Memory Safety in Systems Languages

Major Area Exam

Michael Christensen

June 11, 2018

Committee:

Ben Hardekopf (🪑)   Tim Sherwood   Rich Wolski

Outline

Motivation

What is a System?

Infrastructure software upon which applications are built

What is a System?

Infrastructure software upon which applications are built

Operating Systems

- Process abstraction
- Multiplex physical hardware resources
- Partition and abstract **memory**

What is a System?

Infrastructure software upon which applications are built

Operating Systems

- Process abstraction
- Multiplex physical hardware resources
- Partition and abstract **memory**

Embedded Systems,

What is a System?

Infrastructure software upon which applications are built

Operating Systems

- Process abstraction
- Multiplex physical hardware resources
- Partition and abstract **memory**

Embedded Systems, Compilers,

What is a System?

Infrastructure software upon which applications are built

Operating Systems

- Process abstraction
- Multiplex physical hardware resources
- Partition and abstract **memory**

Embedded Systems, Compilers, Garbage Collectors,

What is a System?

Infrastructure software upon which applications are built

Operating Systems

- Process abstraction
- Multiplex physical hardware resources
- Partition and abstract **memory**

Embedded Systems, Compilers, Garbage Collectors, Device Drivers,

What is a System?

Infrastructure software upon which applications are built

Operating Systems

- Process abstraction
- Multiplex physical hardware resources
- Partition and abstract **memory**

Embedded Systems, Compilers, Garbage Collectors, Device Drivers, File Systems

Systems Languages

- Past systems languages: ALGOL, PL/I, Fortran, BCPL/B, C, Mesa/Cedar, Pascal/Modula-2/Oberon, C++, ...

Systems Languages

- Past systems languages: ALGOL, PL/I, Fortran, BCPL/B, C, Mesa/Cedar, Pascal/Modula-2/Oberon, C++, ...
- C: the de-facto standard

Systems Languages

- Past systems languages: ALGOL, PL/I, Fortran, BCPL/B, C, Mesa/Cedar, Pascal/Modula-2/Oberon, C++, ...
- C: the de-facto standard
  - Data structure representation control

Systems Languages

- Past systems languages: ALGOL, PL/I, Fortran, BCPL/B, C, Mesa/Cedar, Pascal/Modula-2/Oberon, C++, ...
- C: the de-facto standard
  - Data structure representation control
  - Memory management control

Systems Languages

- Past systems languages: ALGOL, PL/I, Fortran, BCPL/B, C, Mesa/Cedar, Pascal/Modula-2/Oberon, C++, ...
- C: the de-facto standard
  - Data structure representation control
  - Memory management control
  - Complete mutability via pointers

Systems Languages

- Past systems languages: ALGOL, PL/I, Fortran, BCPL/B, C, Mesa/Cedar, Pascal/Modula-2/Oberon, C++, ...
- C: the de-facto standard
  - Data structure representation control
  - Memory management control
  - Complete mutability via pointers
  - Performant

Systems Languages

- Past systems languages: ALGOL, PL/I, Fortran, BCPL/B, C, Mesa/Cedar, Pascal/Modula-2/Oberon, C++, ...
- C: the de-facto standard
  - Data structure representation control
  - Memory management control
  - Complete mutability via pointers
  - Performant
  - Legacy

Systems Languages

- Past systems languages: ALGOL, PL/I, Fortran, BCPL/B, C, Mesa/Cedar, Pascal/Modula-2/Oberon, C++, ...
- C: the de-facto standard
  - Data structure representation control
  - Memory management control
  - Complete mutability via pointers
  - Performant
  - Legacy
- C: the unsafe standard

Systems Languages

- Past systems languages: ALGOL, PL/I, Fortran, BCPL/B, C, Mesa/Cedar, Pascal/Modula-2/Oberon, C++, ...
- C: the de-facto standard
  - Data structure representation control
  - Memory management control
  - Complete mutability via pointers
  - Performant
  - Legacy
- C: the unsafe standard
  - Unchecked array operations

Systems Languages

- Past systems languages: ALGOL, PL/I, Fortran, BCPL/B, C, Mesa/Cedar, Pascal/Modula-2/Oberon, C++, ...
- C: the de-facto standard
  - Data structure representation control
  - Memory management control
  - Complete mutability via pointers
  - Performant
  - Legacy
- C: the unsafe standard
  - Unchecked array operations
  - Pointers $\equiv$ arrays

Systems Languages

- Past systems languages: ALGOL, PL/I, Fortran, BCPL/B, C, Mesa/Cedar, Pascal/Modula-2/Oberon, C++, ...
- C: the de-facto standard
  - Data structure representation control
  - Memory management control
  - Complete mutability via pointers
  - Performant
  - Legacy
- C: the unsafe standard
  - Unchecked array operations
  - Pointers $\equiv$ arrays
  - Unsafe casts

Systems Languages

- Past systems languages: ALGOL, PL/I, Fortran, BCPL/B, C, Mesa/Cedar, Pascal/Modula-2/Oberon, C++, ...
- C: the de-facto standard
  - Data structure representation control
  - Memory management control
  - Complete mutability via pointers
  - Performant
  - Legacy
- C: the unsafe standard
  - Unchecked array operations
  - Pointers $\equiv$ arrays
  - Unsafe casts
  - Aliasing

Systems Languages

- Past systems languages: ALGOL, PL/I, Fortran, BCPL/B, C, Mesa/Cedar, Pascal/Modula-2/Oberon, C++, ...
- C: the de-facto standard
  - Data structure representation control
  - Memory management control
  - Complete mutability via pointers
  - Performant
  - Legacy
- C: the unsafe standard
  - Unchecked array operations
  - Pointers $\equiv$ arrays
  - Unsafe casts
  - Aliasing
  - Undefined behavior

Memory Safety

- Memory safety error: reads or writes outside the referent's storage

Memory Safety

- Memory safety error: reads or writes outside the referent's storage
  - Spatial: outside referent's address bounds

Memory Safety

- Memory safety error: reads or writes outside the referent's storage
  - Spatial: outside referent's address bounds
  - Temporal: outside referent's lifetime

Memory Safety

- Memory safety error: reads or writes outside the referent's storage
  - Spatial: outside referent's address bounds
  - Temporal: outside referent's lifetime
- Ideal technique is

Memory Safety

- Memory safety error: reads or writes outside the referent's storage
  - Spatial: outside referent's address bounds
  - Temporal: outside referent's lifetime
- Ideal technique is
  - Efficient and expressive

Memory Safety

- Memory safety error: reads or writes outside the referent's storage
  - Spatial: outside referent's address bounds
  - Temporal: outside referent's lifetime
- Ideal technique is
  - Efficient and expressive
  - Purely static

Memory Safety

- Memory safety error: reads or writes outside the referent's storage
  - Spatial: outside referent's address bounds
  - Temporal: outside referent's lifetime
- Ideal technique is
  - Efficient and expressive
  - Purely static
  - Precise

Memory Safety

- Memory safety error: reads or writes outside the referent's storage
  - Spatial: outside referent's address bounds
  - Temporal: outside referent's lifetime
- Ideal technique is
  - Efficient and expressive
  - Purely static
  - Precise
  - Automatic

Memory Safety

- Memory safety error: reads or writes outside the referent's storage
  - Spatial: outside referent's address bounds
  - Temporal: outside referent's lifetime
- Ideal technique is
  - Efficient and expressive
  - Purely static
  - Precise
  - Automatic
- Memory errors become type errors, management happens at compile-time

Outline

- Motivation
- Spatial Safety
  - Fat Pointers and Shadow Structures
  - Referent Objects
  - Dependent Types
- Temporal Safety
  - Capabilities and Locks
  - Effects and Regions
  - Linear Types and Ownership

Spatial Safety

Always access within object's bounds

Spatial Safety

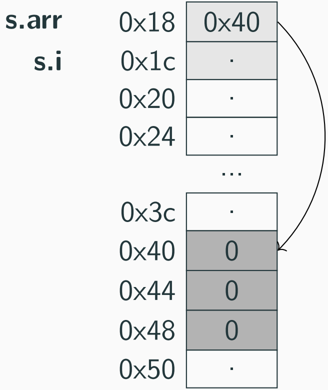
Always access within object's bounds

```
struct { int *arr; int i; } s;
s.arr = calloc(3, sizeof(int));
```

Spatial Safety

Always access within object's bounds

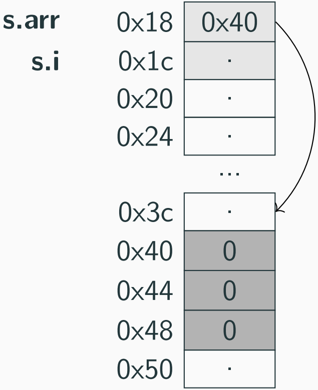
```
struct { int *arr; int i; } s;
s.arr = calloc(3, sizeof(int));
```



Spatial Safety

Always access within object's bounds

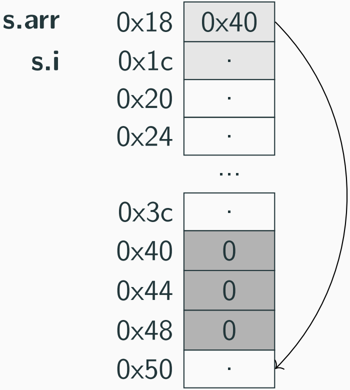
```
struct { int *arr; int i; } s;
s.arr = calloc(3, sizeof(int));
```



Spatial Safety

Always access within object's bounds

```
struct { int *arr; int i; } s;
s.arr = calloc(3, sizeof(int));
```



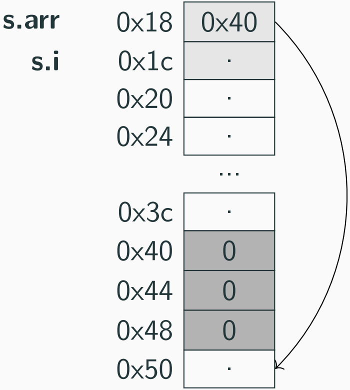
Spatial Safety

Always access within object's bounds

Some approaches:

- Fat Pointers and Shadow Structures
- Referent Objects
- Dependent Types

```
struct { int *arr; int i; } s;
s.arr = calloc(3, sizeof(int));
```



Spatial Safety Example

```
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

Spatial Safety Example

```
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

Spatial Safety Example

```
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

Spatial Safety Example

```
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

Spatial Safety Example

```c
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

Spatial Safety Example

```c
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

Spatial Safety Example

```
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

Spatial Safety Example

```
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

Potential pointer dereference problems:

- Null
- Uninitialized
- Out-of-bounds
- Manufactured

Outline

Fat Pointers

Fat Pointers

- Added base and bound addresses

Fat Pointers

- Added base and bound addresses
- Instrument all pointers and functions

Fat Pointers

- Added base and bound addresses
- Instrument all pointers and functions
- Insert runtime checks before dereferences

Fat Pointers

- Added base and bound addresses
- Instrument all pointers and functions
- Insert runtime checks before dereferences
- Advantages:

Fat Pointers

- Added base and bound addresses
- Instrument all pointers and functions
- Insert runtime checks before dereferences
- Advantages:
    - Quickly find and retrieve metadata

Fat Pointers

- Added base and bound addresses
- Instrument all pointers and functions
- Insert runtime checks before dereferences
- Advantages:
  - Quickly find and retrieve metadata
  - Complete spatial safety

Fat Pointers

- Added base and bound addresses
- Instrument all pointers and functions
- Insert runtime checks before dereferences
- Advantages:
  - Quickly find and retrieve metadata
  - Complete spatial safety
  - No sub-object overflows

Fat Pointers

- Added base and bound addresses
- Instrument all pointers and functions
- Insert runtime checks before dereferences
- Advantages:
  - Quickly find and retrieve metadata
  - Complete spatial safety
  - No sub-object overflows
- Disadvantages:

Fat Pointers

- Added base and bound addresses
- Instrument all pointers and functions
- Insert runtime checks before dereferences
- Advantages:
  - Quickly find and retrieve metadata
  - Complete spatial safety
  - No sub-object overflows
- Disadvantages:
  - Breaks binary compatibility

Fat Pointers

- Added base and bound addresses
- Instrument all pointers and functions
- Insert runtime checks before dereferences
- Advantages:
  - Quickly find and retrieve metadata
  - Complete spatial safety
  - No sub-object overflows
- Disadvantages:
  - Breaks binary compatibility
  - Metadata propagation

Fat Pointers

- Added base and bound addresses
- Instrument all pointers and functions
- Insert runtime checks before dereferences
- Advantages:
  - Quickly find and retrieve metadata
  - Complete spatial safety
  - No sub-object overflows
- Disadvantages:
  - Breaks binary compatibility
  - Metadata propagation
  - Code bloat, memory usage, runtime overhead

Fat Pointers

- Added base and bound addresses
- Instrument all pointers and functions
- Insert runtime checks before dereferences
- Advantages:
  - Quickly find and retrieve metadata
  - Complete spatial safety
  - No sub-object overflows
- Disadvantages:
  - Breaks binary compatibility
  - Metadata propagation
  - Code bloat, memory usage, runtime overhead
  - Unsafe casts overwriting metadata

Fat Pointers

- Added base and bound addresses
- Instrument all pointers and functions
- Insert runtime checks before dereferences
- Advantages:
  - Quickly find and retrieve metadata
  - Complete spatial safety
  - No sub-object overflows
- Disadvantages:
  - Breaks binary compatibility
  - Metadata propagation
  - Code bloat, memory usage, runtime overhead
  - Unsafe casts overwriting metadata

Fat Pointers

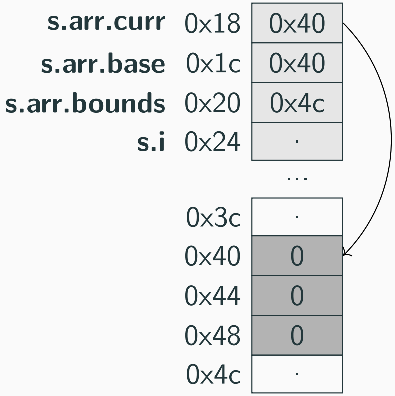
- Added base and bound addresses
- Instrument all pointers and functions
- Insert runtime checks before dereferences
- Advantages:
  - Quickly find and retrieve metadata
  - Complete spatial safety
  - No sub-object overflows
- Disadvantages:
  - Breaks binary compatibility
  - Metadata propagation
  - Code bloat, memory usage, runtime overhead
  - Unsafe casts overwriting metadata

```
struct fptr { int *curr;
     int *base; int *bound; };
struct { struct fptr arr; int i; } s;
s.arr.curr = calloc(3, sizeof(int));
```

Fat Pointers

- Added base and bound addresses
- Instrument all pointers and functions
- Insert runtime checks before dereferences
- Advantages:
  - Quickly find and retrieve metadata
  - Complete spatial safety
  - No sub-object overflows
- Disadvantages:
  - Breaks binary compatibility
  - Metadata propagation
  - Code bloat, memory usage, runtime overhead
  - Unsafe casts overwriting metadata

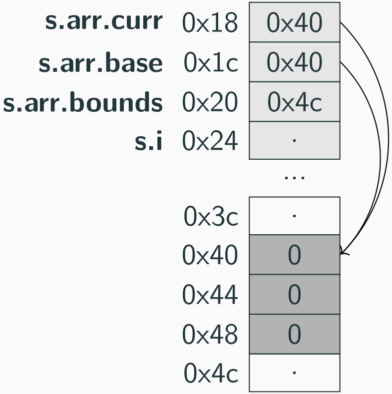
```
struct fptr { int *curr;
     int *base; int *bound; };
struct { struct fptr arr; int i; } s;
s.arr.curr = calloc(3, sizeof(int));
```



Fat Pointers

- Added base and bound addresses
- Instrument all pointers and functions
- Insert runtime checks before dereferences
- Advantages:
  - Quickly find and retrieve metadata
  - Complete spatial safety
  - No sub-object overflows
- Disadvantages:
  - Breaks binary compatibility
  - Metadata propagation
  - Code bloat, memory usage, runtime overhead
  - Unsafe casts overwriting metadata

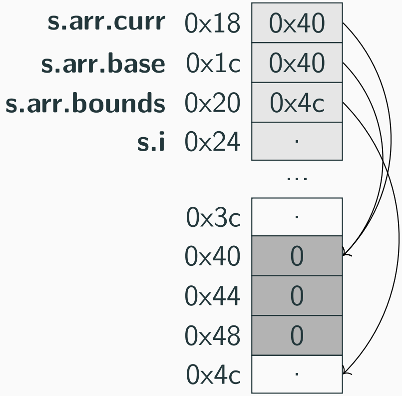
```
struct fptr { int *curr;
     int *base; int *bound; };
struct { struct fptr arr; int i; } s;
s.arr.curr = calloc(3, sizeof(int));
```



Fat Pointers

- Added base and bound addresses
- Instrument all pointers and functions
- Insert runtime checks before dereferences
- Advantages:
  - Quickly find and retrieve metadata
  - Complete spatial safety
  - No sub-object overflows
- Disadvantages:
  - Breaks binary compatibility
  - Metadata propagation
  - Code bloat, memory usage, runtime overhead
  - Unsafe casts overwriting metadata

```
struct fptr { int *curr;
     int *base; int *bound; };
struct { struct fptr arr; int i; } s;
s.arr.curr = calloc(3, sizeof(int));
```



Fat Pointer Approaches

SafeC[1]:

- Safe pointers have value, base, and size
- Complete spatial safety, if transparent storage management and no safe pointer attribute manipulation
- 275% space overhead, 2-6x runtime overhead, 0.35-3x code size overhead
- Some static optimization based on still-valid previous checks

[1] Austin, Breach, and Sohi, "Efficient Detection of All Pointer and Array Access Errors", 1994.

