Context-sensitive Correlation Analysis for Detecting Races

Polyvios Pratikakis
Jeff Foster
Michael Hicks

University of Maryland, College Park

Data Races are Bad

- Race: two threads access memory without synchronization and at least one is a write
- Races are bad:
  - August 14th 2004, Northeastern Blackout
  - 1985-1987, Therac-25 medical accelerator
- Programs with races are difficult to understand

A way to prevent races

- Shared locations ρ
- Locks ℓ
- Correlation $\rho \triangleright \ell$:
  Lock ℓ is correlated with pointer ρ if-f ℓ is held while ρ is accessed
- *Consistent correlation*:
  A given pointer ρ is *only* correlated with one lock ℓ
- Assert that every shared location ρ is *consistently correlated* with a *single* lock ℓ

Contribution

- Inference of *correlation* between locks and pointers for C:
- Universal and Existential context sensitivity in correlation propagation
- *Sound* race detection using assertion of *consistent correlation*
- It works: we found races!

This presentation

- Correlation Inference
- Universal and Existential context sensitivity
- Linearity of locks
- Lock State (which locks are held at every program point?)
- Experimental Results

Type Based Analysis

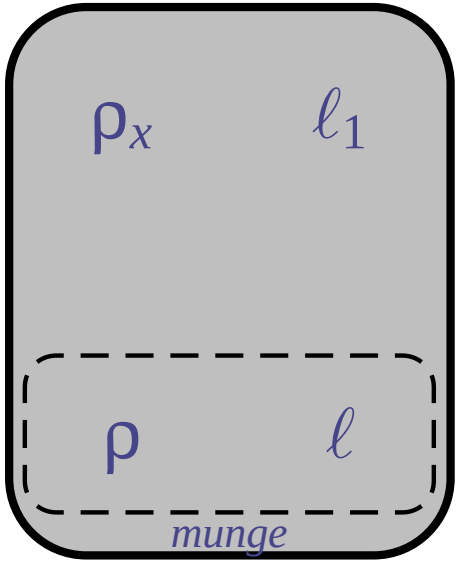
- Annotate types with labels:
  - $\texttt{pthread\_mutex\_t} \rightarrow \texttt{pthread\_mutex\_t}\langle \ell \rangle$
  - $\tau^* \rightarrow \tau^* \langle \rho \rangle$
- Create constraints among labels to capture data flow and correlation
  - Dereferencing ρ while ℓ is held: $\rho \triangleright \ell$
  - Aliasing ρ to ρ': $\rho \leq \rho'$
  - Aliasing ℓ to ℓ': $\ell = \ell'$
- Solve constraints to close the relation $\rho \triangleright \ell$
- Verify *consistent correlation* of every shared ρ with a single lock ℓ for all dereferences of ρ

Correlation

```
pthread_mutex_t    L1 = ...;
int x; // &x:  int*
void munge(pthread_mutex_t   *l, int *    p) {
   pthread_mutex_lock(l);
   *p = 3;
   pthread_mutex_unlock(l);
}
...
munge(&L1, &x);
```

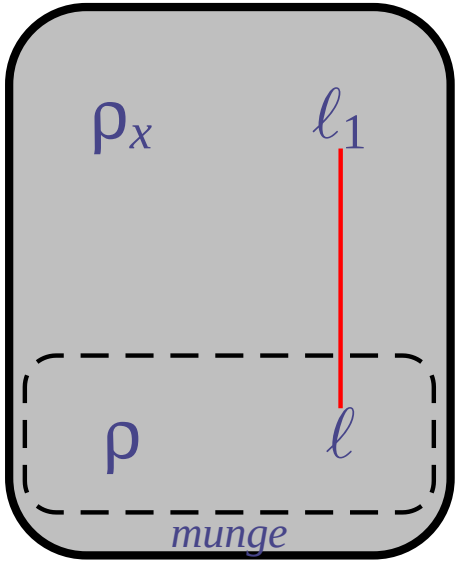
Correlation

```
pthread_mutex_t⟨ℓ1⟩ L1 = ...;
int x; // &x:  int*⟨ρx⟩
void munge(pthread_mutex_t⟨ℓ⟩ *l, int *⟨ρ⟩ p) {
   pthread_mutex_lock(l);
   *p = 3;
   pthread_mutex_unlock(l);
}
...
munge(&L1, &x);
```



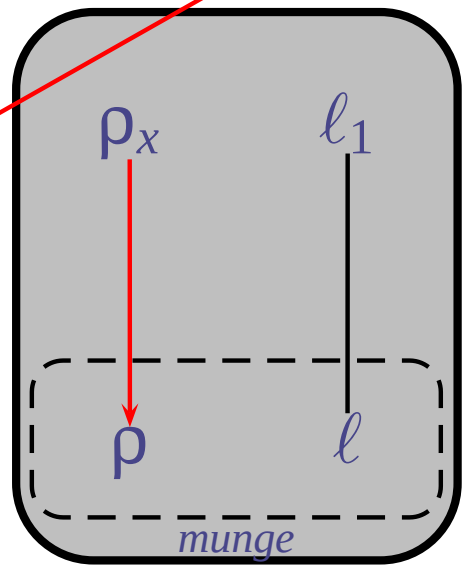
Correlation

```
pthread_mutex_t⟨ℓ1⟩ L1 = ...;
int x; // &x:  int*⟨ρx⟩
void munge(pthread_mutex_t⟨ℓ⟩ *l, int *⟨ρ⟩ p) {
   pthread_mutex_lock(l);
   *p = 3;
   pthread_mutex_unlock(l);
}
...
munge(&L1, &x);
```



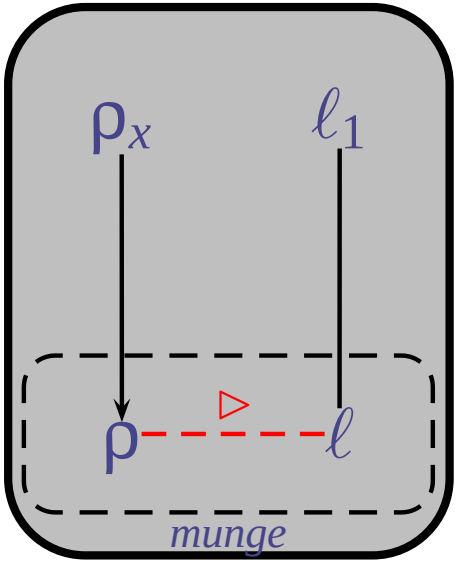
Correlation

```
pthread_mutex_t⟨ℓ1⟩ L1 = ...;
int x; // &x:  int*⟨ρx⟩
void munge(pthread_mutex_t⟨ℓ⟩ *l, int *⟨ρ⟩ p) {
   pthread_mutex_lock(l);
   *p = 3;
   pthread_mutex_unlock(l);
}
...
munge(&L1, &x);
```



