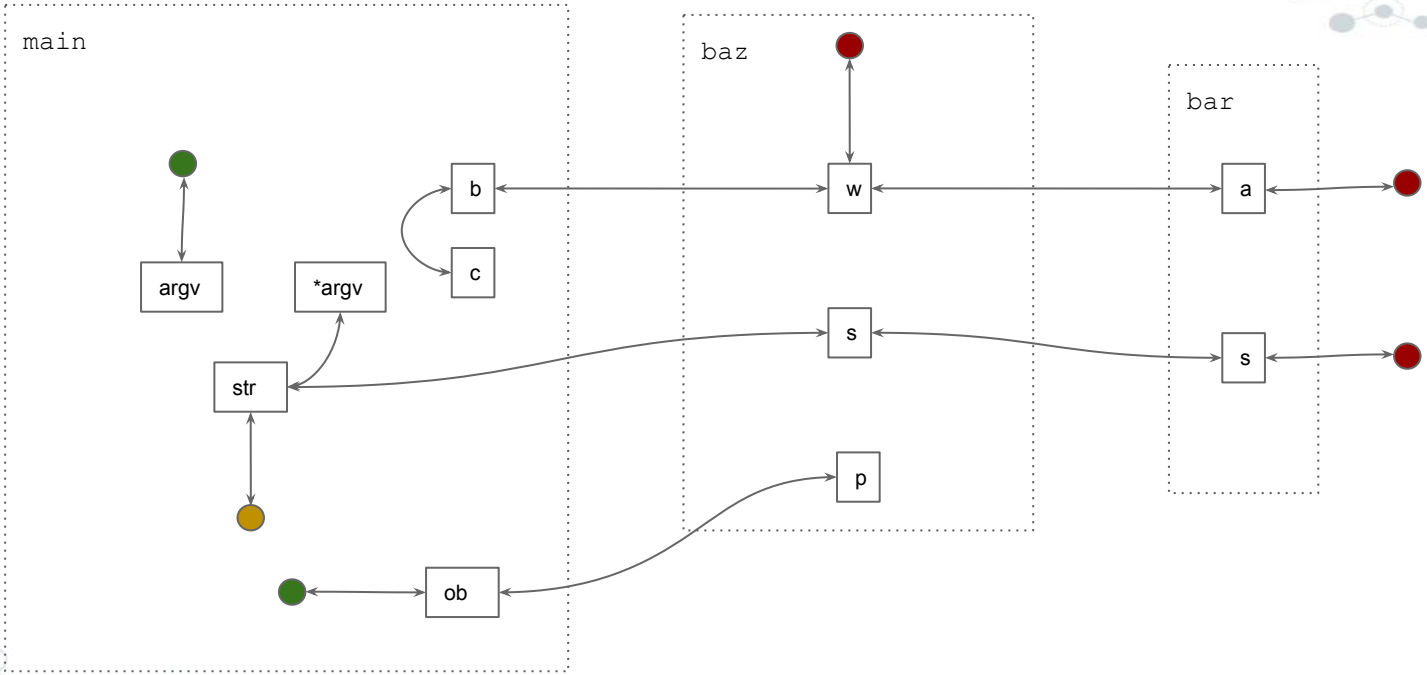


Constraint Solving

- Propagate Checked C types along the edges
- Join based on the type lattice

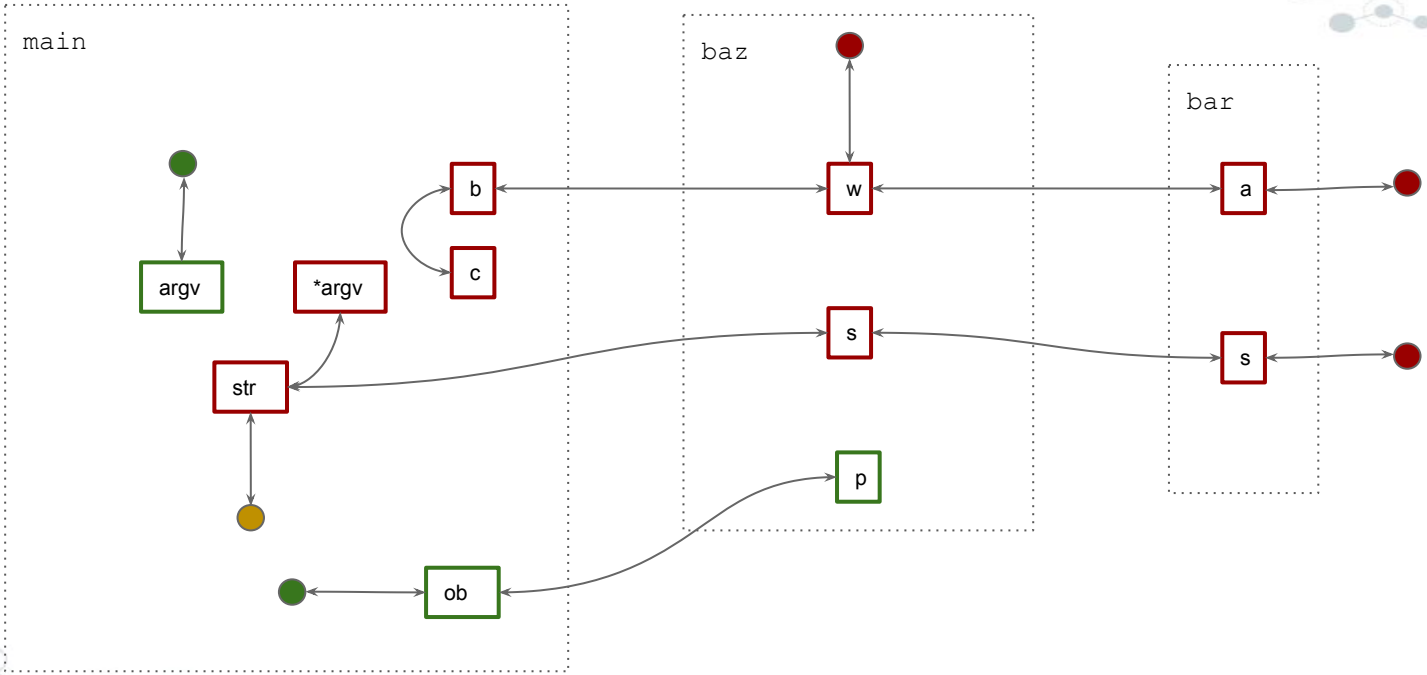
WILD **<:** _Nt_array_ptr **<:** _Array_ptr **<:** _Ptr