[2] Jim et al., "Cyclone: A Safe Dialect of C.", 2002.

Fat Pointer Approaches

SafeC[1]:

- Safe pointers have value, base, and size
- Complete spatial safety, if transparent storage management and no safe pointer attribute manipulation
- 275% space overhead, 2-6x runtime overhead, 0.35-3x code size overhead
- Some static optimization based on still-valid previous checks

Cyclone[2]:

- Annotations for non-array vs array pointers (can specify size)
- Tagged unions and automatic tag injection

[1] Austin, Breach, and Sohi, "Efficient Detection of All Pointer and Array Access Errors", 1994.

[2] Jim et al., "Cyclone: A Safe Dialect of C.", 2002.

Fat Pointer Approaches

CCured[3]

- Separate pointers on usage (SAFE, SEQ, WILD)
- Whole-program inference to find as many SAFE then SEQ pointers as possible
- Reduce WILD pointers[4] using physical subtyping[5] for upcasts
- Special pointer RTTI carrying runtime type for downcasts

---

[3] Necula, McPeak, and Weimer, "CCured", 2002.

[4] Necula, Condit, et al., "CCured", 2005.

[5] Siff et al., "Coping with Type Casts in C", 1999.

[6] Oiwa, "Implementation of the Memory-safe Full ANSI-C Compiler", 2009.

Fat Pointer Approaches

CCured[3]

- Separate pointers on usage (SAFE, SEQ, WILD)
- Whole-program inference to find as many SAFE then SEQ pointers as possible
- Reduce WILD pointers[4] using physical subtyping[5] for upcasts
- Special pointer RTTI carrying runtime type for downcasts

Fail-Safe C[6]:

- Combines fat pointers w/ fat integers and virtual structure offsets

---

[3] Necula, McPeak, and Weimer, "CCured", 2002.

[4] Necula, Condit, et al., "CCured", 2005.

[5] Siff et al., "Coping with Type Casts in C", 1999.

[6] Oiwa, "Implementation of the Memory-safe Full ANSI-C Compiler", 2009.

Fat Pointers Preventing Spatial Errors

```
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {



        if (*p == token) break;


        p++;
    }

    return (*p == token);
}
```

Fat Pointers Preventing Spatial Errors

```c
1   int find_token(int *data,
2                  int *end,
3                  int token)
4   {
5       int *p = data;
6       while (p < end) {
7
8
9
10          if (*p == token) break;
11
12
13          p++;
14      }
15
16      return (*p == token);
17  }
```

```c
1   int find_token(int *SEQ data,
2                  int *SAFE end,
3                  int token)
4   {
5       int *SEQ p = data;
6       while (p.cur < end) {
7           assert(p.base != 0 &&
8                  0 <= p.cur &&
9                  p.cur < p.bound);
10          if (*p.cur == token) break;
11          p.cur = p.cur + (1 * sizeof(int));
12          p.base  = p.base;   // optimized out
13          p.bound = p.bound;  // "            "
14      }
15      ...(repeat lines 7-9)...
16      return (*p.cur == token);
17  }
```

Fat Pointers Preventing Spatial Errors

```
1   int find_token(int *data,
2                  int *end,
3                  int token)
4   {
5       int *p = data;
6       while (p < end) {
7
8
9
10          if (*p == token) break;
11
12
13          p++;
14      }
15
16      return (*p == token);
17  }
```

```
1   int find_token(int *SEQ data,
2                  int *SAFE end,
3                  int token)
4   {
5       int *SEQ p = data;
6       while (p.cur < end) {
7           assert(p.base != 0 &&
8                  0 <= p.cur &&
9                  p.cur < p.bound);
10          if (*p.cur == token) break;
11          p.cur = p.cur + (1 * sizeof(int));
12          p.base  = p.base;   // optimized out
13          p.bound = p.bound;  // "            "
14      }
15      ...(repeat lines 7-9)...
16      return (*p.cur == token);
17  }
```

Fat Pointers Preventing Spatial Errors

```
1  int find_token(int *data,
2                 int *end,
3                 int token)
4  {
5      int *p = data;
6      while (p < end) {
7
8
9
10         if (*p == token) break;
11
12
13         p++;
14     }
15
16     return (*p == token);
17 }
```

```
1  int find_token(int *SEQ data,
2                 int *SAFE end,
3                 int token)
4  {
5      int *SEQ p = data;
6      while (p.cur < end) {
7          assert(p.base != 0 &&
8                 0 <= p.cur &&
9                 p.cur < p.bound);
10         if (*p.cur == token) break;
11         p.cur = p.cur + (1 * sizeof(int));
12         p.base  = p.base;   // optimized out
13         p.bound = p.bound;  // "           "
14     }
15     ...(repeat lines 7-9)...
16     return (*p.cur == token);
17 }
```

Fat Pointers Preventing Spatial Errors

```
1   int find_token(int *data,
2                  int *end,
3                  int token)
4   {
5       int *p = data;
6       while (p < end) {
7
8
9
10          if (*p == token) break;
11
12
13          p++;
14      }
15
16      return (*p == token);
17  }
```

```
1   int find_token(int *SEQ data,
2                  int *SAFE end,
3                  int token)
4   {
5       int *SEQ p = data;
6       while (p.cur < end) {
7           assert(p.base != 0 &&
8                  0 <= p.cur &&
9                  p.cur < p.bound);
10          if (*p.cur == token) break;
11          p.cur = p.cur + (1 * sizeof(int));
12          p.base  = p.base;   // optimized out
13          p.bound = p.bound;  // "           "
14      }
15      ...(repeat lines 7-9)...
16      return (*p.cur == token);
17  }
```

Fat Pointers Preventing Spatial Errors

```c
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

```c
1  int find_token(int *SEQ data,
2                 int *SAFE end,
3                 int token)
4  {
5      int *SEQ p = data;
6      while (p.cur < end) {
7          assert(p.base != 0 &&
8                 0 <= p.cur &&
9                 p.cur < p.bound);
10         if (*p.cur == token) break;
11         p.cur = p.cur + (1 * sizeof(int));
12         p.base  = p.base;   // optimized out
13         p.bound = p.bound;  // "            "
14     }
15     ...(repeat lines 7-9)...
16     return (*p.cur == token);
17 }
```

Fat Pointers Preventing Spatial Errors

```c
1   int find_token(int *data,
2                  int *end,
3                  int token)
4   {
5       int *p = data;
6       while (p < end) {
7
8
9
10          if (*p == token) break;
11
12
13          p++;
14      }
15
16      return (*p == token);
17  }
```

```c
1   int find_token(int *SEQ data,
2                  int *SAFE end,
3                  int token)
4   {
5       int *SEQ p = data;
6       while (p.cur < end) {
7           assert(p.base != 0 &&
8                  0 <= p.cur &&
9                  p.cur < p.bound);
10          if (*p.cur == token) break;
11          p.cur = p.cur + (1 * sizeof(int));
12          p.base  = p.base;   // optimized out
13          p.bound = p.bound;  // "           "
14      }
15      ...(repeat lines 7-9)...
16      return (*p.cur == token);
17  }
```

Pointer-Based – Shadow Structures

MSCC[7]

[7] Xu, DuVarney, and Sekar, "An Efficient and Backwards-compatible Transformation to Ensure Memory Safety of C Programs", 2004.

Pointer-Based – Shadow Structures

MSCC[7]

- *Split* metadata from pointer, preserving layout
- Every value has linked shadow structure mirroring entire data structure
- Transform every function call to take additional metadata parameters
- Wrappers for external functions; cannot detect memory errors

[7] Xu, DuVarney, and Sekar, "An Efficient and Backwards-compatible Transformation to Ensure Memory Safety of C Programs", 2004.

Shadow Structures Example

```c




int find_token(
  int *data,
  int *end,
  int token)
{
    int *p = data;

    while (p < end) {

        if (*p == token) break;
        p++;
    }

    return (*p == token);
}
```

Shadow Structures Example

```c
int find_token(
  int *data,
  int *end,
  int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

```c
struct ptr_info {
    void *base;
    unsigned long bound;
};
int find_token(
  int *data, ptr_info *data_info,
  int *end,  ptr_info *end_info,
  int token)
{
    int *p = data;
    ptr_info  p_info = *data_info;
    while (p < end) {
        CHECK_SPATIAL(p, sizeof(*p), p_info);
        if (*p == token) break;
        p++;
    }
    CHECK_SPATIAL(p, sizeof(*p), p_info);
    return (*p == token);
}
```

Shadow Structures Example

```c




int find_token(
  int *data,
  int *end,
  int token)
{
    int *p = data;

    while (p < end) {

        if (*p == token) break;
        p++;
    }

    return (*p == token);
}
```

```c
struct ptr_info {
    void *base;
    unsigned long bound;
};
int find_token(
  int *data, ptr_info *data_info,
  int *end,  ptr_info *end_info,
  int token)
{
    int *p = data;
    ptr_info  p_info = *data_info;
    while (p < end) {
        CHECK_SPATIAL(p, sizeof(*p), p_info);
        if (*p == token) break;
        p++;
    }
    CHECK_SPATIAL(p, sizeof(*p), p_info);
    return (*p == token);
}
```

Shadow Structures Example

```c
int find_token(
  int *data,
  int *end,
  int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

```c
struct ptr_info {
    void *base;
    unsigned long bound;
};
int find_token(
  int *data, ptr_info *data_info,
  int *end,  ptr_info *end_info,
  int token)
{
    int *p = data;
    ptr_info  p_info = *data_info;
    while (p < end) {
        CHECK_SPATIAL(p, sizeof(*p), p_info);
        if (*p == token) break;
        p++;
    }
    CHECK_SPATIAL(p, sizeof(*p), p_info);
    return (*p == token);
}
```

Shadow Structures Example

```c




int find_token(
  int *data,
  int *end,
  int token)
{
    int *p = data;

    while (p < end) {

        if (*p == token) break;
        p++;
    }

    return (*p == token);
}
```

```c
struct ptr_info {
    void *base;
    unsigned long bound;
};
int find_token(
  int *data, ptr_info *data_info,
  int *end,  ptr_info *end_info,
  int token)
{
    int *p = data;
    ptr_info  p_info = *data_info;
    while (p < end) {
        CHECK_SPATIAL(p, sizeof(*p), p_info);
        if (*p == token) break;
        p++;
    }
    CHECK_SPATIAL(p, sizeof(*p), p_info);
    return (*p == token);
}
```

Shadow Structures Example

```c
int find_token(
  int *data,
  int *end,
  int token)
{
    int *p = data;

    while (p < end) {

        if (*p == token) break;
        p++;
    }

    return (*p == token);
}
```

```c
struct ptr_info {
    void *base;
    unsigned long bound;
};
int find_token(
  int *data, ptr_info *data_info,
  int *end,  ptr_info *end_info,
  int token)
{
    int *p = data;
    ptr_info  p_info = *data_info;
    while (p < end) {
        CHECK_SPATIAL(p, sizeof(*p), p_info);
        if (*p == token) break;
        p++;
    }
    CHECK_SPATIAL(p, sizeof(*p), p_info);
    return (*p == token);
}
```

Outline

Referent Objects

Objects[8,9]

- Metadata about objects, not pointers

[8] Jones and Kelly, "Backwards-compatible bounds checking for arrays and pointers in C programs", 1997.

[9] Ruwase and Lam, "A Practical Dynamic Buffer Overflow Detector", 2004.

Referent Objects

Objects[8, 9]

- Metadata about objects, not pointers
- Global database relates every allocated address to corresponding object metadata

[8] Jones and Kelly, “Backwards-compatible bounds checking for arrays and pointers in C programs”, 1997.

[9] Ruwase and Lam, “A Practical Dynamic Buffer Overflow Detector”, 2004.

Referent Objects

Objects[8,9]

- Metadata about objects, not pointers
- Global database relates every allocated address to corresponding object metadata
- Every pointer to same object shares same metadata

[8] Jones and Kelly, "Backwards-compatible bounds checking for arrays and pointers in C programs", 1997.

[9] Ruwase and Lam, "A Practical Dynamic Buffer Overflow Detector", 2004.

Referent Objects

Objects[8,9]

- Metadata about objects, not pointers
- Global database relates every allocated address to corresponding object metadata
- Every pointer to same object shares same metadata
- Bounds check on pointer arithmetic

[8] Jones and Kelly, "Backwards-compatible bounds checking for arrays and pointers in C programs", 1997.

[9] Ruwase and Lam, "A Practical Dynamic Buffer Overflow Detector", 2004.

Referent Objects

Objects[8,9]

- Metadata about objects, not pointers
- Global database relates every allocated address to corresponding object metadata
- Every pointer to same object shares same metadata
- Bounds check on pointer arithmetic
- 2-12x overhead

[8] Jones and Kelly, "Backwards-compatible bounds checking for arrays and pointers in C programs", 1997.

[9] Ruwase and Lam, "A Practical Dynamic Buffer Overflow Detector", 2004.

Referent Objects

Objects[8,9]

- Metadata about objects, not pointers
- Global database relates every allocated address to corresponding object metadata
- Every pointer to same object shares same metadata
- Bounds check on pointer arithmetic
- 2-12x overhead

[8] Jones and Kelly, “Backwards-compatible bounds checking for arrays and pointers in C programs”, 1997.

[9] Ruwase and Lam, “A Practical Dynamic Buffer Overflow Detector”, 2004.

Referent Objects

Objects[8, 9]

- Metadata about objects, not pointers
- Global database relates every allocated address to corresponding object metadata
- Every pointer to same object shares same metadata
- Bounds check on pointer arithmetic
- 2-12x overhead

Advantages:

- Compatible with uninstrumented code

[8] Jones and Kelly, "Backwards-compatible bounds checking for arrays and pointers in C programs", 1997.

[9] Ruwase and Lam, "A Practical Dynamic Buffer Overflow Detector", 2004.

Referent Objects

Objects[8,9]

- Metadata about objects, not pointers
- Global database relates every allocated address to corresponding object metadata
- Every pointer to same object shares same metadata
- Bounds check on pointer arithmetic
- 2-12x overhead

Advantages:

- Compatible with uninstrumented code

Disadvantages:

- Special mechanisms to handle legal OOB pointers
- Splay-tree object lookup overhead
- Incomplete spatial safety: sub-object overflows

[8] Jones and Kelly, "Backwards-compatible bounds checking for arrays and pointers in C programs", 1997.

[9] Ruwase and Lam, "A Practical Dynamic Buffer Overflow Detector", 2004.

The Sub-Object Problem