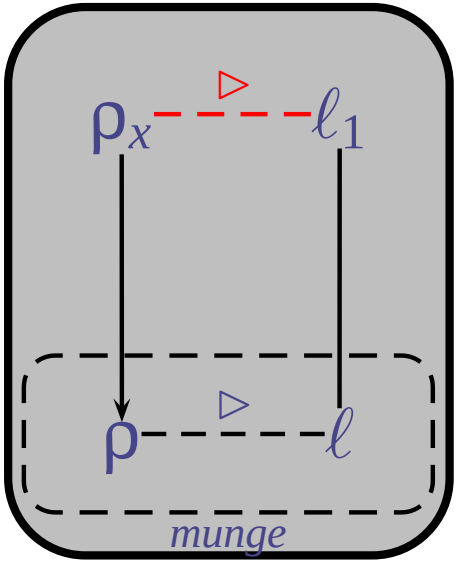
Correlation

```
pthread_mutex_t⟨ℓ1⟩ L1 = ...;
int x; // &x:  int*⟨ρx⟩
void munge(pthread_mutex_t⟨ℓ⟩ *l, int *⟨ρ⟩ p) {
   pthread_mutex_lock(l);
   *p = 3;
   pthread_mutex_unlock(l);
}
...
munge(&L1, &x);
```



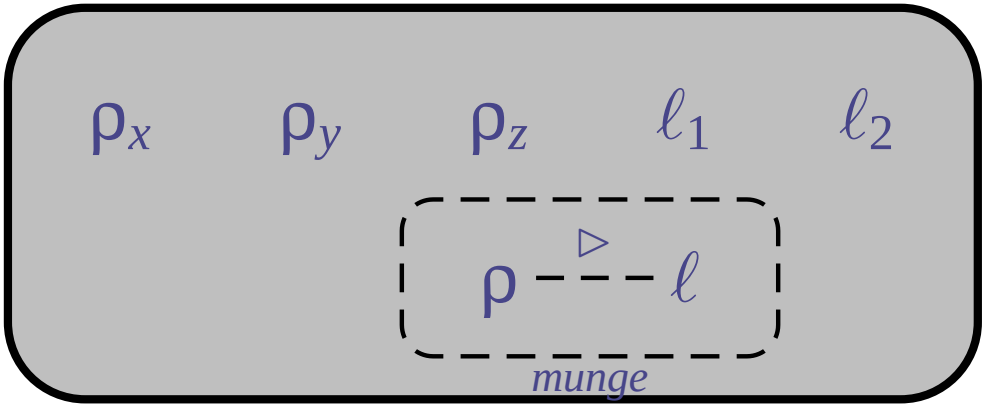
Correlation

```
pthread_mutex_t⟨ℓ1⟩ L1 = ...;
int x; // &x:  int*⟨ρx⟩
void munge(pthread_mutex_t⟨ℓ⟩ *l, int *⟨ρ⟩ p) {
   pthread_mutex_lock(l);
   *p = 3;
   pthread_mutex_unlock(l);
}
...
munge(&L1, &x);
```



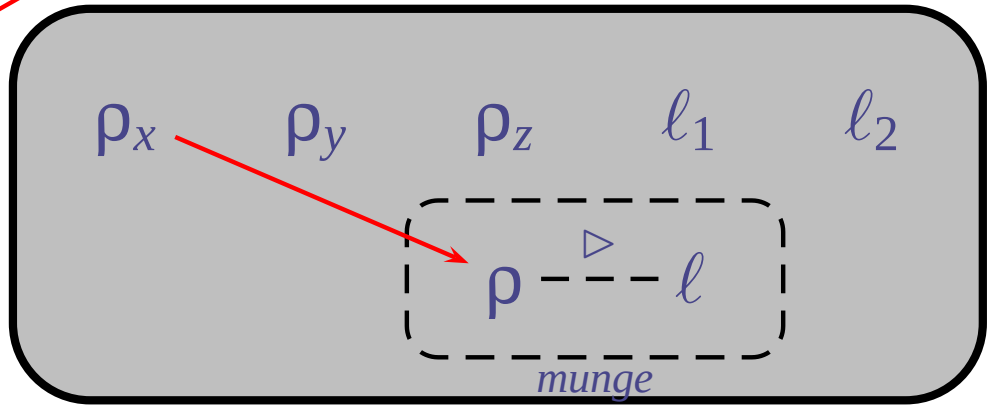
Context Sensitivity

```
pthread_mutex_t⟨ℓ1⟩ L1 = ..., ⟨ℓ2⟩ L2 = ...;
int x, y, z; // ⟨ρx⟩, ⟨ρy⟩, ⟨ρz⟩
void munge(pthread_mutex_t⟨ℓ⟩ *l, int *⟨ρ⟩ p) {
   pthread_mutex_lock(l);
   *p = 3;
   pthread_mutex_unlock(l);
}
...
munge(&L1, &x);
munge(&L2, &y);
munge(&L2, &z);
```



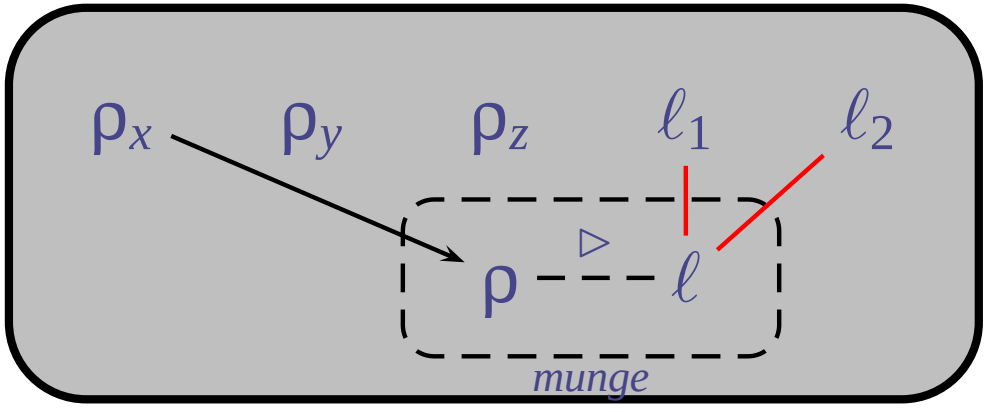
Context Sensitivity

```
pthread_mutex_t⟨ℓ1⟩ L1 = ..., ⟨ℓ2⟩ L2 = ...;
int x, y, z; // ⟨ρx⟩, ⟨ρy⟩, ⟨ρz⟩
void munge(pthread_mutex_t⟨ℓ⟩ *l, int *⟨ρ⟩ p) {
   pthread_mutex_lock(l);
   *p = 3;
   pthread_mutex_unlock(l);
}
...
munge(&L1, &x);
munge(&L2, &y);
munge(&L2, &z);
```



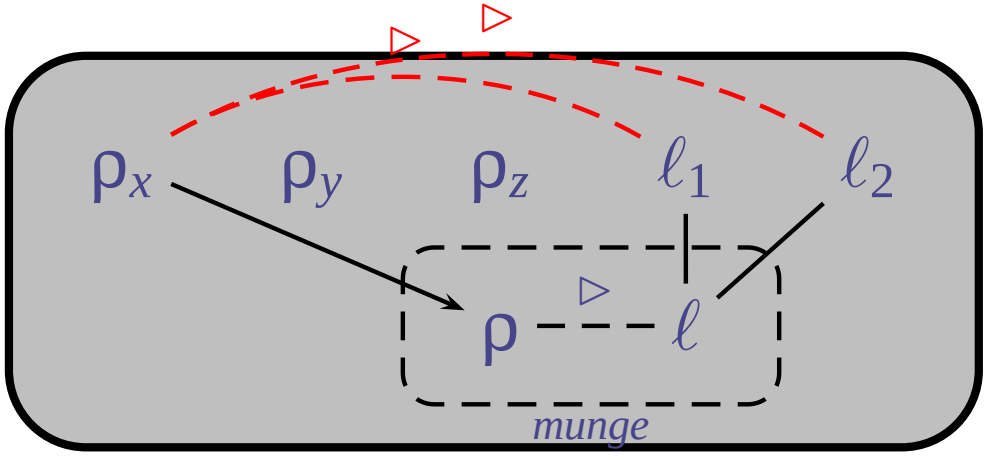
Context Sensitivity

```
pthread_mutex_t⟨ℓ1⟩ L1 = ..., ⟨ℓ2⟩ L2 = ...;
int x, y, z; // ⟨ρx⟩, ⟨ρy⟩, ⟨ρz⟩
void munge(pthread_mutex_t⟨ℓ⟩ *l, int *⟨ρ⟩ p) {
   pthread_mutex_lock(l);
   *p = 3;
   pthread_mutex_unlock(l);
}
...
munge(&L1, &x);
munge(&L2, &y);
munge(&L2, &z);
```