Constraint Graph



WILD <: _Nt_array_ptr <: _Array_ptr <: _Ptr

Constraint Graph Solution



Conversion with unification

```
int main(int argc,  _Array_ptr<char *> argv) {

  int *b, *c;

  struct f ob  _Checked[5];

  char *str = argv[0];

  b = &global;

  baz(b, str, ob);

  c = b;

  printf("%d", strlen(str));

  return 0;

}
```

```
int *efunc();

int global;

int baz(int *w, char *s,  _Array_ptr<struct f> p) {

   …

   w = (int*)0xdeadbeef;

   *s = p->ch;

   return 0;

}
```

```
int bar() {

   int *a = efunc();

   char *s = a;

   baz(a, s, NULL);

   return 0;

}
```

Unification Based Inference

- Simple and easy

- Imprecise
  - WILD pointers
  - Unnecessarily specific types

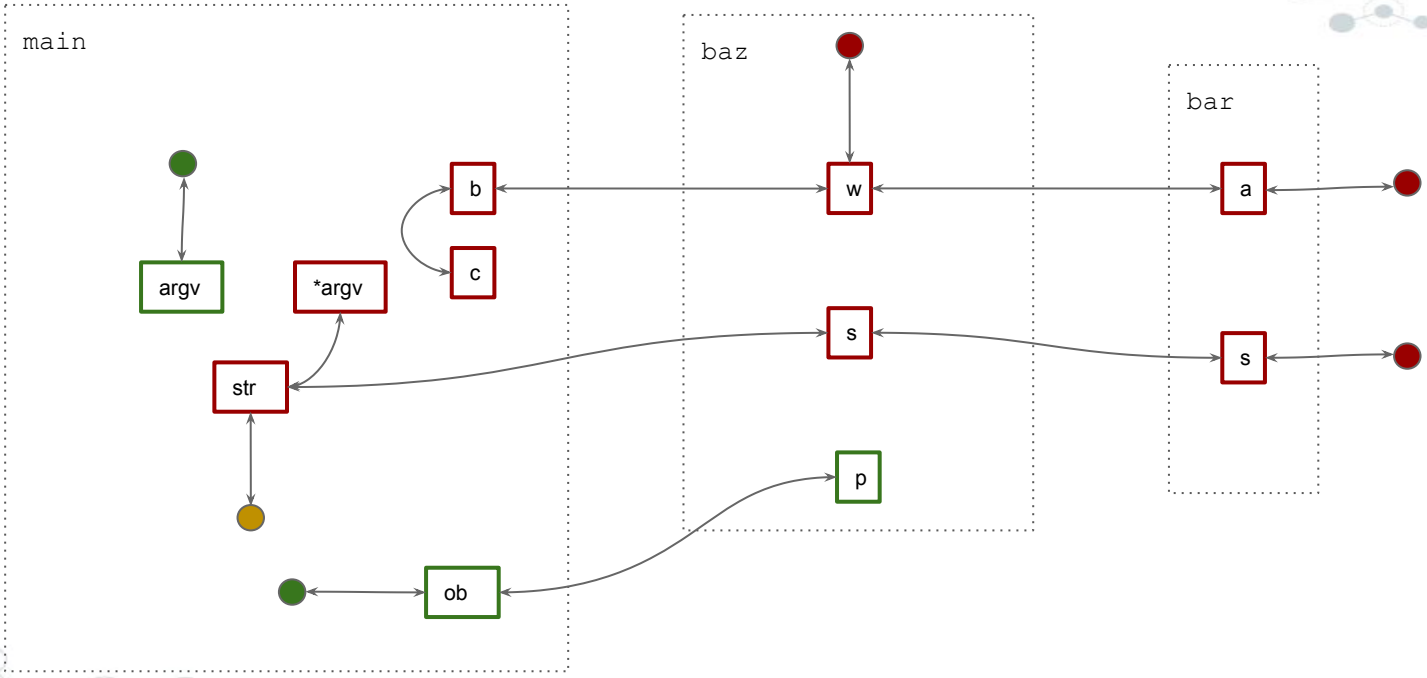
Conversion with unification

```
int main(int argc, _Array_ptr< char *> argv) {

  int *b, *c;

  struct f ob _Checked[5];

  char *str = argv[0];

  b = &global;

  baz(b, str, ob);

  c = b;

  printf("%d", strlen(str));

  return 0;

}
```

```
int *efunc();

int global;

int baz(int *w, char *s,  _Array_ptr<struct f> p) {

   …

   w = (int*)0xdeadbeef;

   *s = p->ch;

   return 0;

}
```