```
struct node {char str[3]; void (*func)(); };
struct node *n = (struct node *) malloc(sizeof(node));
char *s = n.str;
strcpy(s, "bad!");
```

The Sub-Object Problem

```c
struct node {char str[3]; void (*func)(); };
struct node *n = (struct node *) malloc(sizeof(node));
char *s = n.str;
strcpy(s, "bad!");
```

The Sub-Object Problem

```
struct node {char str[3]; void (*func)(); };
struct node *n = (struct node *) malloc(sizeof(node));
char *s = n.str;
strcpy(s, "bad!");
```

The Sub-Object Problem

```
struct node {char str[3]; void (*func)(); };
struct node *n = (struct node *) malloc(sizeof(node));
char *s = n.str;
strcpy(s, "bad!");
```

The Sub-Object Problem

```
struct node {char str[3]; void (*func)(); };
struct node *n = (struct node *) malloc(sizeof(node));
char *s = n.str;
strcpy(s, "bad!");
```

The Sub-Object Problem

```c
struct node {char str[3]; void (*func)(); };
struct node *n = (struct node *) malloc(sizeof(node));
char *s = n.str;
strcpy(s, "bad!");
```

- `n` and `s` have the same address ⇒ map to same object in global database
- `strcpy` will see `s`'s size as that of `n`

Referent Objects Approaches

SafeCode[10]

- Use automatic pool allocation (APA)[11]
- Use separate, smaller data structures to store bounds metadata for each partition
- 1.2x overhead

[10] Dhurjati and Adve, "Backwards-compatible Array Bounds Checking for C with Very Low Overhead", 2006.

[11] Lattner and Adve, "Automatic Pool Allocation", 2005.

[12] Akritidis et al., "Baggy Bounds Checking", 2009.

Referent Objects Approaches

SafeCode[10]

- Use automatic pool allocation (APA)[11]
- Use separate, smaller data structures to store bounds metadata for each partition
- 1.2x overhead

Baggy Bounds Checking (BBC)[12]

- Compact bounds representation and efficient way to look up object bounds
- Align base addresses to be multiple of padded size
- Replace splay tree with small lookup table
- 0.6x overhead

[10] Dhurjati and Adve, "Backwards-compatible Array Bounds Checking for C with Very Low Overhead", 2006.

[11] Lattner and Adve, "Automatic Pool Allocation", 2005.

[12] Akritidis et al., "Baggy Bounds Checking", 2009.

Referent Objects Example

```c
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;




        p++;
    }
    return (*p == token);
}
```

Referent Objects Example

```
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

```
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        int *q = p + 1;
        int size = 1 << TABLE[p>>4];
        int base = p & ~(size - 1);
        assert(q >= base && q - base < size);
        p++;
    }
    return (*p == token);
}
```

Referent Objects Example

```
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

```
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        int *q = p + 1;
        int size = 1 << TABLE[p>>4];
        int base = p & ~(size - 1);
        assert(q >= base && q - base < size);
        p++;
    }
    return (*p == token);
}
```

Referent Objects Example

```c
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

```c
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        int *q = p + 1;
        int size = 1 << TABLE[p>>4];
        int base = p & ~(size - 1);
        assert(q >= base && q - base < size);
        p++;
    }
    return (*p == token);
}
```

Referent Objects Example

```c
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

```c
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        int *q = p + 1;
        int size = 1 << TABLE[p>>4];
        int base = p & ~(size - 1);
        assert(q >= base && q - base < size);
        p++;
    }
    return (*p == token);
}
```

Referent Objects Example

```
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

```
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        int *q = p + 1;
        int size = 1 << TABLE[p>>4];
        int base = p & ~(size - 1);
        assert(q >= base && q - base < size);
        p++;
    }
    return (*p == token);
}
```

Softbound

Softbound[13]

- Base and bound metadata for each pointer, stored in disjoint metadata table

[13] Nagarakatte, Zhao, Milo MK Martin, et al., "SoftBound", 2009.

Softbound

Softbound[13]

- Base and bound metadata for each pointer, stored in disjoint metadata table
- Total spatial safety of pointer-based approaches

[13] Nagarakatte, Zhao, Milo MK Martin, et al., “SoftBound”, 2009.

Softbound

Softbound[13]

- Base and bound metadata for each pointer, stored in disjoint metadata table
- Total spatial safety of pointer-based approaches
- Source compatibility, separate compilation of object-based approaches

[13] Nagarakatte, Zhao, Milo MK Martin, et al., "SoftBound", 2009.

Softbound[13]

- Base and bound metadata for each pointer, stored in disjoint metadata table
- Total spatial safety of pointer-based approaches
- Source compatibility, separate compilation of object-based approaches
- Runtime bounds checks on each dereference

[13] Nagarakatte, Zhao, Milo MK Martin, et al., "SoftBound", 2009.

Softbound

Softbound[13]

- Base and bound metadata for each pointer, stored in disjoint metadata table
- Total spatial safety of pointer-based approaches
- Source compatibility, separate compilation of object-based approaches
- Runtime bounds checks on each dereference
- Propagate metadata as extra arguments

[13] Nagarakatte, Zhao, Milo MK Martin, et al., "SoftBound", 2009.

Softbound

Softbound[13]

- Base and bound metadata for each pointer, stored in disjoint metadata table
- Total spatial safety of pointer-based approaches
- Source compatibility, separate compilation of object-based approaches
- Runtime bounds checks on each dereference
- Propagate metadata as extra arguments
- Arbitrary casts allowed

[13] Nagarakatte, Zhao, Milo MK Martin, et al., "SoftBound", 2009.

Softbound

Softbound[13]

- Base and bound metadata for each pointer, stored in disjoint metadata table
- Total spatial safety of pointer-based approaches
- Source compatibility, separate compilation of object-based approaches
- Runtime bounds checks on each dereference
- Propagate metadata as extra arguments
- Arbitrary casts allowed
- 67% overhead

[13] Nagarakatte, Zhao, Milo MK Martin, et al., "SoftBound", 2009.

Outline

Dependent Types

Dependent types are *typed-valued functions*[14]

[14]Pierce, *Advanced topics in types and programming languages*, 2005.

[15]Martin-Löf, "Constructive mathematics and computer programming", 1984.

Dependent Types

Dependent types are *typed-valued functions*[14]

$$\texttt{Vector} : \texttt{Nat} \to \texttt{Type} \to \texttt{Type}$$

[14] Pierce, *Advanced topics in types and programming languages*, 2005.

[15] Martin-Löf, "Constructive mathematics and computer programming", 1984.

Dependent Types

Dependent types are *typed-valued functions*[14]

$$\texttt{Vector} : \texttt{Nat} \rightarrow \texttt{Type} \rightarrow \texttt{Type}$$
$$\texttt{nil} : \texttt{Vector 0 a}$$

[14]Pierce, *Advanced topics in types and programming languages*, 2005.

[15]Martin-Löf, "Constructive mathematics and computer programming", 1984.

Dependent Types

Dependent types are *typed-valued functions*[14]

$$\texttt{Vector} \;:\; \texttt{Nat} \to \texttt{Type} \to \texttt{Type}$$
$$\texttt{nil} \;:\; \texttt{Vector 0 a}$$
$$\texttt{cons} \;:\; \Pi\texttt{n:Nat.a} \to \texttt{Vector n a} \to \texttt{Vector (n+1) a}$$

[14] Pierce, *Advanced topics in types and programming languages*, 2005.

[15] Martin-Löf, "Constructive mathematics and computer programming", 1984.

Dependent Types

Dependent types are *typed-valued functions*[14]

$$\begin{aligned}
\texttt{Vector} &: \texttt{Nat} \to \texttt{Type} \to \texttt{Type} \\
\texttt{nil} &: \texttt{Vector 0 a} \\
\texttt{cons} &: \Pi\texttt{n:Nat.a} \to \texttt{Vector n a} \to \texttt{Vector (n+1) a} \\
\texttt{(cons 'a' (cons 'b' nil))} &: \texttt{Vector 2 Char}
\end{aligned}$$

[14] Pierce, *Advanced topics in types and programming languages*, 2005.

[15] Martin-Löf, "Constructive mathematics and computer programming", 1984.

Dependent Types

Dependent types are *typed-valued functions*[14]

$$\texttt{Vector : Nat} \to \texttt{Type} \to \texttt{Type}$$

$$\texttt{nil : Vector 0 a}$$

$$\texttt{cons : }\Pi\texttt{n:Nat.a} \to \texttt{Vector n a} \to \texttt{Vector (n+1) a}$$

$$\texttt{(cons 'a' (cons 'b' nil)) : Vector 2 Char}$$

$$\texttt{head : }\Pi\texttt{n:Nat.Vector (n+1) a -> a}$$

[14] Pierce, *Advanced topics in types and programming languages*, 2005.

[15] Martin-Löf, "Constructive mathematics and computer programming", 1984.

Dependent Types

Dependent types are *typed-valued functions*[14]

$$
\begin{aligned}
\texttt{Vector} &: \texttt{Nat} \rightarrow \texttt{Type} \rightarrow \texttt{Type} \\
\texttt{nil} &: \texttt{Vector 0 a} \\
\texttt{cons} &: \Pi\texttt{n:Nat.a} \rightarrow \texttt{Vector n a} \rightarrow \texttt{Vector (n+1) a} \\
\texttt{(cons 'a' (cons 'b' nil))} &: \texttt{Vector 2 Char} \\
\texttt{head} &: \Pi\texttt{n:Nat.Vector (n+1) a -> a} \\
&\texttt{head nil}
\end{aligned}
$$

[14] Pierce, *Advanced topics in types and programming languages*, 2005.

[15] Martin-Löf, "Constructive mathematics and computer programming", 1984.

Dependent Types

Dependent types are *typed-valued functions*[14]

$$\texttt{Vector} : \texttt{Nat} \to \texttt{Type} \to \texttt{Type}$$
$$\texttt{nil} : \texttt{Vector 0 a}$$
$$\texttt{cons} : \Pi\texttt{n:Nat.a} \to \texttt{Vector n a} \to \texttt{Vector (n+1) a}$$
$$\texttt{(cons 'a' (cons 'b' nil))} : \texttt{Vector 2 Char}$$
$$\texttt{head} : \Pi\texttt{n:Nat.Vector (n+1) a -> a}$$
$$\texttt{head nil} \Rightarrow \text{Rejected!}$$

[14] Pierce, *Advanced topics in types and programming languages*, 2005.

[15] Martin-Löf, "Constructive mathematics and computer programming", 1984.

Dependent Types

Dependent types are *typed-valued functions*[14]

$$\texttt{Vector} : \texttt{Nat} \rightarrow \texttt{Type} \rightarrow \texttt{Type}$$
$$\texttt{nil} : \texttt{Vector 0 a}$$
$$\texttt{cons} : \Pi\texttt{n:Nat.a} \rightarrow \texttt{Vector n a} \rightarrow \texttt{Vector (n+1) a}$$
$$\texttt{(cons 'a' (cons 'b' nil))} : \texttt{Vector 2 Char}$$
$$\texttt{head} : \Pi\texttt{n:Nat.Vector (n+1) a -> a}$$
$$\texttt{head nil} \Rightarrow \text{Rejected!}$$

- Based on type theory work by Martin-Löf[15]
- Undecidability of type checking: arbitrary computation to check type equality
- Work on defining equality and restricting forms of index terms

[14] Pierce, *Advanced topics in types and programming languages*, 2005.

[15] Martin-Löf, "Constructive mathematics and computer programming", 1984.

Early Uses of Dependent Types

Dependent ML[16] and Cayenne[17]

- Reduce static array bound checking to constraint satisfiability
- DML uses *indexed types*: limit indices to linear integer and boolean expressions; compile-time decidable
- Cayenne has *no restrictions* on types: undecidability of arbitrary expression equivalence and thus type checking

Xanadu[18]

- Imperative environment
- Restrict index expressions in types to integer constraint domain

[16] Xi and Pfenning, "Eliminating Array Bound Checking Through Dependent Types", 1998.

[17] Augustsson, "Cayenne—a Language with Dependent Types", 1998.

[18] Xi, "Imperative programming with dependent types", 2000.

Dependent Types in Imperative Languages

SafeDrive[19] and Deputy[20,21]

- User-added annotations relating pointers to bounds

[19]Zhou et al., "SafeDrive", 2006.

[20]Condit et al., "Dependent Types for Low-Level Programming", 2007.

[21]Anderson, "Static Analysis of C for Hybrid Type Checking", 2007.

[22]Cooprider et al., "Efficient Memory Safety for TinyOS", 2007.

Dependent Types in Imperative Languages

SafeDrive[19] and Deputy[20,21]

- User-added annotations relating pointers to bounds
  - `safe`, `sentinel`, `count(n)`, `bound(lo,hi)`

[19] Zhou et al., "SafeDrive", 2006.

[20] Condit et al., "Dependent Types for Low-Level Programming", 2007.

[21] Anderson, "Static Analysis of C for Hybrid Type Checking", 2007.

[22] Cooprider et al., "Efficient Memory Safety for TinyOS", 2007.

Dependent Types in Imperative Languages

SafeDrive[19] and Deputy[20,21]

- User-added annotations relating pointers to bounds
  - `safe`, `sentinel`, `count(n)`, `bound(lo,hi)`
  - Use constants/variables/field names in immediately enclosing scope

[19] Zhou et al., "SafeDrive", 2006.

[20] Condit et al., "Dependent Types for Low-Level Programming", 2007.

[21] Anderson, "Static Analysis of C for Hybrid Type Checking", 2007.

[22] Cooprider et al., "Efficient Memory Safety for TinyOS", 2007.

Dependent Types in Imperative Languages

SafeDrive[19] and Deputy[20,21]

- User-added annotations relating pointers to bounds
  - `safe`, `sentinel`, `count(n)`, `bound(lo,hi)`
  - Use constants/variables/field names in immediately enclosing scope
- Three-phase pass over annotated C programs, emits C code

[19] Zhou et al., "SafeDrive", 2006.

[20] Condit et al., "Dependent Types for Low-Level Programming", 2007.

[21] Anderson, "Static Analysis of C for Hybrid Type Checking", 2007.

[22] Cooprider et al., "Efficient Memory Safety for TinyOS", 2007.

Dependent Types in Imperative Languages

SafeDrive[19] and Deputy[20,21]

- User-added annotations relating pointers to bounds
  - `safe`, `sentinel`, `count(n)`, `bound(lo,hi)`
  - Use constants/variables/field names in immediately enclosing scope
- Three-phase pass over annotated C programs, emits C code
  1. Automatic addition of bounds annotations for pointer types

[19] Zhou et al., "SafeDrive", 2006.

[20] Condit et al., "Dependent Types for Low-Level Programming", 2007.

[21] Anderson, "Static Analysis of C for Hybrid Type Checking", 2007.

[22] Cooprider et al., "Efficient Memory Safety for TinyOS", 2007.

Dependent Types in Imperative Languages

SafeDrive[19] and Deputy[20,21]

- User-added annotations relating pointers to bounds
  - `safe`, `sentinel`, `count(n)`, `bound(lo,hi)`
  - Use constants/variables/field names in immediately enclosing scope
- Three-phase pass over annotated C programs, emits C code
  1. Automatic addition of bounds annotations for pointer types
  2. Flow-insensitive type checking (insert run-time checks; helps decidability)

---

[19] Zhou et al., "SafeDrive", 2006.

[20] Condit et al., "Dependent Types for Low-Level Programming", 2007.

[21] Anderson, "Static Analysis of C for Hybrid Type Checking", 2007.

[22] Cooprider et al., "Efficient Memory Safety for TinyOS", 2007.