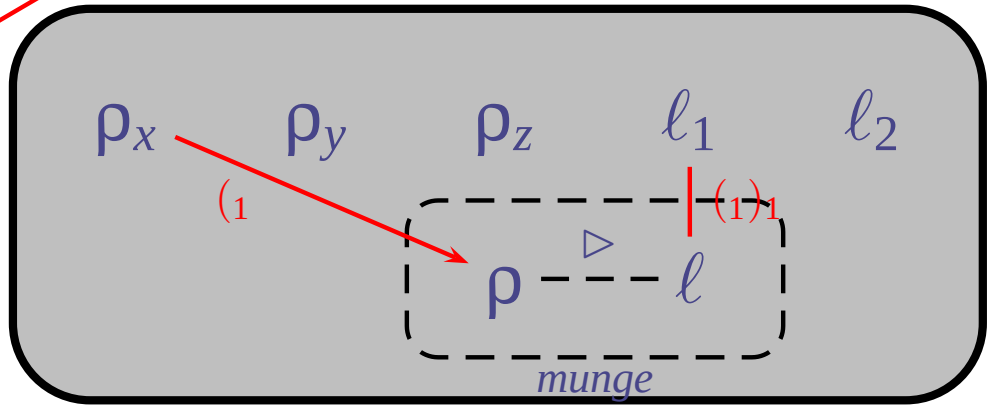
Context Sensitivity

```
pthread_mutex_t⟨ℓ1⟩ L1 = ..., ⟨ℓ2⟩ L2 = ...;
int x, y, z; // ⟨ρx⟩, ⟨ρy⟩, ⟨ρz⟩
void munge(pthread_mutex_t⟨ℓ⟩ *l, int *⟨ρ⟩ p) {
   pthread_mutex_lock(l);
   *p = 3;
   pthread_mutex_unlock(l);
}
...
munge(&L1, &x);
munge(&L2, &y);
munge(&L2, &z);
```



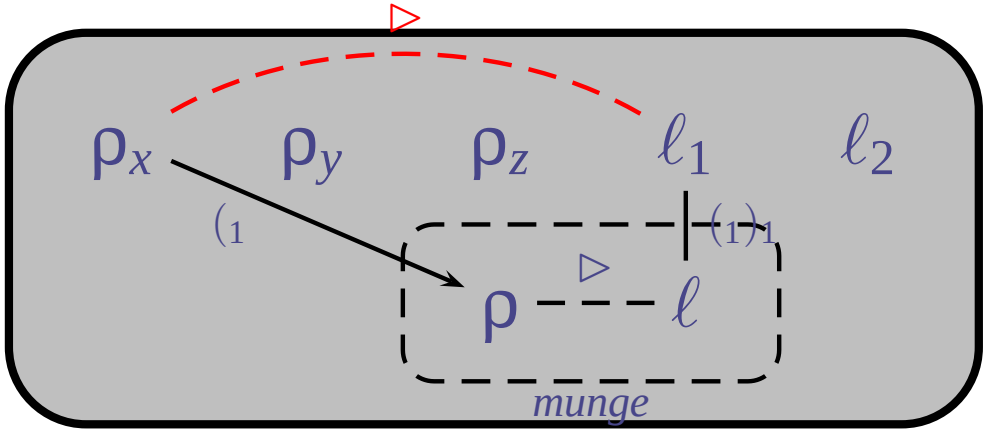
Context Sensitivity

```
pthread_mutex_t⟨ℓ1⟩ L1 = ..., ⟨ℓ2⟩ L2 = ...;
int x, y, z; // ⟨ρx⟩, ⟨ρy⟩, ⟨ρz⟩
void munge(pthread_mutex_t⟨ℓ⟩ *l, int *⟨ρ⟩ p) {
   pthread_mutex_lock(l);
   *p = 3;
   pthread_mutex_unlock(l);
}
...
munge1(&L1, &x);
munge2(&L2, &y);
munge3(&L2, &z);
```



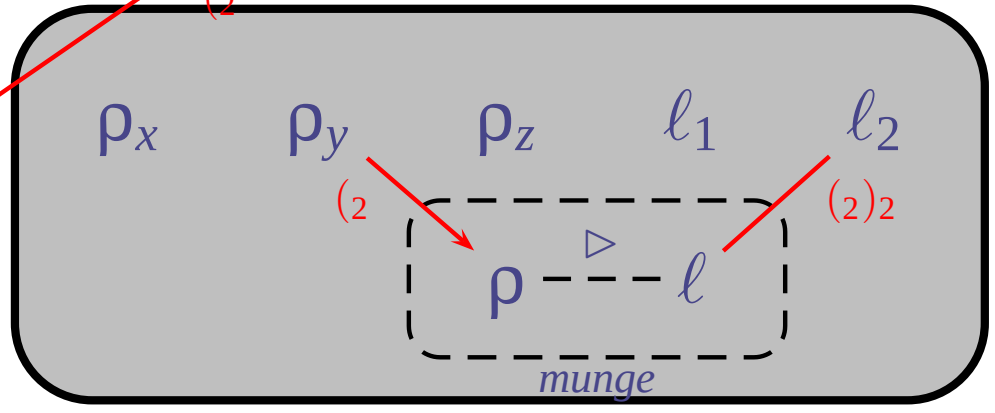
Context Sensitivity

```
pthread_mutex_t⟨ℓ1⟩ L1 = ..., ⟨ℓ2⟩ L2 = ...;
int x, y, z; // ⟨ρx⟩, ⟨ρy⟩, ⟨ρz⟩
void munge(pthread_mutex_t⟨ℓ⟩ *l, int *⟨ρ⟩ p) {
   pthread_mutex_lock(l);
   *p = 3;
   pthread_mutex_unlock(l);
}
...
munge1(&L1, &x);
munge2(&L2, &y);
munge3(&L2, &z);
```