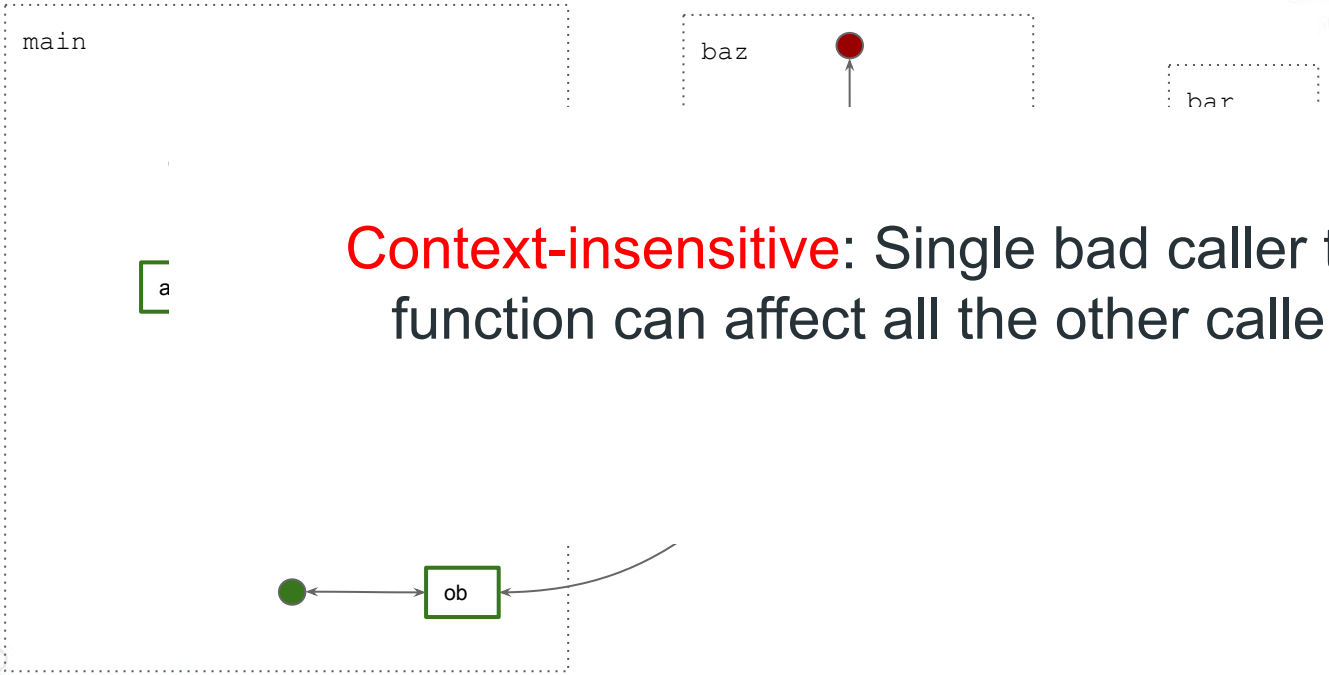
```
int bar() {

   int *a = efunc();

   char *s = a;

   baz(a, s, NULL);

   return 0;

}
```

Constraint Graph Solution



WILD <: _Nt_array_ptr <: _Array_ptr <: _Ptr

Constraint Graph Solution



Context-insensitive: Single bad caller to a function can affect all the other callers