Dependent Types in Imperative Languages

SafeDrive[19] and Deputy[20,21]

- User-added annotations relating pointers to bounds
  - `safe`, `sentinel`, `count(n)`, `bound(lo,hi)`
  - Use constants/variables/field names in immediately enclosing scope
- Three-phase pass over annotated C programs, emits C code
  1. Automatic addition of bounds annotations for pointer types
  2. Flow-insensitive type checking (insert run-time checks; helps decidability)
  3. Flow-sensitive check optimization

---

[19] Zhou et al., "SafeDrive", 2006.

[20] Condit et al., "Dependent Types for Low-Level Programming", 2007.

[21] Anderson, "Static Analysis of C for Hybrid Type Checking", 2007.

[22] Cooprider et al., "Efficient Memory Safety for TinyOS", 2007.

Dependent Types in Imperative Languages

SafeDrive[19] and Deputy[20,21]

- User-added annotations relating pointers to bounds
  - `safe`, `sentinel`, `count(n)`, `bound(lo,hi)`
  - Use constants/variables/field names in immediately enclosing scope
- Three-phase pass over annotated C programs, emits C code
  1. Automatic addition of bounds annotations for pointer types
  2. Flow-insensitive type checking (insert run-time checks; helps decidability)
  3. Flow-sensitive check optimization
- More C Support with dependent union tags , Safe TinyOS[22]

---

[19] Zhou et al., "SafeDrive", 2006.

[20] Condit et al., "Dependent Types for Low-Level Programming", 2007.

[21] Anderson, "Static Analysis of C for Hybrid Type Checking", 2007.

[22] Cooprider et al., "Efficient Memory Safety for TinyOS", 2007.

Deputy Example for Spatial Safety

```c
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

Deputy Example for Spatial Safety

```c
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

```c
int find_token(int * bound(data, end) data,
               int * sentinel end,
               int token)
{
    assert(data != NULL);
    assert(end != NULL);
    int *p bound(p, end) = data;
    while (p < end) {
        assert(p != NULL);
        assert(p < end);
        if (*p == token) break;
        assert(p <= p + 1 <= end);
        p++;
    }
    assert(p != NULL);
    assert(p < end);
    return (*p == token);
}
```

Deputy Example for Spatial Safety

```c
int find_token(int *data,
               int *end,
               int token)
{


    int *p = data;
    while (p < end) {


        if (*p == token) break;

        p++;
    }


    return (*p == token);
}
```

```c
int find_token(int * bound(data, end) data,
               int * sentinel end,
               int token)
{
    assert(data != NULL);
    assert(end != NULL);
    int *p bound(p, end) = data;
    while (p < end) {
        assert(p != NULL);
        assert(p < end);
        if (*p == token) break;
        assert(p <= p + 1 <= end);
        p++;
    }
    assert(p != NULL);
    assert(p < end);
    return (*p == token);
}
```

Deputy Example for Spatial Safety

```c
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

```c
int find_token(int * bound(data, end) data,
               int * sentinel end,
               int token)
{
    assert(data != NULL);
    assert(end != NULL);
    int *p bound(p, end) = data;
    while (p < end) {
        assert(p != NULL);
        assert(p < end);
        if (*p == token) break;
        assert(p <= p + 1 <= end);
        p++;
    }
    assert(p != NULL);
    assert(p < end);
    return (*p == token);
}
```

Deputy Example for Spatial Safety

```c
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

```c
int find_token(int * bound(data, end) data,
               int * sentinel end,
               int token)
{
    assert(data != NULL);
    assert(end != NULL);
    int *p bound(p, end) = data;
    while (p < end) {
        assert(p != NULL);
        assert(p < end);
        if (*p == token) break;
        assert(p <= p + 1 <= end);
        p++;
    }
    assert(p != NULL);
    assert(p < end);
    return (*p == token);
}
```

Deputy Example for Spatial Safety

```c
int find_token(int *data,
               int *end,
               int token)
{


    int *p = data;
    while (p < end) {


        if (*p == token) break;

        p++;
    }


    return (*p == token);
}
```

```c
int find_token(int * bound(data, end) data,
               int * sentinel end,
               int token)
{
    assert(data != NULL);
    assert(end != NULL);
    int *p bound(p, end) = data;
    while (p < end) {
        assert(p != NULL);
        assert(p < end);
        if (*p == token) break;
        assert(p <= p + 1 <= end);
        p++;
    }
    assert(p != NULL);
    assert(p < end);
    return (*p == token);
}
```

Deputy Example for Spatial Safety

```c
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

```c
int find_token(int * bound(data, end) data,
               int * sentinel end,
               int token)
{
    assert(data != NULL);
    assert(end != NULL);
    int *p bound(p, end) = data;
    while (p < end) {
        assert(p != NULL);
        assert(p < end);
        if (*p == token) break;
        assert(p <= p + 1 <= end);
        p++;
    }
    assert(p != NULL);
    assert(p < end);
    return (*p == token);
}
```

Deputy Example for Spatial Safety

```c
int find_token(int *data,
               int *end,
               int token)
{


    int *p = data;
    while (p < end) {


        if (*p == token) break;

        p++;
    }


    return (*p == token);
}
```

```c
int find_token(int * bound(data, end) data,
               int * sentinel end,
               int token)
{
    assert(data != NULL);
    assert(end != NULL);
    int *p bound(p, end) = data;
    while (p < end) {
        assert(p != NULL);
        assert(p < end);
        if (*p == token) break;
        assert(p <= p + 1 <= end);
        p++;
    }
    assert(p != NULL);
    assert(p < end);
    return (*p == token);
}
```

Deputy Example for Spatial Safety

```c
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

```c
int find_token(int * bound(data, end) data,
               int * sentinel end,
               int token)
{
    assert(data != NULL);
    assert(end != NULL);
    int *p bound(p, end) = data;
    while (p < end) {
        assert(p != NULL);
        assert(p < end);
        if (*p == token) break;
        assert(p <= p + 1 <= end);
        p++;
    }
    assert(p != NULL);
    assert(p < end);
    return (*p == token);
}
```

Abstract Syntax, For Your Consideration

$x, y \in \text{Variables} \quad \text{op} \in \text{Binary ops} \quad n \in \text{Integers} \quad \text{comp} \in \text{Comparison Ops}$

$$
\begin{array}{rrcl}
\textit{Ctors} & C & ::= & \text{int} \mid \text{ref} \mid \text{array} \\
\textit{Types} & \tau & ::= & C \mid \tau_1\ \tau_2 \mid \tau\ e \\
\textit{L-exprs} & l & ::= & x \mid *\,e
\end{array}
$$

Abstract Syntax, For Your Consideration

$x, y \in \text{Variables} \quad \text{op} \in \text{Binary ops} \quad n \in \text{Integers} \quad \text{comp} \in \text{Comparison Ops}$

$$
\begin{array}{rrcl}
\textit{Ctors} & C & ::= & \text{int} \mid \text{ref} \mid \text{array} \\
\textit{Types} & \tau & ::= & C \mid \tau_1\ \tau_2 \mid \tau\ e \\
\textit{L-exprs} & l & ::= & x \mid *\, e
\end{array}
$$

$$
\begin{array}{rrcl}
\textit{Exprs} & e & ::= & n \mid l \mid e_1\ \text{op}\ e_2 \\
\textit{Cmds} & c & ::= & l := e \mid \text{assert}(\gamma) \mid c_1 ; c_2 \mid \ldots \\
\text{Preds} & \gamma & ::= & e_1\ \text{comp}\ e_2 \mid \text{true} \mid \gamma_1 \wedge \gamma_2
\end{array}
$$

Typing Rules, For Your Consideration

Local Expressions: $\Gamma \vdash_L e : \tau$

Typing Rules, For Your Consideration

Local Expressions: $\Gamma \vdash_L e : \tau$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_L x : \tau}\ (\textsc{local name}) \qquad \frac{}{\Gamma \vdash_L n : \mathit{int}}\ (\textsc{local num})$$

Typing Rules, For Your Consideration

Local Expressions: $\Gamma \vdash_L e : \tau$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_L x : \tau}\ (\textsc{local name}) \qquad \frac{}{\Gamma \vdash_L n : \mathit{int}}\ (\textsc{local num})$$

Non-local Expressions: $\Gamma \vdash e : \tau \Rightarrow \gamma$

Typing Rules, For Your Consideration

Local Expressions: $\Gamma \vdash_L e : \tau$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_L x : \tau} \text{ (LOCAL NAME)} \qquad \frac{}{\Gamma \vdash_L n : \mathit{int}} \text{ (LOCAL NUM)}$$

Non-local Expressions: $\Gamma \vdash e : \tau \Rightarrow \gamma$

$$\frac{\Gamma \vdash e : \text{ref } \tau \Rightarrow \gamma}{\Gamma \vdash *e : \tau \Rightarrow \gamma} \text{ (DEREF)}$$

Typing Rules, For Your Consideration

Local Expressions: $\Gamma \vdash_L e : \tau$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_L x : \tau}\ (\textsc{local name}) \qquad \frac{}{\Gamma \vdash_L n : \mathit{int}}\ (\textsc{local num})$$

Non-local Expressions: $\Gamma \vdash e : \tau \Rightarrow \gamma$

$$\frac{\Gamma \vdash e : \mathsf{ref}\ \tau \Rightarrow \gamma}{\Gamma \vdash *e : \tau \Rightarrow \gamma}\ (\textsc{deref})$$

Commands: $\Gamma \vdash c \Rightarrow c'$

Typing Rules, For Your Consideration

Local Expressions: $\Gamma \vdash_L e : \tau$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_L x : \tau}\ (\textsc{Local Name}) \qquad \frac{}{\Gamma \vdash_L n : int}\ (\textsc{Local Num})$$

Non-local Expressions: $\Gamma \vdash e : \tau \Rightarrow \gamma$

$$\frac{\Gamma \vdash e : \text{ref } \tau \Rightarrow \gamma}{\Gamma \vdash *e : \tau \Rightarrow \gamma}\ (\textsc{Deref})$$

Commands: $\Gamma \vdash c \Rightarrow c'$

$$\frac{x \in \text{Dom}(\Gamma) \qquad \forall (y : \tau_y) \in \Gamma,\ \Gamma \vdash y[e/x] : \tau_y[e/x] \Rightarrow \gamma_y}{\Gamma \vdash x := e \Rightarrow \text{assert}(\bigwedge_{y \in \text{Dom}(\Gamma)} \gamma_y);\ x := e}\ (\textsc{Var Write})$$

Typing Rules, For Your Consideration

Local Expressions: $\Gamma \vdash_L e : \tau$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_L x : \tau}\ \textsc{(Local Name)} \qquad \frac{}{\Gamma \vdash_L n : int}\ \textsc{(Local Num)}$$

Non-local Expressions: $\Gamma \vdash e : \tau \Rightarrow \gamma$

$$\frac{\Gamma \vdash e : \mathsf{ref}\ \tau \Rightarrow \gamma}{\Gamma \vdash *e : \tau \Rightarrow \gamma}\ \textsc{(Deref)}$$

Commands: $\Gamma \vdash c \Rightarrow c'$

$$\frac{x \in \mathsf{Dom}(\Gamma) \qquad \forall (y : \tau_y) \in \Gamma,\ \Gamma \vdash y[e/x] : \tau_y[e/x] \Rightarrow \gamma_y}{\Gamma \vdash x := e \Rightarrow \mathsf{assert}(\bigwedge_{y \in \mathsf{Dom}(\Gamma)} \gamma_y);\ \ x := e}\ \textsc{(Var Write)}$$

The Interesting Rules

Dereferencing

$$\frac{\Gamma \vdash e : \mathsf{array}\ \tau\ \ e_{len} \Rightarrow \gamma_e}{\Gamma \vdash *e; \tau \Rightarrow \gamma_e \wedge (0 < e_{len})}\ (\textsc{array deref})$$

Dereferencing

$$\frac{\Gamma \vdash e : \mathsf{array}\ \tau\ e_{len} \Rightarrow \gamma_e}{\Gamma \vdash *e; \tau \Rightarrow \gamma_e \wedge (0 < e_{len})}\ (\textsc{array deref})$$

Arithmetic

$$\frac{\Gamma \vdash e : \mathsf{array}\ \tau\ e_{len} \Rightarrow \gamma_e \qquad \Gamma \vdash e' : \mathsf{int} \Rightarrow \gamma_{e'}}{\Gamma \vdash e + e' : \mathsf{array}\ \tau\ (e_{len} - e') \Rightarrow \gamma_e \wedge \gamma'_e \wedge (0 \leqslant e' \leqslant e_{len})}\ (\textsc{array arith})$$

Dependent Types in Imperative Languages

Týr[23]

- Augments LLVM IR with dependent pointer types
- Uses programmer annotations insert run-time bounds checks
- LLVM optimizations remove always-true checks; error if always-false

[23] De Araújo, Moreira, and Machado, "Týr", 2016.

[24] Ruef et al., "Checked C for Safety, Gradually", 2017.

[25] Protzenko et al., "Verified Low-level Programming Embedded in F*", 2017.

Týr[23]

- Augments LLVM IR with dependent pointer types
- Uses programmer annotations insert run-time bounds checks
- LLVM optimizations remove always-true checks; error if always-false

Checked C[24]

- Extend C with two *checked pointer types*: `_Ptr<T>` and `_Array_ptr<T>`
- Associated bounds expressions indicating where bounds are stored
- Isolate (un)safe code with *checked code regions*

[23] De Araújo, Moreira, and Machado, "Týr", 2016.

[24] Ruef et al., "Checked C for Safety, Gradually", 2017.

[25] Protzenko et al., "Verified Low-level Programming Embedded in F*", 2017.

Dependent Types in Imperative Languages

Týr[23]

- Augments LLVM IR with dependent pointer types
- Uses programmer annotations insert run-time bounds checks
- LLVM optimizations remove always-true checks; error if always-false

Checked C[24]

- Extend C with two *checked pointer types*: `_Ptr<T>` and `_Array_ptr<T>`
- Associated bounds expressions indicating where bounds are stored
- Isolate (un)safe code with *checked code regions*

Low*[25]

- DSL for verified, efficient low-level programming in F*
- Write F* syntax against library modelling lower-level view of C memory

[23] De Araújo, Moreira, and Machado, "Týr", 2016.

[24] Ruef et al., "Checked C for Safety, Gradually", 2017.

[25] Protzenko et al., "Verified Low-level Programming Embedded in F*", 2017.