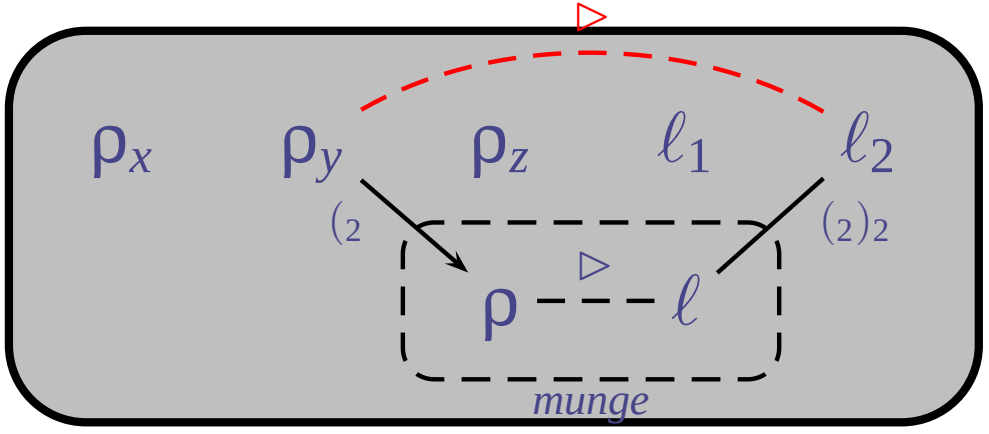
Context Sensitivity

```
pthread_mutex_t⟨ℓ1⟩ L1 = ..., ⟨ℓ2⟩ L2 = ...;
int x, y, z; // ⟨ρx⟩, ⟨ρy⟩, ⟨ρz⟩
void munge(pthread_mutex_t⟨ℓ⟩ *l, int *⟨ρ⟩ p) {
   pthread_mutex_lock(l);
   *p = 3;
   pthread_mutex_unlock(l);
}
...
munge1(&L1, &x);
munge2(&L2, &y);
munge3(&L2, &z);
```



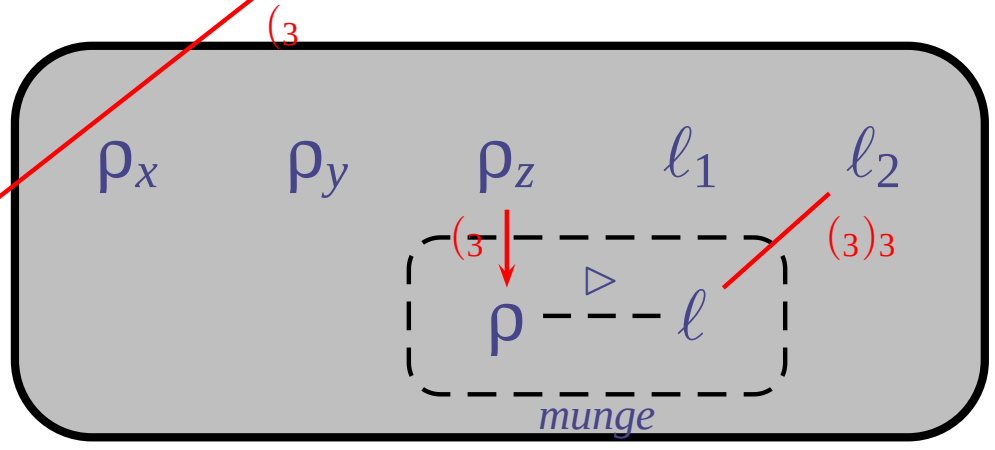
Context Sensitivity

```
pthread_mutex_t⟨ℓ1⟩ L1 = ..., ⟨ℓ2⟩ L2 = ...;
int x, y, z; // ⟨ρx⟩, ⟨ρy⟩, ⟨ρz⟩
void munge(pthread_mutex_t⟨ℓ⟩ *l, int *⟨ρ⟩ p) {
   pthread_mutex_lock(l);
   *p = 3;
   pthread_mutex_unlock(l);
}
...
munge1(&L1, &x);
munge2(&L2, &y);
munge3(&L2, &z);
```



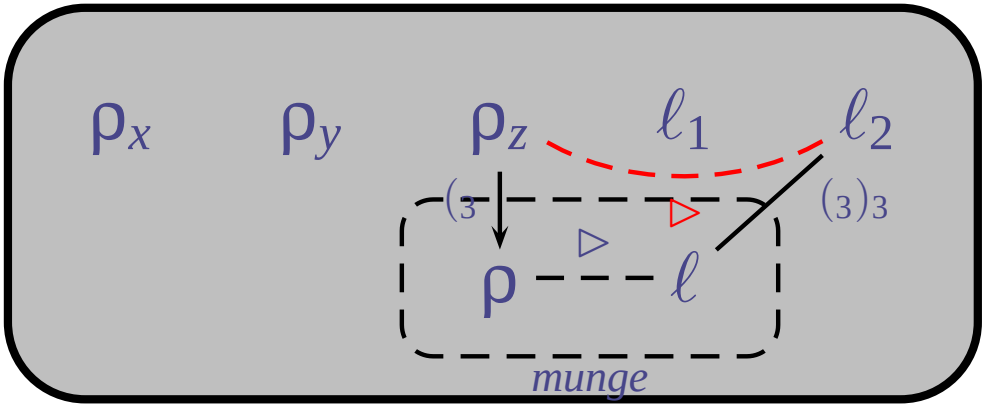
Context Sensitivity

```
pthread_mutex_t⟨ℓ1⟩ L1 = ..., ⟨ℓ2⟩ L2 = ...;
int x, y, z; // ⟨ρx⟩, ⟨ρy⟩, ⟨ρz⟩
void munge(pthread_mutex_t⟨ℓ⟩ *l, int *⟨ρ⟩ p) {
   pthread_mutex_lock(l);
   *p = 3;
   pthread_mutex_unlock(l);
}
...
munge1(&L1, &x);
munge2(&L2, &y);
munge3(&L2, &z);
```



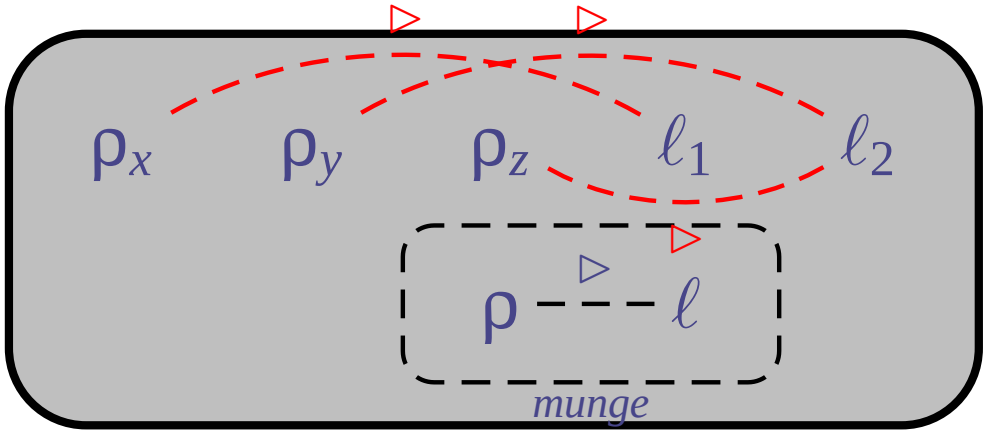
Context Sensitivity

```
pthread_mutex_t⟨ℓ1⟩ L1 = ..., ⟨ℓ2⟩ L2 = ...;
int x, y, z; // ⟨ρx⟩, ⟨ρy⟩, ⟨ρz⟩
void munge(pthread_mutex_t⟨ℓ⟩ *l, int *⟨ρ⟩ p) {
   pthread_mutex_lock(l);
   *p = 3;
   pthread_mutex_unlock(l);
}
...
munge1(&L1, &x);
munge2(&L2, &y);
munge3(&L2, &z);
```