Let us make the flow-directional across caller-calle boundaries

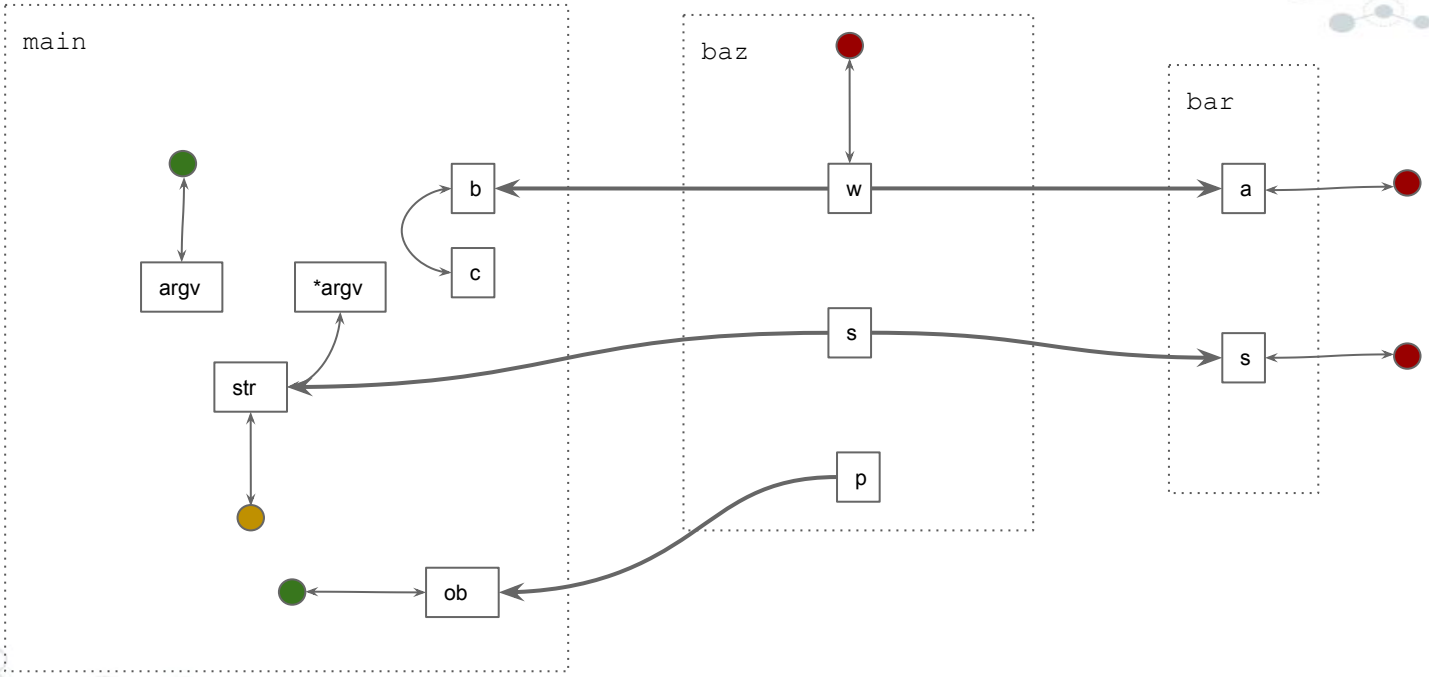
- Function Subtyping

Function Subtyping

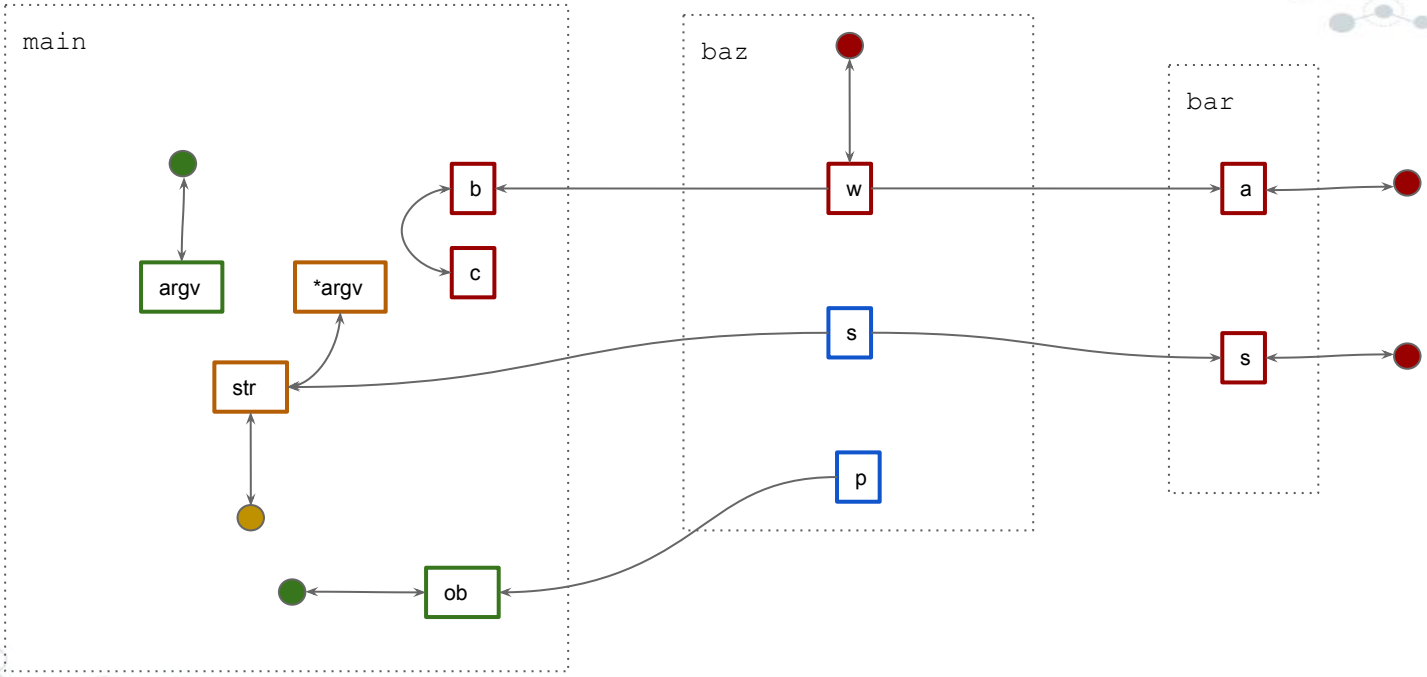
- Covariant: Argument type should be a subtype of corresponding parameter type

- [Reverse of above] Contra-variant: Return type should be a subtype of the type to which we assign

Constraint Graph - Directional Caller-Calle Edges



Constraint Graph Solution - Directional Caller-Calle Edges



Conversion with Directional Caller-Calle Edges

```
int main(int argc, _Array_ptr< _Nt_array_ptr<char>>
argv) {

  int *b, *c;

  struct f ob _Checked[5];

  _Nt_array_ptr<char> str = argv[0];

  b = &global;

  baz(b, str, ob);

  c = b;

  printf("%d", strlen(str));

  return 0;

}
```

```
int *efunc();

int global;

int baz(int *w, _Ptr<char> s, _Ptr<struct f> p) {

   …

   w = (int*)0xdeadbeef;

   *s = p->ch;

   return 0;

}
```

```
int bar() {

   int *a = efunc();

   char *s = a;

   baz(a, s, NULL);

   return 0;

}
```