Quick Spatial Recap

Spatial Safety

- Arrays and pointers

Quick Spatial Recap

Spatial Safety

- Arrays and pointers
- Fat pointers

Quick Spatial Recap

Spatial Safety

- Arrays and pointers
- Fat pointers
- Referent objects

Quick Spatial Recap

Spatial Safety

- Arrays and pointers
- Fat pointers
- Referent objects
- Dependent types

Outline

Temporal Safety

Prevent accessing object that has been previously deallocated

Temporal Safety

Prevent accessing object that has been previously deallocated

- Capabilities and locks
- Effects and regions
- Linear types and ownership

Temporal Safety Example

```
int attach(struct sock *sk) {
    if (sk->bad) {
        free(sk); return 1;
    }
    return 0;
}
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    while (n->try) {
        sock = malloc_sock(n->info);
        if (attach(sock)){
            //sock = NULL;
            break;
        }
    }
    if (sock) free(sock);
}
```

Temporal Safety Example

```
int attach(struct sock *sk) {
    if (sk->bad) {
        free(sk); return 1;
    }
    return 0;
}
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    while (n->try) {
        sock = malloc_sock(n->info);
        if (attach(sock)){
               //sock = NULL;
            break;
        }
    }
    if (sock) free(sock);
}
```

Temporal Safety Example

```
int attach(struct sock *sk) {
    if (sk->bad) {
        free(sk); return 1;
    }
    return 0;
}
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    while (n->try) {
        sock = malloc_sock(n->info);
        if (attach(sock)){
            //sock = NULL;
            break;
        }
    }
    if (sock) free(sock);
}
```

Temporal Safety Example

```
int attach(struct sock *sk) {
    if (sk->bad) {
        free(sk); return 1;
    }
    return 0;
}
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    while (n->try) {
        sock = malloc_sock(n->info);
        if (attach(sock)){
              //sock = NULL;
            break;
        }
    }
    if (sock) free(sock);
}
```

Temporal Safety Example

```
int attach(struct sock *sk) {
    if (sk->bad) {
        free(sk); return 1;
    }
    return 0;
}
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    while (n->try) {
        sock = malloc_sock(n->info);
        if (attach(sock)){
              //sock = NULL;
            break;
        }
    }
    if (sock) free(sock);
}
```

Temporal Safety Example

```
int attach(struct sock *sk) {
    if (sk->bad) {
        free(sk); return 1;
    }
    return 0;
}
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    while (n->try) {
        sock = malloc_sock(n->info);
        if (attach(sock)){
            //sock = NULL;
            break;
        }
    }
    if (sock) free(sock);
}
```

Temporal Safety Example

```
int attach(struct sock *sk) {
    if (sk->bad) {
        free(sk); return 1;
    }
    return 0;
}
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    while (n->try) {
        sock = malloc_sock(n->info);
        if (attach(sock)){
            //sock = NULL;
            break;
        }
    }
    if (sock) free(sock);
}
```

Temporal Safety Example

```
int attach(struct sock *sk) {
    if (sk->bad) {
        free(sk); return 1;
    }
    return 0;
}
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    while (n->try) {
        sock = malloc_sock(n->info);
        if (attach(sock)){
            //sock = NULL;
            break;
        }
    }
    if (sock) free(sock);
}
```

Temporal Safety Example

```
int attach(struct sock *sk) {
    if (sk->bad) {
        free(sk); return 1;
    }
    return 0;
}
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    while (n->try) {
        sock = malloc_sock(n->info);
        if (attach(sock)){
            //sock = NULL;
            break;
        }
    }
    if (sock) free(sock);
}
```

Temporal Safety Example

```c
int attach(struct sock *sk) {
    if (sk->bad) {
        free(sk); return 1;
    }
    return 0;
}
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    while (n->try) {
        sock = malloc_sock(n->info);
        if (attach(sock)){
               //sock = NULL;
            break;
        }
    }
    if (sock) free(sock);
}
```

Temporal Safety Example

```
int attach(struct sock *sk) {
    if (sk->bad) {
        free(sk); return 1;
    }
    return 0;
}
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    while (n->try) {
        sock = malloc_sock(n->info);
        if (attach(sock)){
                //sock = NULL;
            break;
        }
    }
    if (sock) free(sock);
}
```

Temporal Safety Example

```
int attach(struct sock *sk) {
    if (sk->bad) {
        free(sk); return 1;
    }
    return 0;
}
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    while (n->try) {
        sock = malloc_sock(n->info);
        if (attach(sock)){
                //sock = NULL;
            break;
        }
    }
    if (sock) free(sock);
}
```

Temporal Safety Example

```
int attach(struct sock *sk) {
    if (sk->bad) {
        free(sk); return 1;
    }
    return 0;
}
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    while (n->try) {
        sock = malloc_sock(n->info);
        if (attach(sock)){
               //sock = NULL;
            break;
        }
    }
    if (sock) free(sock);
}
```

Potential pointer dereference problems:

- Double frees
- Dangling pointers

Temporal Safety Example

```
int attach(struct sock *sk) {
    if (sk->bad) {
        free(sk); return 1;
    }
    return 0;
}
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    while (n->try) {
        sock = malloc_sock(n->info);
        if (attach(sock)){
                //sock = NULL;
            break;
        }
    }
    if (sock) free(sock);
}
```

Potential pointer dereference problems:

- Double frees
- Dangling pointers

A Real Bug

- Linux Kernel in `ipc/mqueue.c`
- July 2017
- `https://bugzilla.redhat.com/show_bug.cgi?id=1470659`

Temporal Safety Example

```
int attach(struct sock *sk) {
    if (sk->bad) {
        free(sk); return 1;
    }
    return 0;
}
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    while (n->try) {
        sock = malloc_sock(n->info);
        if (attach(sock)){
            //sock = NULL;
            break;
        }
    }
    if (sock) free(sock);
}
```

Potential pointer dereference problems:

- Double frees
- Dangling pointers

A Real Bug

- Linux Kernel in `ipc/mqueue.c`
- July 2017
- `https://bugzilla.redhat.com/show_bug.cgi?id=1470659`

Goals:

- Good: Detecting use-after-free
- Better: Eliminating free entirely

A Comment on Garbage Collection

Garbage collection

[26] Boehm and Weiser, "Garbage Collection in an Uncooperative Environment", 1988.

A Comment on Garbage Collection

Garbage collection

- Relinquish control of object location and layout to runtime

[26] Boehm and Weiser, "Garbage Collection in an Uncooperative Environment", 1988.

A Comment on Garbage Collection

Garbage collection

- Relinquish control of object location and layout to runtime
- Complete temporal safety, but...

[26] Boehm and Weiser, "Garbage Collection in an Uncooperative Environment", 1988.

A Comment on Garbage Collection

Garbage collection

- Relinquish control of object location and layout to runtime
- Complete temporal safety, but...
  - Non-zero overhead

[26] Boehm and Weiser, "Garbage Collection in an Uncooperative Environment", 1988.

A Comment on Garbage Collection

Garbage collection

- Relinquish control of object location and layout to runtime
- Complete temporal safety, but...
    - Non-zero overhead
    - Drag

[26] Boehm and Weiser, "Garbage Collection in an Uncooperative Environment", 1988.

A Comment on Garbage Collection

Garbage collection

- Relinquish control of object location and layout to runtime
- Complete temporal safety, but...
  - Non-zero overhead
  - Drag
  - Loss of real-time guarantees/predictability

[26] Boehm and Weiser, "Garbage Collection in an Uncooperative Environment", 1988.

A Comment on Garbage Collection

Garbage collection

- Relinquish control of object location and layout to runtime
- Complete temporal safety, but...
  - Non-zero overhead
  - Drag
  - Loss of real-time guarantees/predictability
  - Reduced reference locality, increased page fault/cache miss rates

[26] Boehm and Weiser, "Garbage Collection in an Uncooperative Environment", 1988.

A Comment on Garbage Collection

Garbage collection

- Relinquish control of object location and layout to runtime
- Complete temporal safety, but...
  - Non-zero overhead
  - Drag
  - Loss of real-time guarantees/predictability
  - Reduced reference locality, increased page fault/cache miss rates
- Some spatial approaches (e.g. Fail-Safe C, CCured) use Boehm-Demers-Weister[26]

[26] Boehm and Weiser, "Garbage Collection in an Uncooperative Environment", 1988.

Outline

Capabilities and Locks

[27] Nagarakatte, Zhao, Milo M.K. Martin, et al., “CETS”, 2010.

[28] Nagarakatte, M. M. K. Martin, and Zdancewic, “Everything You Want to Know About Pointer-Based Checking”, 2015.

[29] Simpson and Barua, “MemSafe”, 2013.

Capabilities and Locks

- SafeC, MSCC
  - Unique capability associated with each memory block
  - Stored in capability store, marked invalid on free
  - Check if pointer's capability copy is still valid on dereference

[27] Nagarakatte, Zhao, Milo M.K. Martin, et al., "CETS", 2010.

[28] Nagarakatte, M. M. K. Martin, and Zdancewic, "Everything You Want to Know About Pointer-Based Checking", 2015.

[29] Simpson and Barua, "MemSafe", 2013.

Capabilities and Locks

- SafeC, MSCC
  - Unique capability associated with each memory block
  - Stored in capability store, marked invalid on free
  - Check if pointer's capability copy is still valid on dereference
- CETS[27,28]
  - Each allocation has unique (never reused) key and lock address
  - Freeing allocated object changes value at lock location, so key and lock value don't match

[27] Nagarakatte, Zhao, Milo M.K. Martin, et al., "CETS", 2010.

[28] Nagarakatte, M. M. K. Martin, and Zdancewic, "Everything You Want to Know About Pointer-Based Checking", 2015.

[29] Simpson and Barua, "MemSafe", 2013.

Capabilities and Locks

- SafeC, MSCC
  - Unique capability associated with each memory block
  - Stored in capability store, marked invalid on free
  - Check if pointer's capability copy is still valid on dereference
- CETS[27,28]
  - Each allocation has unique (never reused) key and lock address
  - Freeing allocated object changes value at lock location, so key and lock value don't match
- Memsafe[29]
  - Set bounds of deallocated pointer to invalid value

[27] Nagarakatte, Zhao, Milo M.K. Martin, et al., "CETS", 2010.

[28] Nagarakatte, M. M. K. Martin, and Zdancewic, "Everything You Want to Know About Pointer-Based Checking", 2015.

[29] Simpson and Barua, "MemSafe", 2013.

Lock Example

```
int attach(struct sock *sk,
        key_t sk_key,
        lock_t *sk_lock_addr) {
    if (sk_key != *sk_lock_addr)
      abort();
    if (sk->bad) {
        if (Freeable_ptrs_map.lookup(sk_key) != sk)
          abort();
        free(sk);
        *sk_lock_addr = INVALID_KEY;
        deallocate_lock(sk_lock_addr);
        return 1;
    }
    return 0;
}
```

```
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    key_t sock_key;
    lock_t *sock_lock_addr;
    while (n->try) {
        sock = malloc_sock(n->info);
        sock_key = Next_key++;
        sock_lock_addr = allocate_lock();
        *(sock_lock_addr) = sock_key;
        Freeable_ptrs_map.insert(sock_key, sock);
        if (attach(sock)){
            break;
        }
    }
    if (sock) {
        if (Freeable_ptrs_map.lookup(sock_key) != sock)
          abort();
        free(sock);
        *(sock_lock_addr) = INVALID_KEY;
        deallocate_lock(sock_lock_addr);
    }
}
```

Lock Example

```
int attach(struct sock *sk,
      key_t sk_key,
      lock_t *sk_lock_addr) {
    if (sk_key != *sk_lock_addr)
      abort();
    if (sk->bad) {
        if (Freeable_ptrs_map.lookup(sk_key) != sk)
          abort();
        free(sk);
        *sk_lock_addr = INVALID_KEY;
        deallocate_lock(sk_lock_addr);
        return 1;
    }
    return 0;
}
```

```
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    key_t sock_key;
    lock_t *sock_lock_addr;
    while (n->try) {
        sock = malloc_sock(n->info);
        sock_key = Next_key++;
        sock_lock_addr = allocate_lock();
        *(sock_lock_addr) = sock_key;
        Freeable_ptrs_map.insert(sock_key, sock);
        if (attach(sock)){
            break;
        }
    }
    if (sock) {
        if (Freeable_ptrs_map.lookup(sock_key) != sock)
          abort();
        free(sock);
        *(sock_lock_addr) = INVALID_KEY;
        deallocate_lock(sock_lock_addr);
    }
}
```

Lock Example

```
int attach(struct sock *sk,
      key_t sk_key,
      lock_t *sk_lock_addr) {
    if (sk_key != *sk_lock_addr)
      abort();
    if (sk->bad) {
        if (Freeable_ptrs_map.lookup(sk_key) != sk)
          abort();
        free(sk);
        *sk_lock_addr = INVALID_KEY;
        deallocate_lock(sk_lock_addr);
        return 1;
    }
    return 0;
}
```

```
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    key_t sock_key;
    lock_t *sock_lock_addr;
    while (n->try) {
        sock = malloc_sock(n->info);
        sock_key = Next_key++;
        sock_lock_addr = allocate_lock();
        *(sock_lock_addr) = sock_key;
        Freeable_ptrs_map.insert(sock_key, sock);
        if (attach(sock)){
            break;
        }
    }
    if (sock) {
        if (Freeable_ptrs_map.lookup(sock_key) != sock)
          abort();
        free(sock);
        *(sock_lock_addr) = INVALID_KEY;
        deallocate_lock(sock_lock_addr);
    }
}
```

Lock Example

```
int attach(struct sock *sk,
      key_t sk_key,
      lock_t *sk_lock_addr) {
    if (sk_key != *sk_lock_addr)
      abort();
    if (sk->bad) {
        if (Freeable_ptrs_map.lookup(sk_key) != sk)
          abort();
        free(sk);
        *sk_lock_addr = INVALID_KEY;
        deallocate_lock(sk_lock_addr);
        return 1;
    }
    return 0;
}
```

```
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    key_t sock_key;
    lock_t *sock_lock_addr;
    while (n->try) {
        sock = malloc_sock(n->info);
        sock_key = Next_key++;
        sock_lock_addr = allocate_lock();
        *(sock_lock_addr) = sock_key;
        Freeable_ptrs_map.insert(sock_key, sock);
        if (attach(sock)){
            break;
        }
    }
    if (sock) {
        if (Freeable_ptrs_map.lookup(sock_key) != sock)
          abort();
        free(sock);
        *(sock_lock_addr) = INVALID_KEY;
        deallocate_lock(sock_lock_addr);
    }
}
```

Lock Example

```
int attach(struct sock *sk,
      key_t sk_key,
      lock_t *sk_lock_addr) {
    if (sk_key != *sk_lock_addr)
      abort();
    if (sk->bad) {
        if (Freeable_ptrs_map.lookup(sk_key) != sk)
          abort();
        free(sk);
        *sk_lock_addr = INVALID_KEY;
        deallocate_lock(sk_lock_addr);
        return 1;
    }
    return 0;
}
```

```
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    key_t sock_key;
    lock_t *sock_lock_addr;
    while (n->try) {
        sock = malloc_sock(n->info);
        sock_key = Next_key++;
        sock_lock_addr = allocate_lock();
        *(sock_lock_addr) = sock_key;
        Freeable_ptrs_map.insert(sock_key, sock);
        if (attach(sock)){
            break;
        }
    }
    if (sock) {
        if (Freeable_ptrs_map.lookup(sock_key) != sock)
          abort();
        free(sock);
        *(sock_lock_addr) = INVALID_KEY;
        deallocate_lock(sock_lock_addr);
    }
}
```

Lock Example

```
int attach(struct sock *sk,
      key_t sk_key,
      lock_t *sk_lock_addr) {
    if (sk_key != *sk_lock_addr)
      abort();
    if (sk->bad) {
        if (Freeable_ptrs_map.lookup(sk_key) != sk)
          abort();
        free(sk);
        *sk_lock_addr = INVALID_KEY;
        deallocate_lock(sk_lock_addr);
        return 1;
    }
    return 0;
}
```

```
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    key_t sock_key;
    lock_t *sock_lock_addr;
    while (n->try) {
        sock = malloc_sock(n->info);
        sock_key = Next_key++;
        sock_lock_addr = allocate_lock();
        *(sock_lock_addr) = sock_key;
        Freeable_ptrs_map.insert(sock_key, sock);
        if (attach(sock)){
            break;
        }
    }
    if (sock) {
        if (Freeable_ptrs_map.lookup(sock_key) != sock)
          abort();
        free(sock);
        *(sock_lock_addr) = INVALID_KEY;
        deallocate_lock(sock_lock_addr);
    }
}
```

Outline

Effects

Effect types describe the *effects* of the computation leading to a value[30]

- Opening a file
- Modifying an object

[30]Pierce, *Advanced topics in types and programming languages*, 2005.

Effects

Fluent Languages,[31] MFX[32]

- Mix functional and imperative languages

[31] David K. Gifford and John M. Lucassen, “Integrating functional and imperative programming”, 1986.

[32] J. M. Lucassen and D. K. Gifford, “Polymorphic Effect Systems”, 1988.

Effects

Fluent Languages,[31] MFX[32]

- Mix functional and imperative languages
- Every expression has a effect class, restricting which subroutines or sublanguage features it may use

[31] David K. Gifford and John M. Lucassen, “Integrating functional and imperative programming”, 1986.

[32] J. M. Lucassen and D. K. Gifford, “Polymorphic Effect Systems”, 1988.