Context Sensitivity

```
pthread_mutex_t⟨ℓ1⟩ L1 = ..., ⟨ℓ2⟩ L2 = ...;
int x, y, z; // ⟨ρx⟩, ⟨ρy⟩, ⟨ρz⟩
void munge(pthread_mutex_t⟨ℓ⟩ *l, int *⟨ρ⟩ p) {
   pthread_mutex_lock(l);
   *p = 3;
   pthread_mutex_unlock(l);
}
...
munge1(&L1, &x);
munge2(&L2, &y);
munge3(&L2, &z);
```



Existential Context Sensitivity

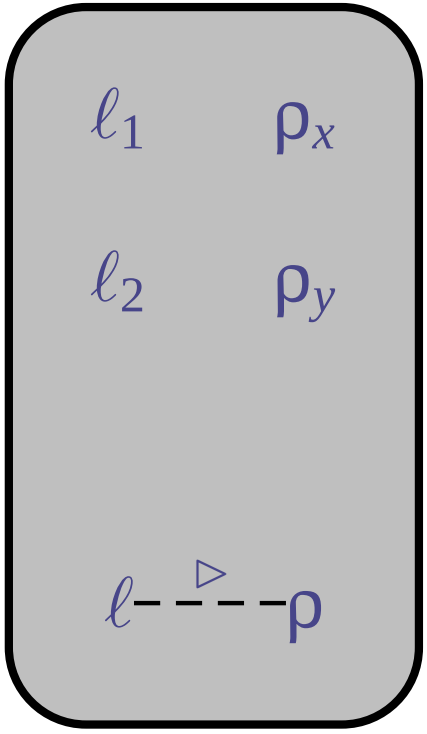
- Often, locks exist in data structures:

```
struct foo {
   pthread_mutex_t⟨ℓ⟩ lock;
   int*⟨ρ⟩ data;
};
```

- Impossible to track locks precisely

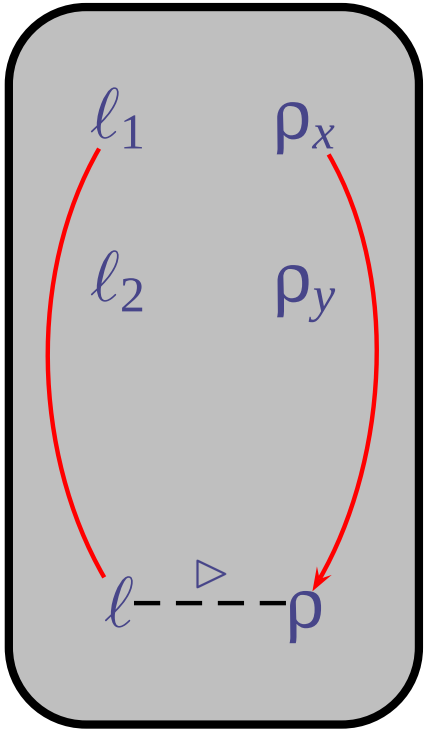
Existential Context Sensitivity

```
struct foo⟨ℓ,ρ⟩ s;
if(...)          {
   s.lock = &L1; s.data = &x;
} else    {
   s.lock = &L2; s.data = &y;
}
pthread_mutex_lock(s.lock);
*(s.data) = 3;
pthread_mutex_unlock(s.lock);
```



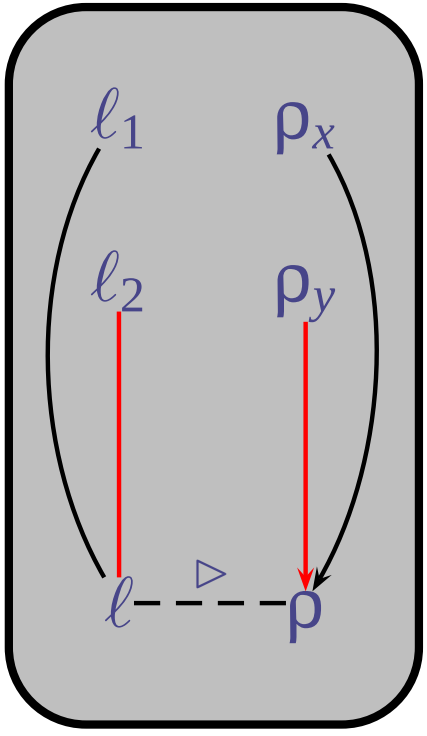
Existential Context Sensitivity

```
struct foo⟨ℓ,ρ⟩ s;
if(...)          {
   s.lock = &L1; s.data = &x;
} else    {
   s.lock = &L2; s.data = &y;
}
pthread_mutex_lock(s.lock);
*(s.data) = 3;
pthread_mutex_unlock(s.lock);
```



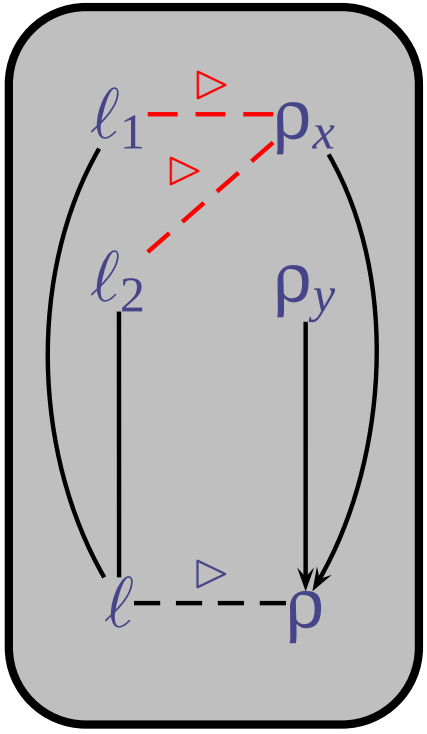
Existential Context Sensitivity

```
struct foo⟨ℓ,ρ⟩ s;
if(...)          {
   s.lock = &L1; s.data = &x;
} else    {
   s.lock = &L2; s.data = &y;
}
pthread_mutex_lock(s.lock);
*(s.data) = 3;
pthread_mutex_unlock(s.lock);
```



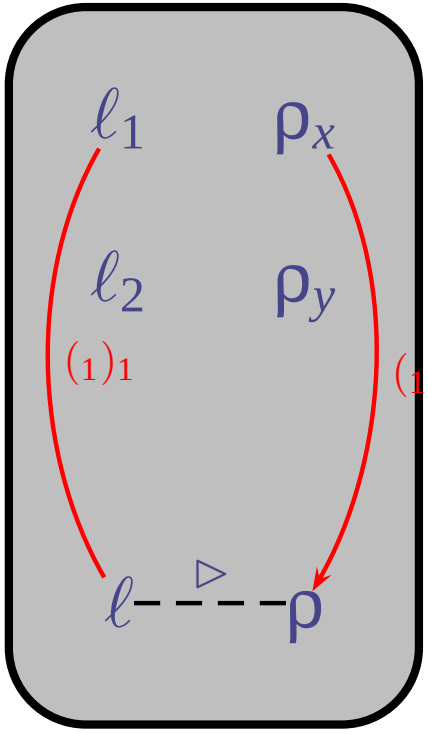
Existential Context Sensitivity

```
struct foo⟨ℓ,ρ⟩ s;
if(...)          {
   s.lock = &L1; s.data = &x;
} else    {
   s.lock = &L2; s.data = &y;
}
pthread_mutex_lock(s.lock);
*(s.data) = 3;
pthread_mutex_unlock(s.lock);
```