Function Subtyping

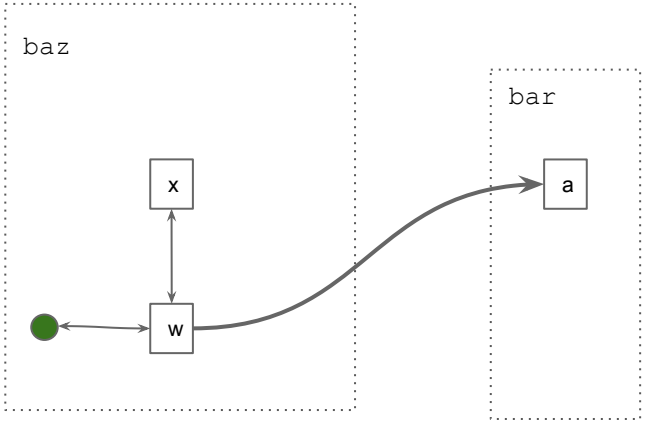
- Works for argument and parameter types
- Does not work for return types

Valid Return Type : Example

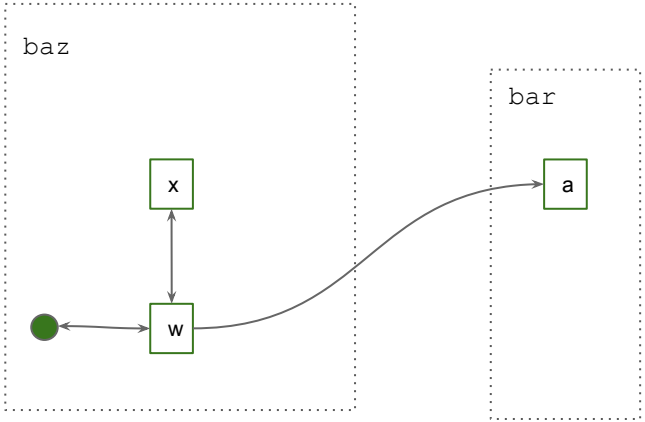
```
void baz() {
   int *x = NULL;
   int *w = bar();
   …
   x = w;
   …
   w[0] = 0;
}
```

```
int *bar() {
   int *a = malloc(5*sizeof(int));
   return a;
}
```

Constraint Graph - Valid Return Type



Constraint Graph Solution - Valid Return Type



Conversion: Valid Return Type

```
void baz() {

   _Array_ptr<int> x = NULL;

   _Array_ptr<int> w = bar();

   …

   x = w;

   …

   w[0] = 0;

}
```

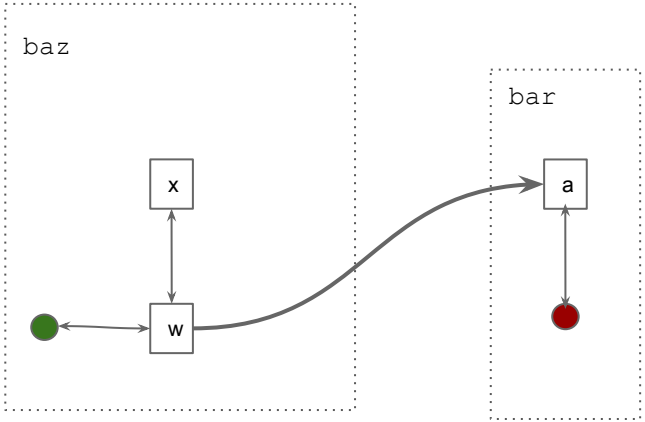
```
_Array_ptr<int> bar() {

   int *a = malloc(5*sizeof(int));

   return a;

}
```

Wrong Return Type : Example

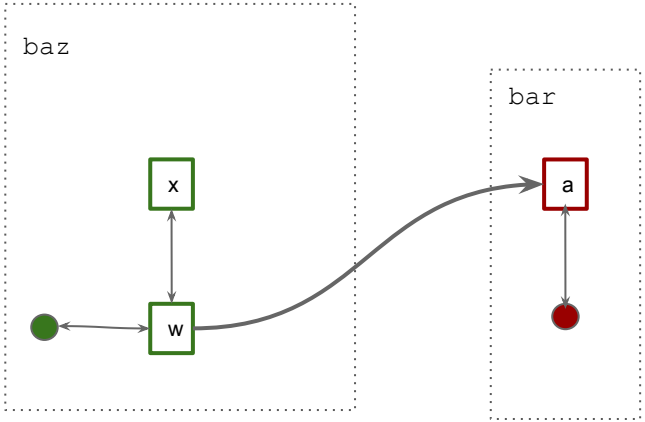
```
void baz() {
   int *x = NULL;
   int *w = bar();
   …
   x = w;
   …
   w[0] = 0;
}
```

```
int *bar() {
   int *a = (int *)0xdeadbeef;
   return a;
}
```

Constraint Graph - Wrong Return Type



Constraint Graph - Wrong Return Type



Conversion : Imprecise Return Type

```
void baz() {

    _Array_ptr<int> x = NULL;

    _Array_ptr<int> w = bar();

    …

    x = w;

    …

    w[0] = 0;

}
```

```
int *bar() {

    int *a = (int *)0xdeadbeef;

    return a;

}
```