Effects

Fluent Languages,[31] MFX[32]

- Mix functional and imperative languages
- Every expression has a effect class, restricting which subroutines or sublanguage features it may use
- Identify referentially transparent expressions via a side effect lattice

[31] David K. Gifford and John M. Lucassen, "Integrating functional and imperative programming", 1986.

[32] J. M. Lucassen and D. K. Gifford, "Polymorphic Effect Systems", 1988.

Effects

Fluent Languages,[31] MFX[32]

- Mix functional and imperative languages
- Every expression has a effect class, restricting which subroutines or sublanguage features it may use
- Identify referentially transparent expressions via a side effect lattice
- Effect masking: inference to delimit regions of memory and their lifetimes

[31] David K. Gifford and John M. Lucassen, "Integrating functional and imperative programming", 1986.
[32] J. M. Lucassen and D. K. Gifford, "Polymorphic Effect Systems", 1988.

Effects

Fluent Languages,[31] MFX[32]

- Mix functional and imperative languages
- Every expression has a effect class, restricting which subroutines or sublanguage features it may use
- Identify referentially transparent expressions via a side effect lattice
- Effect masking: inference to delimit regions of memory and their lifetimes

[31] David K. Gifford and John M. Lucassen, "Integrating functional and imperative programming", 1986.
[32] J. M. Lucassen and D. K. Gifford, "Polymorphic Effect Systems", 1988.

Effects

Fluent Languages,[31] MFX[32]

- Mix functional and imperative languages
- Every expression has a effect class, restricting which subroutines or sublanguage features it may use
- Identify referentially transparent expressions via a side effect lattice
- Effect masking: inference to delimit regions of memory and their lifetimes

$\{W\ R^?\ A^?\}$ Procedure

$\{R\ A^?\}$ Observer

$\{A\}$ Function

$\{\}$ Pure

[31] David K. Gifford and John M. Lucassen, "Integrating functional and imperative programming", 1986.

[32] J. M. Lucassen and D. K. Gifford, "Polymorphic Effect Systems", 1988.

Effects

Fluent Languages,[31] MFX[32]

- Mix functional and imperative languages
- Every expression has a effect class, restricting which subroutines or sublanguage features it may use
- Identify referentially transparent expressions via a side effect lattice
- Effect masking: inference to delimit regions of memory and their lifetimes

$\{W\ R^?\ A^?\}$ Procedure

$\{R\ A^?\}$ Observer

$\{A\}$ Function

$\{\}$ Pure

E.g.

- `update` is Procedure
- `nth` is Observer
- `arrayCreate` is Function
- `length` is Pure

[31] David K. Gifford and John M. Lucassen, "Integrating functional and imperative programming", 1986.

[32] J. M. Lucassen and D. K. Gifford, "Polymorphic Effect Systems", 1988.

Effects

Type and Effect Systems[33]

- Extend the simply-typed lambda calculus with annotations

[33] F. Nielson and H. R. Nielson, "Type and Effect Systems", 1999.

Effects

Type and Effect Systems[33]

- Extend the simply-typed lambda calculus with annotations
- Various forms of analyses, incl. Side Effect Analysis and Region Inference

[33] F. Nielson and H. R. Nielson, “Type and Effect Systems”, 1999.

Effects

Type and Effect Systems[33]

- Extend the simply-typed lambda calculus with annotations
- Various forms of analyses, incl. Side Effect Analysis and Region Inference

[33] F. Nielson and H. R. Nielson, "Type and Effect Systems", 1999.

Effects

Type and Effect Systems[33]

- Extend the simply-typed lambda calculus with annotations
- Various forms of analyses, incl. Side Effect Analysis and Region Inference

$$
\begin{aligned}
\text{Exprs} \quad & e ::= c \mid x \mid \mathtt{fn}_\pi\ x \Rightarrow e \mid e_1\ e_2 \\
\text{Types} \quad & \tau ::= \mathtt{int} \mid \mathtt{bool} \mid \tau_1 \to \tau_2
\end{aligned}
$$

[33] F. Nielson and H. R. Nielson, "Type and Effect Systems", 1999.

Effects

Type and Effect Systems[33]

- Extend the simply-typed lambda calculus with annotations
- Various forms of analyses, incl. Side Effect Analysis and Region Inference

$$
\begin{array}{rl}
\text{Exprs} & e ::= c \mid x \mid \mathtt{fn}_\pi\ x \Rightarrow e \mid e_1\ e_2 \mid \mathtt{new}_\pi\ x := e_1\ \mathtt{in}\ e_2 \\
\text{Types} & \tau ::= \mathtt{int} \mid \mathtt{bool} \mid \tau_1 \to \tau_2
\end{array}
$$

[33] F. Nielson and H. R. Nielson, "Type and Effect Systems", 1999.

Effects

Type and Effect Systems[33]

- Extend the simply-typed lambda calculus with annotations
- Various forms of analyses, incl. Side Effect Analysis and Region Inference

$$
\begin{array}{rl}
\text{Exprs} & e ::= c \mid x \mid \texttt{fn}_\pi\ x \Rightarrow e \mid e_1\ e_2 \mid \texttt{new}_\pi\ x := e_1\ \texttt{in}\ e_2 \mid !x \\
\text{Types} & \tau ::= \texttt{int} \mid \texttt{bool} \mid \tau_1 \rightarrow \tau_2
\end{array}
$$

[33] F. Nielson and H. R. Nielson, "Type and Effect Systems", 1999.

Effects

Type and Effect Systems[33]

- Extend the simply-typed lambda calculus with annotations
- Various forms of analyses, incl. Side Effect Analysis and Region Inference

$$\begin{array}{rl} \text{Exprs} & e ::= c \mid x \mid \mathtt{fn}_\pi\ x \Rightarrow e \mid e_1\ e_2 \mid \mathtt{new}_\pi\ x := e_1\ \mathtt{in}\ e_2 \mid !x \mid x := e \\ \text{Types} & \tau ::= \mathtt{int} \mid \mathtt{bool} \mid \tau_1 \rightarrow \tau_2 \end{array}$$

[33] F. Nielson and H. R. Nielson, "Type and Effect Systems", 1999.

Effects

Type and Effect Systems[33]

- Extend the simply-typed lambda calculus with annotations
- Various forms of analyses, incl. Side Effect Analysis and Region Inference

$$
\begin{array}{rl}
\text{Exprs} & e ::= c \mid x \mid \mathtt{fn}_\pi\ x \ \Rightarrow e \mid e_1\ e_2 \mid \mathtt{new}_\pi\ x\ :=\ e_1\ \mathtt{in}\ e_2 \mid\ !x \mid x := e \\
\text{Types} & \tau ::= \mathtt{int} \mid \mathtt{bool} \mid \tau_1 \rightarrow \tau_2 \mid \tau\ \mathtt{ref}
\end{array}
$$

[33] F. Nielson and H. R. Nielson, "Type and Effect Systems", 1999.

Effects

Type and Effect Systems[33]

- Extend the simply-typed lambda calculus with annotations
- Various forms of analyses, incl. Side Effect Analysis and Region Inference

$$\begin{array}{ll} \text{Exprs} & e ::= c \mid x \mid \texttt{fn}_\pi\ x\ \Rightarrow e \mid e_1\ e_2 \mid \texttt{new}_\pi\ x\ :=\ e_1\ \texttt{in}\ e_2 \mid\ !x \mid x := e \\ \text{Types} & \tau ::= \texttt{int} \mid \texttt{bool} \mid \tau_1 \xrightarrow{\phi} \tau_2 \mid \tau\ \texttt{ref}\ \varrho \end{array}$$

[33] F. Nielson and H. R. Nielson, "Type and Effect Systems", 1999.

Effects

Type and Effect Systems[33]

- Extend the simply-typed lambda calculus with annotations
- Various forms of analyses, incl. Side Effect Analysis and Region Inference

$$
\begin{array}{rl}
\text{Exprs} & e ::= c \mid x \mid \texttt{fn}_\pi\ x\ \Rightarrow e \mid e_1\ e_2 \mid \texttt{new}_\pi\ x\ :=\ e_1\ \texttt{in}\ e_2 \mid\ !x \mid x := e \\
\text{Types} & \tau ::= \texttt{int} \mid \texttt{bool} \mid \tau_1 \xrightarrow{\phi} \tau_2 \mid \tau\ \texttt{ref}\ \varrho \\
\text{Effect} & \phi ::= \{!\pi\} \mid \{\pi\ :=\} \mid \{\texttt{new}\ \pi\} \mid \phi_1 \cup \phi_2 \mid \varnothing
\end{array}
$$

[33] F. Nielson and H. R. Nielson, "Type and Effect Systems", 1999.

Effects

Type and Effect Systems[33]

- Extend the simply-typed lambda calculus with annotations
- Various forms of analyses, incl. Side Effect Analysis and Region Inference

$$
\begin{array}{rl}
\text{Exprs} & e ::= c \mid x \mid \mathtt{fn}_\pi\ x \Rightarrow e \mid e_1\ e_2 \mid \mathtt{new}_\pi\ x := e_1\ \mathtt{in}\ e_2 \mid\ !x \mid x := e \\
\text{Types} & \tau ::= \mathtt{int} \mid \mathtt{bool} \mid \tau_1 \xrightarrow{\phi} \tau_2 \mid \tau\ \mathtt{ref}\ \varrho \\
\text{Effect} & \phi ::= \{!\pi\} \mid \{\pi :=\} \mid \{\mathtt{new}\ \pi\} \mid \phi_1 \cup \phi_2 \mid \emptyset \\
\text{PrgPts} & \varrho ::= \{\pi\} \mid \varrho_1 \cup \varrho_2 \mid \emptyset
\end{array}
$$

[33] F. Nielson and H. R. Nielson, "Type and Effect Systems", 1999.

Effects

Type and Effect Systems[33]

- Extend the simply-typed lambda calculus with annotations
- Various forms of analyses, incl. Side Effect Analysis and Region Inference

$$
\begin{array}{rl}
\text{Exprs} & e ::= c \mid x \mid \mathtt{fn}_\pi\ x \Rightarrow e \mid e_1\ e_2 \mid \mathtt{new}_\pi\ x := e_1\ \mathtt{in}\ e_2 \mid !x \mid x := e \\
\text{Types} & \tau ::= \mathtt{int} \mid \mathtt{bool} \mid \tau_1 \xrightarrow{\phi} \tau_2 \mid \tau\ \mathtt{ref}\ \varrho \\
\text{Region} & \phi ::= \{!\pi\} \mid \{\pi :=\} \mid \{\mathtt{new}\ \pi\} \mid \phi_1 \cup \phi_2 \mid \emptyset \\
\text{PrgPts} & \varrho ::= \{\pi\} \mid \varrho_1 \cup \varrho_2 \mid \emptyset
\end{array}
$$

[33] F. Nielson and H. R. Nielson, "Type and Effect Systems", 1999.

Effects

Type and Effect Systems[33]

- Extend the simply-typed lambda calculus with annotations
- Various forms of analyses, incl. Side Effect Analysis and Region Inference

$$
\begin{array}{rl}
\text{Exprs} & e ::= c \mid x \mid \mathtt{fn}_\pi\ x \Rightarrow e \mid e_1\ e_2 \mid \mathtt{new}_\pi\ x := e_1\ \mathtt{in}\ e_2 \mid !x \mid x := e \\
\text{Types} & \tau ::= \mathtt{int} \mid \mathtt{bool} \mid \tau_1 \xrightarrow{\phi} \tau_2 \mid \tau\ \mathtt{ref}\ \varrho \\
\text{Region} & \phi ::= \{!\pi\} \mid \{\pi :=\} \mid \{\mathtt{new}\ \pi\} \mid \phi_1 \cup \phi_2 \mid \varnothing \\
\text{PrgPts} & \varrho ::= \{\pi\} \mid \varrho_1 \cup \varrho_2 \mid \varnothing
\end{array}
$$

$$\Gamma \vdash e : \tau \ \&\ \phi$$

[33] F. Nielson and H. R. Nielson, "Type and Effect Systems", 1999.

Region-Based Memory Managememt

Regions[34]

- Divide heap into stack of sub-heaps (i.e. regions)

[34] Tofte and Talpin, "Region-Based Memory Management", 1997.

Region-Based Memory Managememt

Regions[34]

- Divide heap into stack of sub-heaps (i.e. regions)
- Regions grow on individual allocation; entire region deallocated

[34] Tofte and Talpin, "Region-Based Memory Management", 1997.

Region-Based Memory Managememt

Regions[34]

- Divide heap into stack of sub-heaps (i.e. regions)
- Regions grow on individual allocation; entire region deallocated
- Explicit region polymorphism

[34] Tofte and Talpin, "Region-Based Memory Management", 1997.

Region-Based Memory Managememt

Regions[34]

- Divide heap into stack of sub-heaps (i.e. regions)
- Regions grow on individual allocation; entire region deallocated
- Explicit region polymorphism
- Sound type system guarantees safety of deallocations

[34] Tofte and Talpin, "Region-Based Memory Management", 1997.

Region-Based Memory Managememt

Regions[34]

- Divide heap into stack of sub-heaps (i.e. regions)
- Regions grow on individual allocation; entire region deallocated
- Explicit region polymorphism
- Sound type system guarantees safety of deallocations
- Region inference identifies:

[34] Tofte and Talpin, "Region-Based Memory Management", 1997.

Region-Based Memory Managememt

Regions[34]

- Divide heap into stack of sub-heaps (i.e. regions)
- Regions grow on individual allocation; entire region deallocated
- Explicit region polymorphism
- Sound type system guarantees safety of deallocations
- Region inference identifies:
  - Points where entire regions are allocated and deallocated

[34] Tofte and Talpin, "Region-Based Memory Management", 1997.

Region-Based Memory Managememt

Regions[34]

- Divide heap into stack of sub-heaps (i.e. regions)
- Regions grow on individual allocation; entire region deallocated
- Explicit region polymorphism
- Sound type system guarantees safety of deallocations
- Region inference identifies:
  - Points where entire regions are allocated and deallocated
  - Into which region values should go

[34] Tofte and Talpin, "Region-Based Memory Management", 1997.

Region-Based Memory Managememt

Regions[34]

- Divide heap into stack of sub-heaps (i.e. regions)
- Regions grow on individual allocation; entire region deallocated
- Explicit region polymorphism
- Sound type system guarantees safety of deallocations
- Region inference identifies:
  - Points where entire regions are allocated and deallocated
  - Into which region values should go
- Unreasonable object lifetimes due to LIFO ordering of region lifetimes

[34] Tofte and Talpin, "Region-Based Memory Management", 1997.

Early Use of Regions

Capability Calculus[35]

- Arbitrarily-ordered region allocation/deallocation, via capability tracking
- Capability: set of regions presently valid to access

[35] Crary, Walker, and Morrisett, "Typed Memory Management in a Calculus of Capabilities", 1999.

[36] Gay and Aiken, "Language Support for Regions", 2001.

[37] Berger, Zorn, and McKinley, "OOPSLA 2002", 2002.

[38] Kowshik, Dhurjati, and Adve, "Ensuring Code Safety Without Runtime Checks for Real-time Control Systems", 2002.

[39] Dhurjati, Kowshik, et al., "Memory safety without runtime checks or garbage collection", 2003.

Early Use of Regions

Capability Calculus[35]

- Arbitrarily-ordered region allocation/deallocation, via capability tracking
- Capability: set of regions presently valid to access

Also see:

- RC[36]
- Reaps[37]
- Control-C[38] and Type Homogeneity[39]

[35] Crary, Walker, and Morrisett, "Typed Memory Management in a Calculus of Capabilities", 1999.

[36] Gay and Aiken, "Language Support for Regions", 2001.

[37] Berger, Zorn, and McKinley, "OOPSLA 2002", 2002.

[38] Kowshik, Dhurjati, and Adve, "Ensuring Code Safety Without Runtime Checks for Real-time Control Systems", 2002.

[39] Dhurjati, Kowshik, et al., "Memory safety without runtime checks or garbage collection", 2003.

Regions in Cyclone

Cyclone,[40] again!