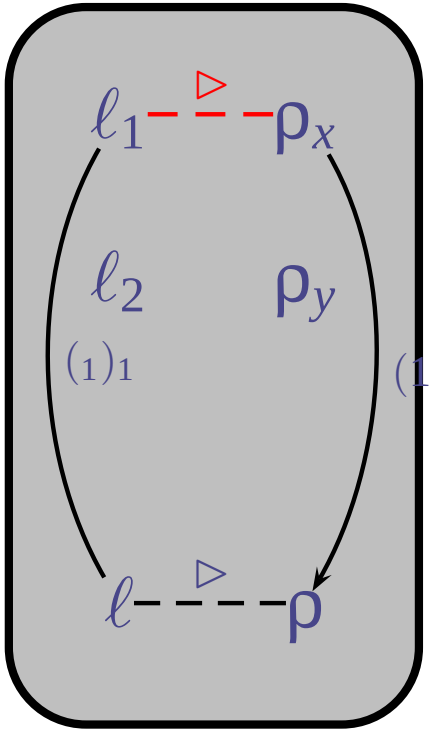
Existential Context Sensitivity

```
struct foo⟨ℓ,ρ⟩ s;
if(...)     pack¹(s)     {
   s.lock = &L1; s.data = &x;
} else     {
   s.lock = &L2; s.data = &y;
}
pthread_mutex_lock(s.lock);
*(s.data) = 3;
pthread_mutex_unlock(s.lock);
```



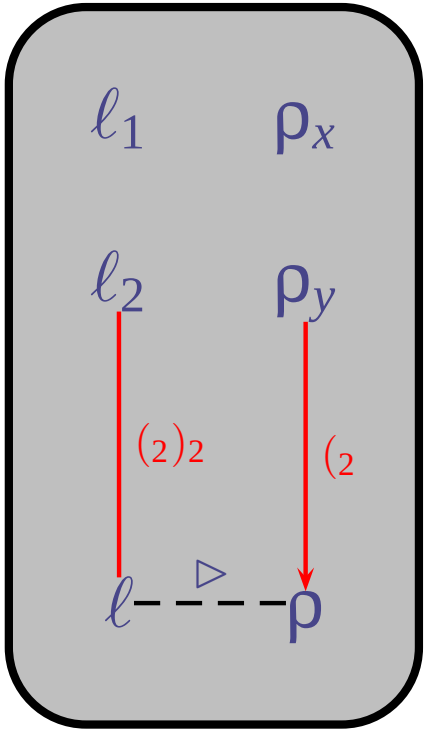
Existential Context Sensitivity

```
struct foo⟨ℓ,ρ⟩ s;
if(...)    pack¹(s)  {
   s.lock = &L1; s.data = &x;
} else    {
   s.lock = &L2; s.data = &y;
}
pthread_mutex_lock(s.lock);
*(s.data) = 3;
pthread_mutex_unlock(s.lock);
```



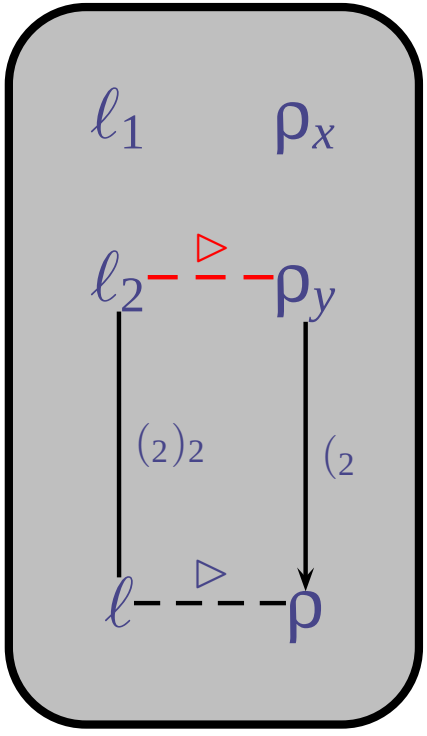
Existential Context Sensitivity

```
struct foo⟨ℓ,ρ⟩ s;
if(...)    pack¹(s)  {
   s.lock = &L1; s.data = &x;
} else  pack²(s)    {
   s.lock = &L2; s.data = &y;
}
pthread_mutex_lock(s.lock);
*(s.data) = 3;
pthread_mutex_unlock(s.lock);
```



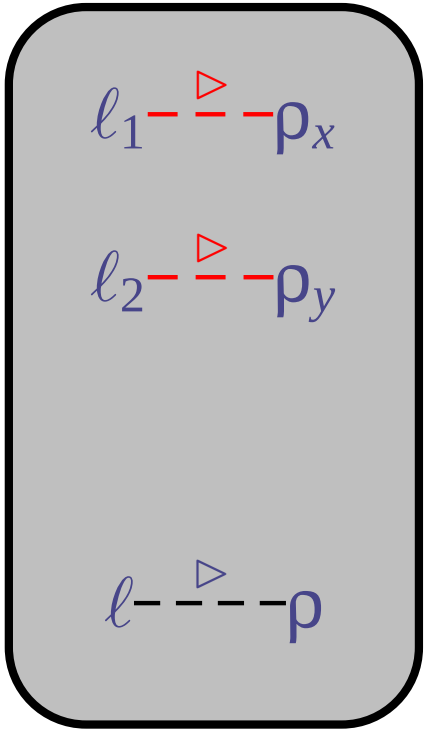
Existential Context Sensitivity

```
struct foo⟨ℓ,ρ⟩ s;
if(...)    pack¹(s)  {
   s.lock = &L1; s.data = &x;
} else  pack²(s)  {
   s.lock = &L2; s.data = &y;
}
pthread_mutex_lock(s.lock);
*(s.data) = 3;
pthread_mutex_unlock(s.lock);
```



Existential Context Sensitivity

```
struct foo⟨ℓ,ρ⟩ s;
if(...)    pack¹(s)  {
   s.lock = &L1; s.data = &x;
} else  pack²(s)  {
   s.lock = &L2; s.data = &y;
}
pthread_mutex_lock(s.lock);
*(s.data) = 3;
pthread_mutex_unlock(s.lock);
```



Linearity of locks

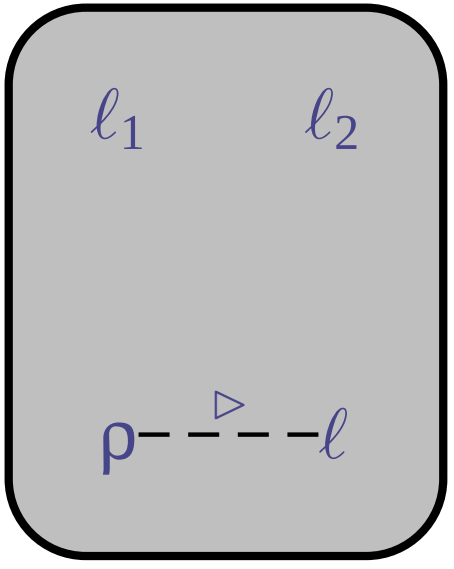
- Each lock label ℓ represents one or more run-time locks
- Locks ℓ have to be linear to check *consistent correlation*
- If ℓ can represent more than one run-time lock and is acquired, which run-time lock is acquired?
- Locks ℓ have to be linear to track their state precisely
- Challenges:
  - dynamic allocation of locks
  - want to avoid being overly conservative in loops