Conversion : Imprecise Return Type

```
void baz() {

    _Array_ptr<int> x = NULL;

    _Array_ptr<int> w = bar();

    …

    x = w;

    …

    w[0] = 0;

}
```

```
int *bar() {

    int *a = (int *)0xdeadbeef;

    return a;

}
```

This is wrong! **We are using WILD pointer as a Checked type i.e., _Array_ptr**

Function Subtyping : Return Type

- If the return type is a Checked Type :
  - Return type should be subtype of its callers
    - Regular Contra-Variant

Valid Return Type : Example

```
void baz() {
   int *x = NULL;
   int *w = bar();
   …
   x = w;
   …
   w[0] = 0;
}
```

```
int *bar() {
   int *a = malloc(5*sizeof(int));
   return a;
}
```

Function Subtyping : Return Type

- If the return type is a Checked Type :
  - Return type should be subtype of its callers
    - Regular Contra-Variant

- **If the return type is WILD:**
  - **Then all the callers should use it as WILD**
    - **Reverse of Contra-Variant**

Wrong Return Type : Example

```
void baz() {
   int *x = NULL;
   int *w = bar();
   …
   x = w;
   …
   w[0] = 0;
}
```

```
int *bar() {
   int *a = (int *)0xdeadbeef;
   return a;
}
```

Chicken And Egg Problem

- The direction of propagation depends on the inferred type of the pointer
- To infer types we need to perform propagation

Split Types

- For each pointer, Create two type variables: (O, P)
  - O: Indicates whether the type is Checked or Wild
    - O : [C|W]

WILD **<:** Checked

  - P: Indicates the Checked pointer type
    - P : [Ptr |Arr | NtArr]

_Nt_array_ptr **<:** _Array_ptr **<:** _Ptr

Constraint Graph

- P edges are introduced according to regular subtyping rules

- O edges follow regular subtyping rules except for return where the direction will be reversed

Constraints Solving

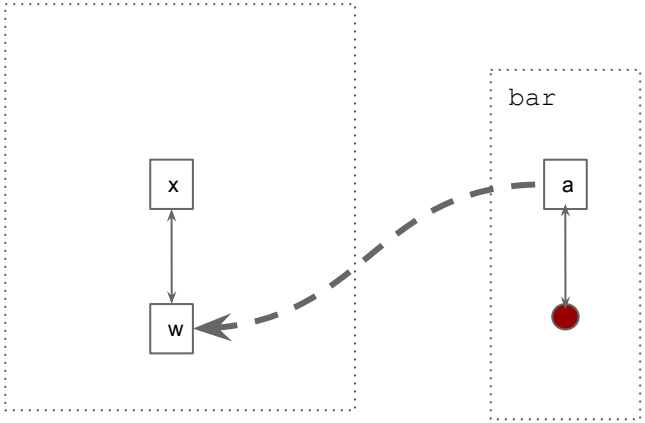
- Solve O graph and P graph independently
- Merge solutions: For all pointers whose O variable is Checked, fetch the corresponding solution for P variable as the final type

Return Type : Example

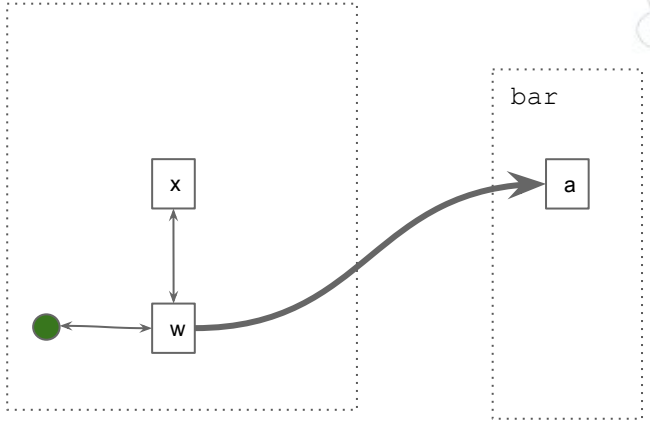
```
void baz() {
    int *x = NULL;
    int *w = bar();
    …
    x = w;
    …
    w[0] = 0;
}
```

```
int *bar() {
    int *a = (int *)0xdeadbeef;
    return a;
}
```