- Three region types:

[40] Grossman et al., "Region-based Memory Management in Cyclone", 2002.

Regions in Cyclone

Cyclone,[40] again!

- Three region types:
  - single heap region

[40] Grossman et al., "Region-based Memory Management in Cyclone", 2002.

Regions in Cyclone

Cyclone,[40] again!

- Three region types:
  - single heap region
  - stack regions

[40] Grossman et al., “Region-based Memory Management in Cyclone”, 2002.

Regions in Cyclone

Cyclone,[40] again!

- Three region types:
  - single heap region
  - stack regions
  - dynamic regions

[40] Grossman et al., "Region-based Memory Management in Cyclone", 2002.

Regions in Cyclone

Cyclone,[40] again!

- Three region types:
  - single heap region
  - stack regions
  - dynamic regions
- Lifetime subtyping: region A $<:$ region B $\Leftrightarrow$ region A outlives region B

[40] Grossman et al., "Region-based Memory Management in Cyclone", 2002.

Regions in Cyclone

Cyclone,[40] again!

- Three region types:
  - single heap region
  - stack regions
  - dynamic regions
- Lifetime subtyping: region A $<:$ region B $\Leftrightarrow$ region A outlives region B
- Sane defaults by inferring region annotations on pointer types

[40] Grossman et al., "Region-based Memory Management in Cyclone", 2002.

Regions in Cyclone

- Pointer can escape scope of their regions

Regions in Cyclone

- Pointer can escape scope of their regions
- Type system tracks capability (live regions set)

Regions in Cyclone

- Pointer can escape scope of their regions
- Type system tracks capability (live regions set)
- Check needed liveness on pointer dereference

Regions in Cyclone

- Pointer can escape scope of their regions
- Type system tracks capability (live regions set)
- Check needed liveness on pointer dereference
- Function effect: set of regions it might access

Regions in Cyclone

- Pointer can escape scope of their regions
- Type system tracks capability (live regions set)
- Check needed liveness on pointer dereference
- Function effect: set of regions it might access
- Calculate function's effect from prototype alone

Regions in Cyclone

- Pointer can escape scope of their regions
- Type system tracks capability (live regions set)
- Check needed liveness on pointer dereference
- Function effect: set of regions it might access
- Calculate function's effect from prototype alone

Regions in Cyclone

- Pointer can escape scope of their regions
- Type system tracks capability (live regions set)
- Check needed liveness on pointer dereference
- Function effect: set of regions it might access
- Calculate function's effect from prototype alone

```
struct Set<α, ρ, ϵ>
{
  list_t<α, ρ> elts;
  int (*cmp)(α,α; ϵ);
}
```

Regions in Cyclone

- Pointer can escape scope of their regions
- Type system tracks capability (live regions set)
- Check needed liveness on pointer dereference
- Function effect: set of regions it might access
- Calculate function's effect from prototype alone

```
struct Set<α, ρ, ε>
{
  list_t<α, ρ> elts;
  int (*cmp)(α,α; ε);
}
```

```
struct Set<α, ρ>
{
  list_t<α, ρ> elts;
  int (*cmp)(α,α; regions_of(α));
}
```

Some Cyclone Abstract Syntax

kinds	κ	$::= T \mid R$
type and region vars	α, ρ	
region sets	ϵ	$::= \emptyset \mid \alpha \mid \epsilon_1 \cup \epsilon_2$
region constraints	γ	$::= \emptyset \mid \gamma, \epsilon <: \rho$
constructors	τ	$::= \alpha \mid \mathsf{int} \mid \tau_1 \xrightarrow{\epsilon} \tau_2 \mid \tau @ \rho \mid \mathsf{handle}(\rho) \mid \forall \alpha : \kappa \rhd \gamma.\tau \mid \ldots$
expressions	e	$::= x_\rho \mid v \mid e \langle \tau \rangle \mid * e \mid \mathbf{new}(e_1) e_2 \mid e_1(e_2) \mid \& e \mid \ldots$
functions	f	$::= \rho : (\tau_1\ x_\rho) \xrightarrow{\epsilon} \tau_2 = \{s\} \mid \Lambda \alpha : \kappa \rhd \gamma.f$
statements	s	$::= e \mid s_1 ; s_2 \mid \mathbf{if}\ (e)\ s_1\ \mathbf{else}\ s_2 \mid \rho : \{\tau\ x_\rho = e;\ s\} \mid \mathbf{region}\langle\rho\rangle\ x_\rho\ s \mid \ldots$
	$\ldots$	

Example Cyclone Judgments

$\Delta; \Gamma; \gamma; \epsilon; \tau \vdash_{stmt} s$

$\Delta; \Gamma; \gamma; \epsilon \vdash e : \tau$

$\Gamma \vdash \epsilon \Rightarrow \rho$

Example Cyclone Judgments

$\Delta; \Gamma; \gamma; \epsilon; \tau \vdash_{stmt} s$ $\quad\quad$ $\Delta; \Gamma; \gamma; \epsilon \vdash e : \tau$ $\quad\quad$ $\Gamma \vdash \epsilon \Rightarrow \rho$

Δ : in-scope type/region vars

Example Cyclone Judgments

$\Delta; \Gamma; \gamma; \epsilon; \tau \vdash_{stmt} s$ $\quad\quad$ $\Delta; \Gamma; \gamma; \epsilon \vdash e \; : \; \tau$ $\quad\quad$ $\Gamma \vdash \epsilon \Rightarrow \rho$

Δ : in-scope type/region vars

Γ : mapping of in-scope vars to types

Example Cyclone Judgments

$\Delta; \Gamma; \gamma; \epsilon; \tau \vdash_{stmt} s$ $\Delta; \Gamma; \gamma; \epsilon \vdash e \ : \ \tau$ $\Gamma \vdash \epsilon \Rightarrow \rho$

Δ : in-scope type/region vars

Γ : mapping of in-scope vars to types

γ : constraints relating lifetimes

Example Cyclone Judgments

$\Delta; \Gamma; \gamma; \epsilon; \tau \vdash_{stmt} s$ $\quad$ $\Delta; \Gamma; \gamma; \epsilon \vdash e : \tau$ $\quad$ $\Gamma \vdash \epsilon \Rightarrow \rho$

- Δ : in-scope type/region vars
- Γ : mapping of in-scope vars to types
- γ : constraints relating lifetimes
- ϵ : capability, i.e. live regions

Example Cyclone Judgments

$\Delta; \Gamma; \gamma; \epsilon; \tau \vdash_{stmt} s$ $\quad$ $\Delta; \Gamma; \gamma; \epsilon \vdash e : \tau$ $\quad$ $\Gamma \vdash \epsilon \Rightarrow \rho$

- Δ : in-scope type/region vars
- Γ : mapping of in-scope vars to types
- γ : constraints relating lifetimes
- ϵ : capability, i.e. live regions

$$\frac{\gamma \vdash \epsilon \Rightarrow \rho}{\Delta; \Gamma; \gamma; \epsilon \vdash x_\rho : \Gamma(x_\rho)} \ (\textsc{var}) \qquad \frac{\Delta; \Gamma; \gamma; \epsilon \vdash e : \tau * \rho \qquad \gamma \vdash \epsilon \Rightarrow \rho}{\Delta; \Gamma; \gamma; \epsilon \vdash *e : \tau} \ (\textsc{deref})$$

Example Cyclone Judgments

$\Delta; \Gamma; \gamma; \epsilon; \tau \vdash_{stmt} s$ $\qquad \Delta; \Gamma; \gamma; \epsilon \vdash e \ : \ \tau$ $\qquad \Gamma \vdash \epsilon \Rightarrow \rho$

- Δ : in-scope type/region vars
- Γ : mapping of in-scope vars to types
- γ : constraints relating lifetimes
- ϵ : capability, i.e. live regions

$$\frac{\gamma \vdash \epsilon \Rightarrow \rho}{\Delta; \Gamma; \gamma; \epsilon \vdash x_\rho : \Gamma(x_\rho)}\ (\text{VAR}) \qquad \frac{\Delta; \Gamma; \gamma; \epsilon \vdash e : \tau * \rho \qquad \gamma \vdash \epsilon \Rightarrow \rho}{\Delta; \Gamma; \gamma; \epsilon \vdash *e : \tau}\ (\text{DEREF})$$

$$\frac{\Delta; \Gamma; \gamma; \epsilon \vdash e_1 : \tau_2 \xrightarrow{\epsilon_1} \tau \qquad \Delta; \Gamma; \gamma; \epsilon \vdash e_2 : \tau_2 \qquad \gamma \vdash \epsilon \Rightarrow \epsilon_1}{\Delta; \Gamma; \gamma; \epsilon \vdash e_1(e_2) : \tau}\ (\text{CALL})$$

Example Cyclone Judgments

$\Delta; \Gamma; \gamma; \epsilon; \tau \vdash_{stmt} s$ $\quad\quad$ $\Delta; \Gamma; \gamma; \epsilon \vdash e \ : \ \tau$ $\quad\quad$ $\Gamma \vdash \epsilon \Rightarrow \rho$

- Δ : in-scope type/region vars
- Γ : mapping of in-scope vars to types
- γ : constraints relating lifetimes
- ϵ : capability, i.e. live regions

$$\frac{\gamma \vdash \epsilon \Rightarrow \rho}{\Delta; \Gamma; \gamma; \epsilon \vdash x_\rho : \Gamma(x_\rho)} \ (\textsc{var}) \qquad \frac{\Delta; \Gamma; \gamma; \epsilon \vdash e : \tau * \rho \qquad \gamma \vdash \epsilon \Rightarrow \rho}{\Delta; \Gamma; \gamma; \epsilon \vdash *e : \tau} \ (\textsc{deref})$$

$$\frac{\Delta; \Gamma; \gamma; \epsilon \vdash e_1 : \tau_2 \xrightarrow{\epsilon_1} \tau \qquad \Delta; \Gamma; \gamma; \epsilon \vdash e_2 : \tau_2 \qquad \gamma \vdash \epsilon \Rightarrow \epsilon_1}{\Delta; \Gamma; \gamma; \epsilon \vdash e_1(e_2) : \tau} \ (\textsc{call})$$

$$\frac{\Delta; \Gamma; \gamma; \epsilon \vdash e : \forall \alpha : \kappa \rhd \gamma_1 . \tau_2 \qquad \Delta \vdash \tau : \kappa \qquad \gamma \vdash \gamma_1[\tau_1/\alpha]}{\Delta; \Gamma; \gamma; \epsilon \vdash e\langle \tau_1 \rangle : \tau_2[\tau_1/\alpha]} \ (\textsc{type-inst})$$

Cyclone Results

Soundness Theorem[41]:

- The program cannot get stuck from type errors or dangling-pointer dereferences.
- The terminating program deallocates all regions it allocates

[41] Grossman et al., *Formal Type Soundness for Cyclone's Region System*, 2001.

Cyclone Results

Soundness Theorem[41]:

- The program cannot get stuck from type errors or dangling-pointer dereferences.
- The terminating program deallocates all regions it allocates

Benchmarks

- 86 lines of region annotation-related changes across 18,000 lines (6%)
- Eliminate heap allocation entirely for web-server
- Near-zero overhead (from garbage collection and bounds checking)

[41] Grossman et al., *Formal Type Soundness for Cyclone's Region System*, 2001.

Outline

Linear Types

Linear types[42,43] ensure that every variable is used exactly *once*.

- The world is a non-duplicatable resource

[42] Girard, "Linear logic", 1987.

[43] Wadler, "Linear types can change the world", 1990.

Linear Types

Linear types[42,43] ensure that every variable is used exactly *once*.

- The world is a non-duplicatable resource
- Track proper modification of world

[42] Girard, "Linear logic", 1987.

[43] Wadler, "Linear types can change the world", 1990.

Linear types[42,43] ensure that every variable is used exactly *once*.

- The world is a non-duplicatable resource
- Track proper modification of world
- Efficiency: safe to destructively update an array

[42] Girard, "Linear logic", 1987.

[43] Wadler, "Linear types can change the world", 1990.

Linear Types

Linear types[42,43] ensure that every variable is used exactly *once*.

- The world is a non-duplicatable resource
- Track proper modification of world
- Efficiency: safe to destructively update an array
- **Memory management: can immediately collect used values**

[42] Girard, "Linear logic", 1987.

[43] Wadler, "Linear types can change the world", 1990.

Applied Linear Types

Clay[44]

- Type-theoretic basis for giving type-safe code more control over memory
- Singleton types to type check loads, coercion functions to modify values' type safely

[44] Hawblitzel et al., "Low-Level Linear Memory Management", 2004.

[45] Ennals, Sharp, and Mycroft, "Linear Types for Packet Processing", 2004.

Applied Linear Types

Clay[44]

- Type-theoretic basis for giving type-safe code more control over memory
- Singleton types to type check loads, coercion functions to modify values' type safely

PACLANG[45]

- Program network processors for handling packets
- Unique ownership property: each packet in heap is referenced by exactly one thread
- Allow mutable aliasing within the same thread
- Operations for a functions to 1) take ownership or 2) create local aliases

[44] Hawblitzel et al., "Low-Level Linear Memory Management", 2004.

[45] Ennals, Sharp, and Mycroft, "Linear Types for Packet Processing", 2004.

COGENT

COGENT[46,47]

- Pure, polymorphic language with linear types for writing low-level systems code

[46] Amani et al., “Cogent”, 2016.

[47] O’Connor et al., “COGENT”, 2016.

COGENT[46,47]

- Pure, polymorphic language with linear types for writing low-level systems code
- Efficient machine code with in-place updates

[46] Amani et al., “Cogent”, 2016.
[47] O'Connor et al., “COGENT”, 2016.

COGENT[46,47]

- Pure, polymorphic language with linear types for writing low-level systems code
- Efficient machine code with in-place updates
- Linear types:

[46] Amani et al., "Cogent", 2016.
[47] O'Connor et al., "COGENT", 2016.

COGENT

COGENT[46,47]

- Pure, polymorphic language with linear types for writing low-level systems code
- Efficient machine code with in-place updates
- Linear types:
    - Ensure safe handling of heap-allocated objects

[46] Amani et al., "Cogent", 2016.
[47] O'Connor et al., "COGENT", 2016.

COGENT

COGENT[46,47]

- Pure, polymorphic language with linear types for writing low-level systems code
- Efficient machine code with in-place updates
- Linear types:
  - Ensure safe handling of heap-allocated objects
  - Equational functional semantics via mutable state/imperative effects

[46] Amani et al., “Cogent”, 2016.

[47] O'Connor et al., “COGENT”, 2016.

COGENT

COGENT[46,47]

- Pure, polymorphic language with linear types for writing low-level systems code
- Efficient machine code with in-place updates
- Linear types:
  - Ensure safe handling of heap-allocated objects
  - Equational functional semantics via mutable state/imperative effects
- Reason with interactive theorem prover

[46] Amani et al., "Cogent", 2016.
[47] O'Connor et al., "COGENT", 2016.

COGENT[46,47]

- Pure, polymorphic language with linear types for writing low-level systems code
- Efficient machine code with in-place updates
- Linear types:
  - Ensure safe handling of heap-allocated objects
  - Equational functional semantics via mutable state/imperative effects
- Reason with interactive theorem prover
- Missing functionality can be implemented in C, manually verified

[46] Amani et al., "Cogent", 2016.

[47] O'Connor et al., "COGENT", 2016.