Linearity Effects

- Prevent simply unifying every ℓ - assert linearity
- Each expression has a *linearity effect* ε
- Allocating a fresh lock has a *fresh* singleton effect $\{\ell\}$
- Effect of composite expressions is *disjoint union* ($\uplus$) of effects
- Filter effects to remove any ℓ that does not escape

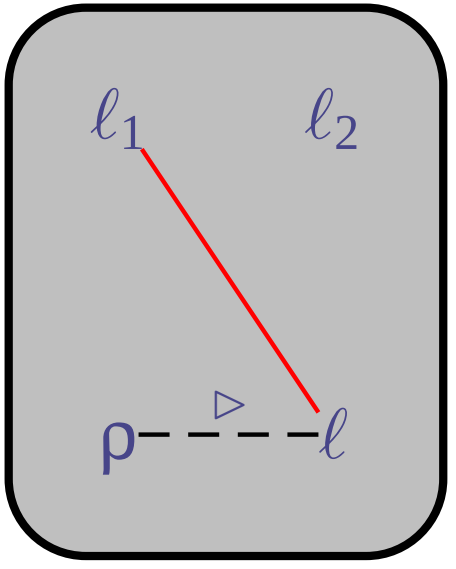
Linearity Example

```
pthread_mutex_t L1⟨ℓ1⟩, L2⟨ℓ2⟩, *l⟨ℓ⟩;
int x; // &x:  int*⟨ρx⟩

pthread_mutex_init(&L1);
pthread_mutex_init(&L2);
if(...)  l = &L1;
else l = &L2;
pthread_mutex_lock(&l);
x = 3;
pthread_mutex_unlock(&l);
```



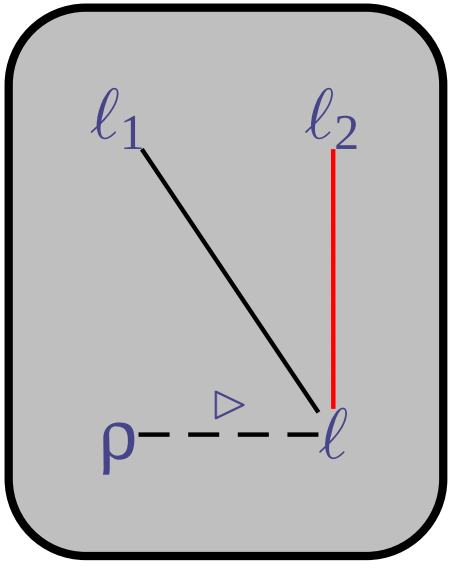
Linearity Example

```
pthread_mutex_t L1⟨ℓ1⟩, L2⟨ℓ2⟩, *l⟨ℓ⟩;
int x; // &x:  int*⟨ρx⟩

pthread_mutex_init(&L1);
pthread_mutex_init(&L2);
if(...)    l = &L1;
else l = &L2;
pthread_mutex_lock(&l);
x = 3;
pthread_mutex_unlock(&l);
```



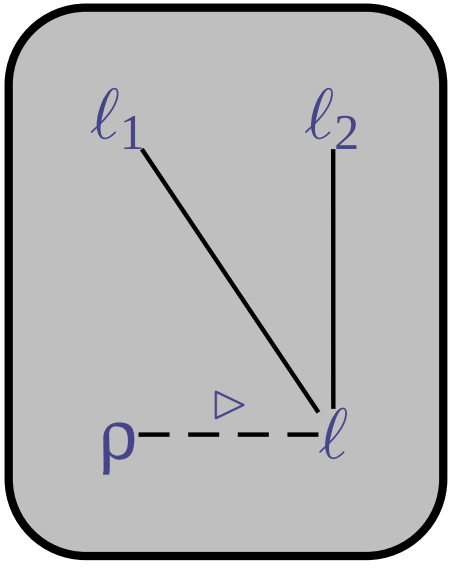
Linearity Example

```
pthread_mutex_t L1⟨ℓ1⟩, L2⟨ℓ2⟩, *l⟨ℓ⟩;
int x; // &x:  int*⟨ρx⟩

pthread_mutex_init(&L1);
pthread_mutex_init(&L2);
if(...)  l = &L1;
else  l = &L2;
pthread_mutex_lock(&l);
x = 3;
pthread_mutex_unlock(&l);
```



Linearity Example

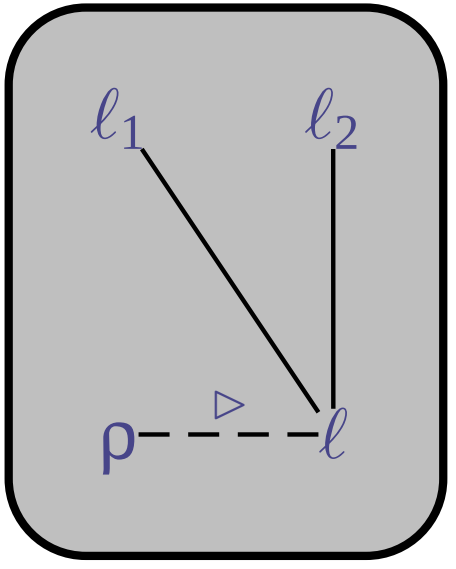
```
pthread_mutex_t L1⟨ℓ1⟩, L2⟨ℓ2⟩, *l⟨ℓ⟩;
int x; // &x:  int*⟨ρx⟩

pthread_mutex_init(&L1); {ℓ1}
pthread_mutex_init(&L2); {ℓ2}
if(...)  l = &L1; ∅
else l = &L2; ∅
pthread_mutex_lock(&l); ∅
x = 3;
pthread_mutex_unlock(&l); ∅
```



Linearity Example

```
pthread_mutex_t L1⟨ℓ1⟩, L2⟨ℓ2⟩, *l⟨ℓ⟩;
int x; // &x:  int*⟨ρx⟩

pthread_mutex_init(&L1); {ℓ1}
pthread_mutex_init(&L2); {ℓ2}
if(...)  l = &L1; 0
else l = &L2; 0
pthread_mutex_lock(&l); 0
x = 3;
pthread_mutex_unlock(&l); 0
```

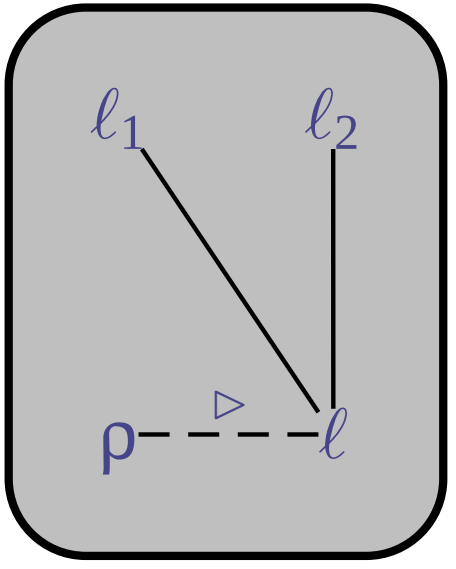
$\{\ell_1\} \uplus \{\ell_2\}$



Linearity Example

```
pthread_mutex_t L1⟨ℓ1⟩, L2⟨ℓ2⟩, *l⟨ℓ⟩;
int x; // &x:  int*⟨ρx⟩

pthread_mutex_init(&L1); {ℓ1}
pthread_mutex_init(&L2); {ℓ2}
if(...)  l = &L1; 0
else l = &L2; 0
pthread_mutex_lock(&l); 0
x = 3;
pthread_mutex_unlock(&l); 0
```