Constraint Graph - (O, P) Graphs

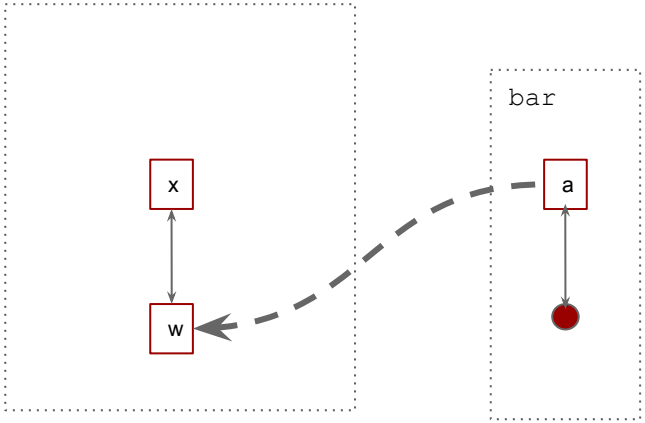


O Graph

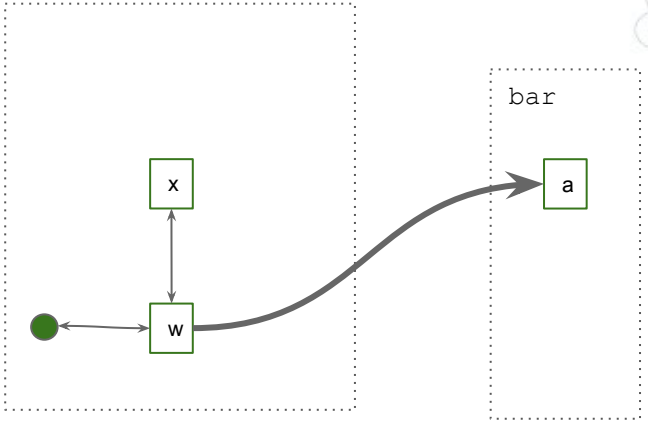


P Graph

Constraint Graph - (O, P) Solution

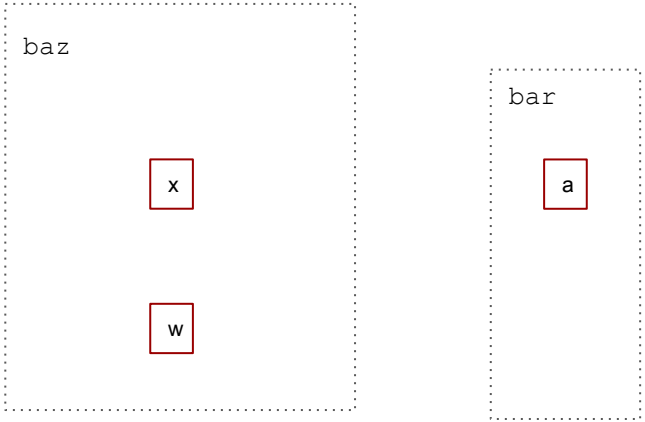


O Graph



P Graph

Constraint Graph - Merged Solution



Correctly Converted Return Type

```
void baz() {
    int *x = NULL;
    int *w = bar();
    …
    x = w;
    …
    w[0] = 0;
}
```

```
int *bar() {
    int *a = (int *)0xdeadbeef;
    return a;
}
```

Conversion: Valid Return Type

```
void baz() {

   _Array_ptr<int> x = NULL;

   _Array_ptr<int> w = bar();

   …

   x = w;

   …

   w[0] = 0;

}
```

```
_Array_ptr<int> bar() {

   int *a = malloc(5*sizeof(int));

   return a;

}
```

Inferring Array Bounds : Primer

```
// Static sized array
int arr _Checked[10]; // Bounds of arr are (arr, arr + 10 * sizeof(int))

// Dynamically allocated array
_Array_ptr<int> v : count(n) = malloc(n*sizeof(int));   // Bounds of v are (v,  v + n*sizeof(int))

// Constant array
_Nt_array_ptr<const char> s : byte_count(6) = “hello”; // Bounds of s are (s, s + 6)

struct hash_table {
   unsigned buckets;
   unsigned idx;
   // Bounds of entries are (entries, entries + buckets * sizeof(struct hash *))
   _Array_ptr<_Ptr<struct hash>> entries : count(buckets);
}
```

Array Bounds Inference

1. Infer Checked types

1. Insert seed bounds:
   a. malloc, built-in annotations, control-dependencies

1. Propagate the bounds information

Array Bounds Inference

```
struct foo {
  int *y;
  int l;
};
```

```
void bar(int *x, int c) {
  struct foo f = { x, c };
  memset(x, 1, c);
  x[0] = 0;
}
```

Array Bounds Inference : Type Inference

```
struct foo {

  _Array_ptr<int> y;

  int l;

};
```

```
void bar(_Array_ptr<int> x, int c) {

  struct foo f = { x, c };

  memset(x, 1, c);

  x[0] = 0;

}
```

Array Bounds Inference : Seed Bounds

```
struct foo {
  _Array_ptr<int> y;
  int l;
};
```

```
void bar(_Array_ptr<int> x, int c) {
  struct foo f = { x, c };
  memset(x, 1, c); // bounds of x are byte_bound(c)
  x[0] = 0;
}
```

Array Bounds Inference : Propagate Bounds

```
struct foo {
  _Array_ptr<int> y;
  int l;
};
```

```
void bar(_Array_ptr<int> x : byte_count(c), int c) {
  struct foo f = { x, c };
  memset(x, 1, c);
  x[0] = 0;
}
```

Array Bounds Inference : Propagate Bounds

```
struct foo {
  _Array_ptr<int> y;
  int l;
};
```

```
void bar(_Array_ptr<int> x : byte_count(c), int c) {
  struct foo f = { x, c };
  memset(x, 1, c);
  x[0] = 0;
}
```

Array Bounds Inference : Propagate Bounds

```
struct foo {
  _Array_ptr<int> y : byte_count(l);
  int l;
};
```

```
void bar(_Array_ptr<int> x : byte_count(c), int c) {
  struct foo f = { x, c };
  memset(x, 1, c);
  x[0] = 0;
}
```

Need For Interactivity

- Handling explicit but safe casting

Need For Interactivity

- Handling explicit but safe casting

```
struct vsf_sysutil_dir* vsf_sysutil_opendir(const char* p_dirname) {
  return (struct vsf_sysutil_dir*) opendir(p_dirname);
}
```

Need For Interactivity

- Handling explicit but safe casting
  - Example: 90 additional pointers can be converted to checked types with a 1 bit (yes/no) input from user