COGENT

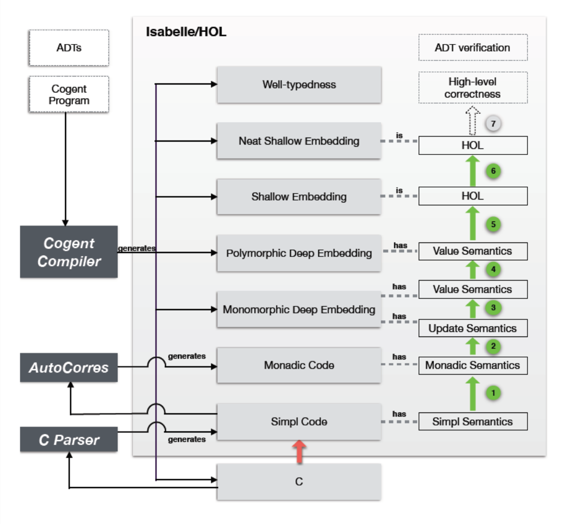
COGENT[46,47]

- Pure, polymorphic language with linear types for writing low-level systems code
- Efficient machine code with in-place updates
- Linear types:
  - Ensure safe handling of heap-allocated objects
  - Equational functional semantics via mutable state/imperative effects
- Reason with interactive theorem prover
- Missing functionality can be implemented in C, manually verified
- No trusted compiler, runtime, or garbage collector needed

[46] Amani et al., "Cogent", 2016.

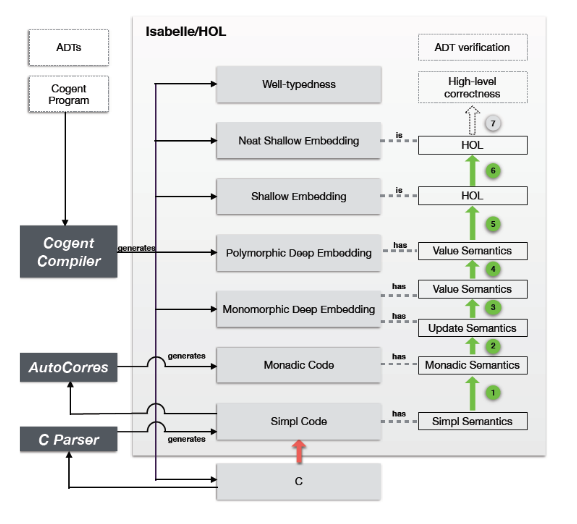
[47] O'Connor et al., "COGENT", 2016.

COGENT



ADTs
Cogent Program
Cogent Compiler
generates
AutoCorres
C Parser
Isabelle/HOL
Well-typedness
Neat Shallow Embedding
Shallow Embedding
Polymorphic Deep Embedding
Monomorphic Deep Embedding
generates
Monadic Code
Simpl Code
generates
C
ADT verification
High-level correctness
7
is
HOL
6
is
HOL
5
has
Value Semantics
4
has
Value Semantics
3
has
Update Semantics
2
has
Monadic Semantics
1
has
Simpl Semantics

COGENT



Person-Months of Work

- Proof Framework: 33.5
- Compiler: 10
- Proofs: 18

Lines (kLOC)

- Isabelle theorems: 17
- Compiler: 9.5
- ext2 Filesystem: 6.5 (Isabelle/HOL: 76.7)

Other Linear Types

Quasi-linear types[48]

- Distinguish consumed values from those that may be returned
- Use κ to control how often a variable of type τ^κ is used (many times locally)

[48] Kobayashi, "Quasi-linear Types", 1999.

[49] DeLine and Fähndrich, "Enforcing High-level Protocols in Low-level Software", 2001.

[50] Petersen et al., "A Type Theory for Memory Allocation and Data Layout", 2003.

Other Linear Types

Quasi-linear types[48]

- Distinguish consumed values from those that may be returned
- Use κ to control how often a variable of type τ^κ is used (many times locally)

Vault[49]

- Keys associate static capabilities with run-time resources
- Annotate functions with effect clause (pre- and post-conditions on held-key set)
- Windows 2000 locking errors, IRP ownership model

[48] Kobayashi, "Quasi-linear Types", 1999.

[49] DeLine and Fähndrich, "Enforcing High-level Protocols in Low-level Software", 2001.

[50] Petersen et al., "A Type Theory for Memory Allocation and Data Layout", 2003.

Other Linear Types

Quasi-linear types[48]

- Distinguish consumed values from those that may be returned
- Use κ to control how often a variable of type τ^κ is used (many times locally)

Vault[49]

- Keys associate static capabilities with run-time resources
- Annotate functions with effect clause (pre- and post-conditions on held-key set)
- Windows 2000 locking errors, IRP ownership model

Ordered types for memory layout[50]

- Variables must be used in order $\Rightarrow$ memory locations
- *Orderly lambda calculus* for size-preserving memory operations

[48] Kobayashi, "Quasi-linear Types", 1999.

[49] DeLine and Fähndrich, "Enforcing High-level Protocols in Low-level Software", 2001.

[50] Petersen et al., "A Type Theory for Memory Allocation and Data Layout", 2003.

Ownership

Types can represent ownership and prevent *aliasing* and *mutation* on the same location.

[51] Evans, "Static Detection of Dynamic Memory Errors", 1996.

[52] Clarke, Potter, and Noble, "Ownership Types for Flexible Alias Protection", 1998.

[53] Fähndrich et al., "Language Support for Fast and Reliable Message-based Communication in Singularity OS", 2006.

Ownership

Types can represent ownership and prevent *aliasing* and *mutation* on the same location.

LCL[51]

- `owned` annotation to denote reference with obligation to release storage
- `dependent` annotation for sharing; user ensures lifetimes contained properly

[51] Evans, "Static Detection of Dynamic Memory Errors", 1996.

[52] Clarke, Potter, and Noble, "Ownership Types for Flexible Alias Protection", 1998.

[53] Fähndrich et al., "Language Support for Fast and Reliable Message-based Communication in Singularity OS", 2006.

Ownership

Types can represent ownership and prevent *aliasing* and *mutation* on the same location.

LCL[51]

- `owned` annotation to denote reference with obligation to release storage
- `dependent` annotation for sharing; user ensures lifetimes contained properly

Ownership Types[52]

- Object's definition includes unique object context that owns it

[51] Evans, "Static Detection of Dynamic Memory Errors", 1996.

[52] Clarke, Potter, and Noble, "Ownership Types for Flexible Alias Protection", 1998.

[53] Fähndrich et al., "Language Support for Fast and Reliable Message-based Communication in Singularity OS", 2006.

Ownership

Types can represent ownership and prevent *aliasing* and *mutation* on the same location.

LCL[51]

- `owned` annotation to denote reference with obligation to release storage
- `dependent` annotation for sharing; user ensures lifetimes contained properly

Ownership Types[52]

- Object's definition includes unique object context that owns it

Singularity[53]

- Type system tracks resources, passes ownership of arguments to callee

---

[51] Evans, "Static Detection of Dynamic Memory Errors", 1996.

[52] Clarke, Potter, and Noble, "Ownership Types for Flexible Alias Protection", 1998.

[53] Fähndrich et al., "Language Support for Fast and Reliable Message-based Communication in Singularity OS", 2006.

Rust[54,55,56]

- Ownership and lifetimes

[54] Matsakis and Klock, "The Rust Language", 2014.
[55] Levy et al., "Ownership is Theft", 2015.
[56] Jung et al., "RustBelt", 2017.

Rust[54,55,56]

- Ownership and lifetimes
- Type system enforces that objects have unique owners

[54] Matsakis and Klock, "The Rust Language", 2014.
[55] Levy et al., "Ownership is Theft", 2015.
[56] Jung et al., "RustBelt", 2017.

Rust[54,55,56]

- Ownership and lifetimes
- Type system enforces that objects have unique owners
- Objects may be borrowed for no longer than owner

[54] Matsakis and Klock, "The Rust Language", 2014.
[55] Levy et al., "Ownership is Theft", 2015.
[56] Jung et al., "RustBelt", 2017.

Rust[54,55,56]

- Ownership and lifetimes
- Type system enforces that objects have unique owners
- Objects may be borrowed for no longer than owner
- Object automatically deallocated when owner leaves scope

[54] Matsakis and Klock, "The Rust Language", 2014.
[55] Levy et al., "Ownership is Theft", 2015.
[56] Jung et al., "RustBelt", 2017.

Rust[54,55,56]

- Ownership and lifetimes
- Type system enforces that objects have unique owners
- Objects may be borrowed for no longer than owner
- Object automatically deallocated when owner leaves scope
- Unsafe sections for mutating raw pointers and aliased state

[54] Matsakis and Klock, "The Rust Language", 2014.
[55] Levy et al., "Ownership is Theft", 2015.
[56] Jung et al., "RustBelt", 2017.

Rust[54,55,56]

- Ownership and lifetimes
- Type system enforces that objects have unique owners
- Objects may be borrowed for no longer than owner
- Object automatically deallocated when owner leaves scope
- Unsafe sections for mutating raw pointers and aliased state
- Regions $\approx$ lifetimes (region capabilities $\approx$ lifetime tokens)

[54] Matsakis and Klock, “The Rust Language”, 2014.
[55] Levy et al., “Ownership is Theft”, 2015.
[56] Jung et al., “RustBelt”, 2017.

Rust[54,55,56]

- Ownership and lifetimes
- Type system enforces that objects have unique owners
- Objects may be borrowed for no longer than owner
- Object automatically deallocated when owner leaves scope
- Unsafe sections for mutating raw pointers and aliased state
- Regions $\approx$ lifetimes (region capabilities $\approx$ lifetime tokens)

---

[54] Matsakis and Klock, "The Rust Language", 2014.
[55] Levy et al., "Ownership is Theft", 2015.
[56] Jung et al., "RustBelt", 2017.

Rust[54,55,56]

- Ownership and lifetimes
- Type system enforces that objects have unique owners
- Objects may be borrowed for no longer than owner
- Object automatically deallocated when owner leaves scope
- Unsafe sections for mutating raw pointers and aliased state
- Regions ≈ lifetimes (region capabilities ≈ lifetime tokens)

```
let (snd, rcv) = channel();
join(move || {
      let mut v = Vec::new();
      v.push(0);
      snd.send(v);
      v.push(1);
    },
    move || {
      let v = rcv.recv().unwrap();
      println!("Received: {:?}", v);
    });
```

[54] Matsakis and Klock, "The Rust Language", 2014.

[55] Levy et al., "Ownership is Theft", 2015.

[56] Jung et al., "RustBelt", 2017.

Rust[54,55,56]

- Ownership and lifetimes
- Type system enforces that objects have unique owners
- Objects may be borrowed for no longer than owner
- Object automatically deallocated when owner leaves scope
- Unsafe sections for mutating raw pointers and aliased state
- Regions ≈ lifetimes (region capabilities ≈ lifetime tokens)

```
let (snd, rcv) = channel();
join(move || {
        let mut v = Vec::new();
        v.push(0);
        snd.send(v);
        v.push(1);
    },
    move || {
        let v = rcv.recv().unwrap();
        println!("Received:␣{:?}", v);
    });
```

[54] Matsakis and Klock, “The Rust Language”, 2014.
[55] Levy et al., “Ownership is Theft”, 2015.
[56] Jung et al., “RustBelt”, 2017.

Rust[54,55,56]

- Ownership and lifetimes
- Type system enforces that objects have unique owners
- Objects may be borrowed for no longer than owner
- Object automatically deallocated when owner leaves scope
- Unsafe sections for mutating raw pointers and aliased state
- Regions ≈ lifetimes (region capabilities ≈ lifetime tokens)

```
let (snd, rcv) = channel();
join(move || {
      let mut v = Vec::new();
      v.push(0);
      snd.send(v);
      v.push(1);
    },
    move || {
      let v = rcv.recv().unwrap();
      println!("Received: {:?}", v);
    });
```

[54] Matsakis and Klock, “The Rust Language”, 2014.

[55] Levy et al., “Ownership is Theft”, 2015.

[56] Jung et al., “RustBelt”, 2017.

Rust[54,55,56]

- Ownership and lifetimes
- Type system enforces that objects have unique owners
- Objects may be borrowed for no longer than owner
- Object automatically deallocated when owner leaves scope
- Unsafe sections for mutating raw pointers and aliased state
- Regions ≈ lifetimes (region capabilities ≈ lifetime tokens)

```
let (snd, rcv) = channel();
join(move || {
      let mut v = Vec::new();
      v.push(0);
      snd.send(v);
      v.push(1);
    },
    move || {
      let v = rcv.recv().unwrap();
      println!("Received:␣{:?}", v);
    });
```

[54] Matsakis and Klock, “The Rust Language”, 2014.

[55] Levy et al., “Ownership is Theft”, 2015.

[56] Jung et al., “RustBelt”, 2017.

Rust[54,55,56]

- Ownership and lifetimes
- Type system enforces that objects have unique owners
- Objects may be borrowed for no longer than owner
- Object automatically deallocated when owner leaves scope
- Unsafe sections for mutating raw pointers and aliased state
- Regions ≈ lifetimes (region capabilities ≈ lifetime tokens)

```
let (snd, rcv) = channel();
join(move || {
      let mut v = Vec::new();
      v.push(0);
      snd.send(v);
      v.push(1);
    },
    move || {
      let v = rcv.recv().unwrap();
      println!("Received: {:?}", v);
    });
```

[54] Matsakis and Klock, "The Rust Language", 2014.

[55] Levy et al., "Ownership is Theft", 2015.

[56] Jung et al., "RustBelt", 2017.

Rust[54,55,56]

- Ownership and lifetimes
- Type system enforces that objects have unique owners
- Objects may be borrowed for no longer than owner
- Object automatically deallocated when owner leaves scope
- Unsafe sections for mutating raw pointers and aliased state
- Regions ≈ lifetimes (region capabilities ≈ lifetime tokens)

```
let (snd, rcv) = channel();
join(move || {
        let mut v = Vec::new();
        v.push(0);
        snd.send(v);
        v.push(1);
    },
    move || {
        let v = rcv.recv().unwrap();
        println!("Received:␣{:?}", v);
    });
```

[54]Matsakis and Klock, "The Rust Language", 2014.
[55]Levy et al., "Ownership is Theft", 2015.
[56]Jung et al., "RustBelt", 2017.

Rust[54,55,56]

- Ownership and lifetimes
- Type system enforces that objects have unique owners
- Objects may be borrowed for no longer than owner
- Object automatically deallocated when owner leaves scope
- Unsafe sections for mutating raw pointers and aliased state
- Regions ≈ lifetimes (region capabilities ≈ lifetime tokens)

```
let (snd, rcv) = channel();
join(move || {
        let mut v = Vec::new();
        v.push(0);
        snd.send(v);
        v.push(1);
    },
    move || {
        let v = rcv.recv().unwrap();
        println!("Received:␣{:?}", v);
    });
```

[54] Matsakis and Klock, "The Rust Language", 2014.

[55] Levy et al., "Ownership is Theft", 2015.

[56] Jung et al., "RustBelt", 2017.

Rust[54,55,56]

- Ownership and lifetimes
- Type system enforces that objects have unique owners
- Objects may be borrowed for no longer than owner
- Object automatically deallocated when owner leaves scope
- Unsafe sections for mutating raw pointers and aliased state
- Regions ≈ lifetimes (region capabilities ≈ lifetime tokens)

```
let (snd, rcv) = channel();
join(move || {
      let mut v = Vec::new();
      v.push(0);
      snd.send(v);
      v.push(1);
    },
    move || {
      let v = rcv.recv().unwrap();
      println!("Received:␣{:?}", v);
    });
```

[54] Matsakis and Klock, "The Rust Language", 2014.

[55] Levy et al., "Ownership is Theft", 2015.

[56] Jung et al., "RustBelt", 2017.

Quick Temporal Recap

- Capabilities and pointer-based metadata

Quick Temporal Recap

- Capabilities and pointer-based metadata
- Effects and regions

Quick Temporal Recap

- Capabilities and pointer-based metadata
- Effects and regions
- Linear types

Quick Temporal Recap

- Capabilities and pointer-based metadata
- Effects and regions
- Linear types
- Ownership and borrowing

Conclusion

- Memory errors $\equiv$ *type errors*

Conclusion

- Memory errors $\equiv$ *type errors*
- *Static* memory management

Conclusion

- Memory errors $\equiv$ *type errors*
- *Static* memory management
- Isolate unsafe world

Conclusion

- Memory errors $\equiv$ *type errors*
- *Static* memory management
- Isolate unsafe world
- Be reasonable and optimistic

Thanks

Thanks!