Preliminary Evaluation : Dataset

Program	Category	Size	Total Pointers (Tot)
vsftpd	FTP Server	17K	2,020
anagram	Pointer Benchmarks	352	72
ft		115	209
ks		609	79
yacr2		2,915	109
Total		**22K**	**2,486**

Preliminary Evaluation : Conversion

Program	Total Pointers (Tot)	Regular Pointers PTR	Array Pointers					Unconverted (WILD) (% over Tot)	Directly UnConverted (% over Tot)
			ARR	NTARR	Constant Size	Need Bounds (nb)	Inferred Bounds (% over nb)		
vsftpd	2,020	1,114	23	35	39	19	7 (36.84%)	848 (41.98%)	**189 (9.35%)**
anagram	72	23	21	2	15	7	1 (14.29%)	26 (36.11%)	**8 (11.11%)**
ft	209	115	8	0	6	2	0 (0%)	86 (41.15%)	**5 (2.39%)**
ks	79	15	8	5	11	4	0 (0%)	48 (63.16%)	**1 (1.32%)**
yacr2	109	12	82	0	77	5	2 (40%)	15 (13.76%)	**4 (3.67%)**
Total	**2,486**	**1,279**	**142**	**42**	**148**	**37**	**10 (27.03)**	**1,023 (41.15%)**	**207 (8.33%)**

Rewriting Challenges

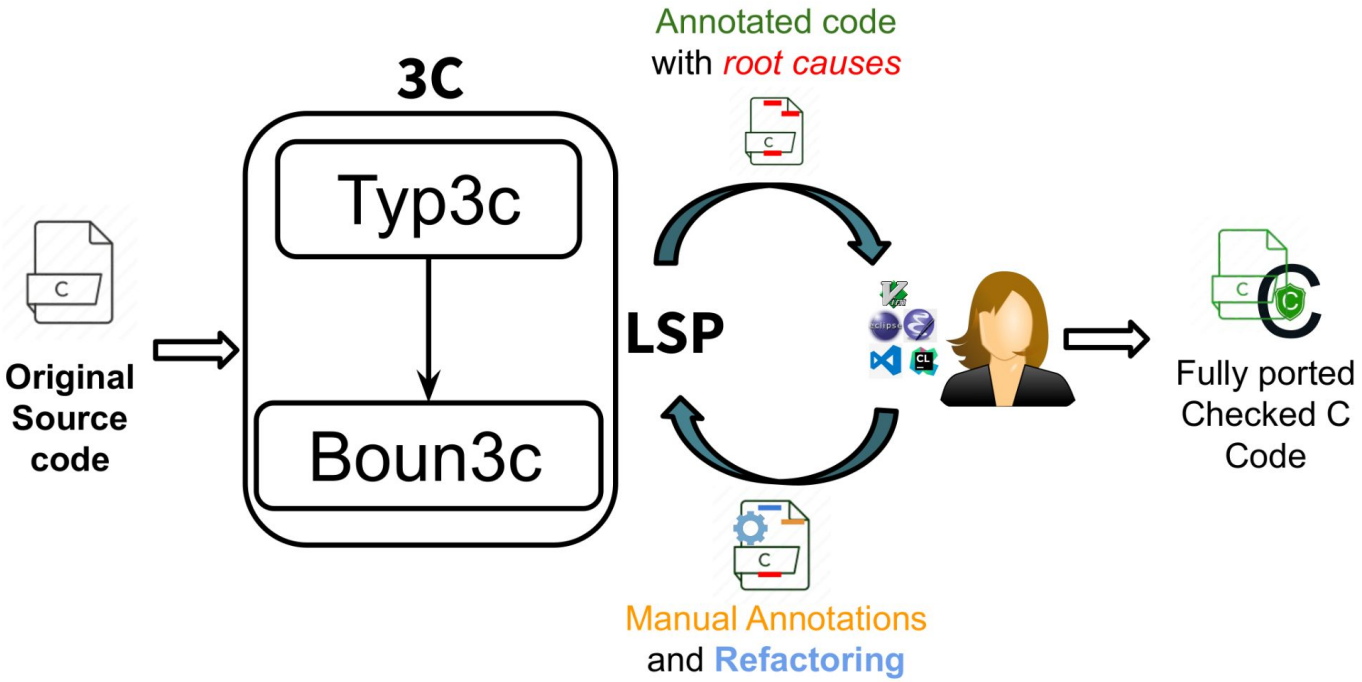
- Rewriting C code is hard
  - Typedefs
  - Preprocessor directives

Infeasibility of Automated Conversion

```
int resize_buf(char **buf, unsigned *sz) {
 char *newbuf = NULL;
 unsigned news = round_up(*sz, 64);
 newbuf = realloc(buf, news);
 *buf = newbuf;
 *sz = news;
 return newbuf != NULL;
}
```

```
typedef struct {
  array_ptr<char> buf : count(sz);
  unsigned sz;
} SIZEBUF;

int resize_buf(ptr<SIZEBUF> buf) _Checked {
  unsigned news = round_up(buf->sz, 64);
  array_ptr<char> newbuf : count(news) = NULL;
  newbuf = realloc<char>(buf->buf, news);
  buf->buf = newbuf;
  buf->sz = news;
  return newbuf != NULL;
}
// Refactor all callers of resize_buf to
// use the new caller.
```

3C : Interactive Conversion to Checked C



Status

- C to Checked C (3C) : well-mainted open source project
  https://github.com/correctcomputation/checkedc-clang

- Supported by Microsoft.