$\ell_1 \neq \ell_2$



Lock State

Create context sensitive control-flow graph:

- For every program point create a state variable ψ
- ψ nodes have kinds (Acquire, Release, Newlock, Deref, etc.)
- $\psi \longrightarrow \psi'$: control flow
- $\psi \xrightarrow{(_i} \psi'$: control enters function at the context of call site i
- $\psi \xrightarrow{)_i} \psi'$: function returns control at context of call site i

Lock State

- Data-flow analysis solving on the ψ control-flow graph
- Transfer function for every ψ depends on its kind
- Acquired set propagation across contexts uses $\xrightarrow{(_i}$ and $\xrightarrow{)_i}$ edges

Lock State Example

ψ_{in}

ψ_1

ψ_2

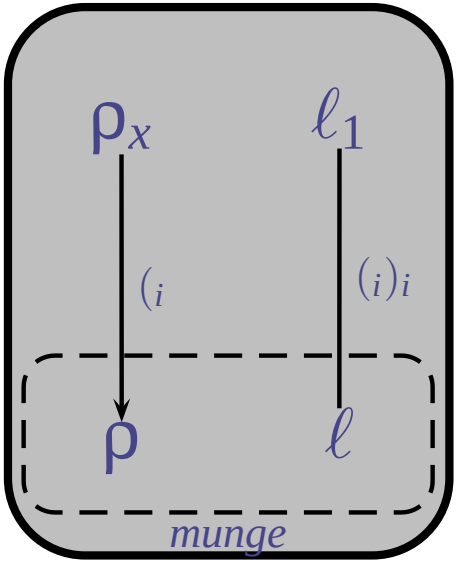
ψ_3

ψ_{out}

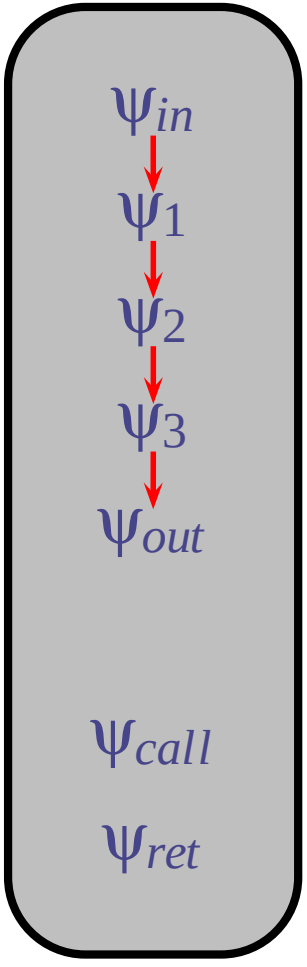
ψ_{call}

ψ_{ret}

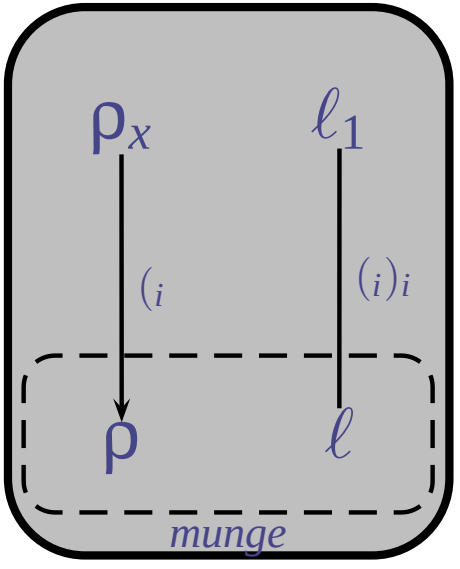
```
pthread_mutex_t⟨ℓ1⟩ L1 = ...;
int x; // &x:  int*⟨ρx⟩
void munge(pthread_mutex_t⟨ℓ⟩ *l, int *⟨ρ⟩ p) {
   pthread_mutex_lock(l);
   *p = 3;
   pthread_mutex_unlock(l);
}
...
munge^i(&L1, &x);
```



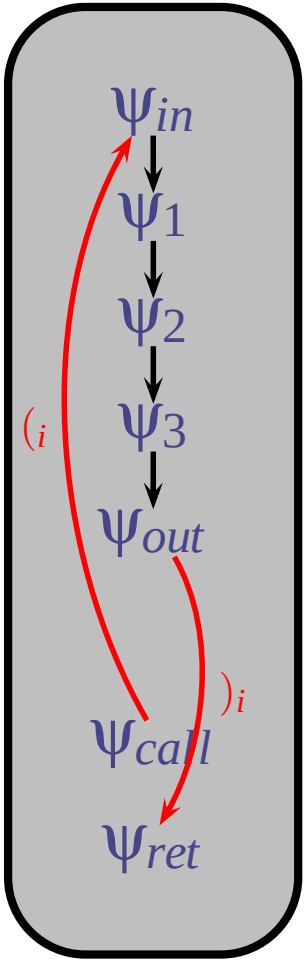
Lock State Example



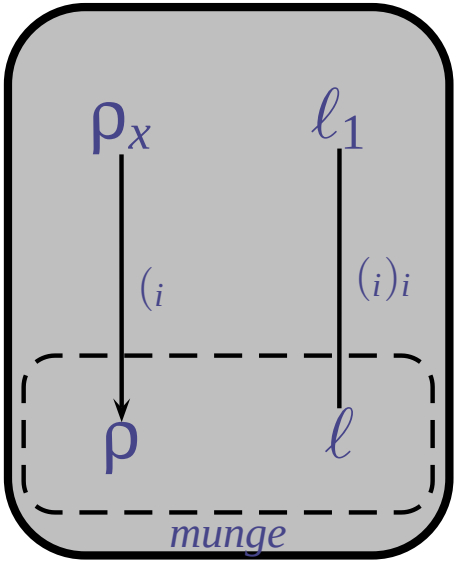

```
pthread_mutex_t⟨ℓ1⟩ L1 = ...;
int x; // &x:  int*⟨ρx⟩
void munge(pthread_mutex_t⟨ℓ⟩ *l, int *⟨ρ⟩ p) {
   pthread_mutex_lock(l);
   *p = 3;
   pthread_mutex_unlock(l);
}
...
munge^i(&L1, &x);
```



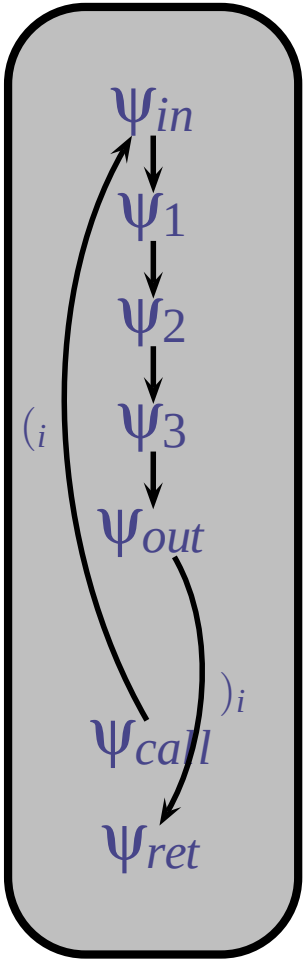
Lock State Example



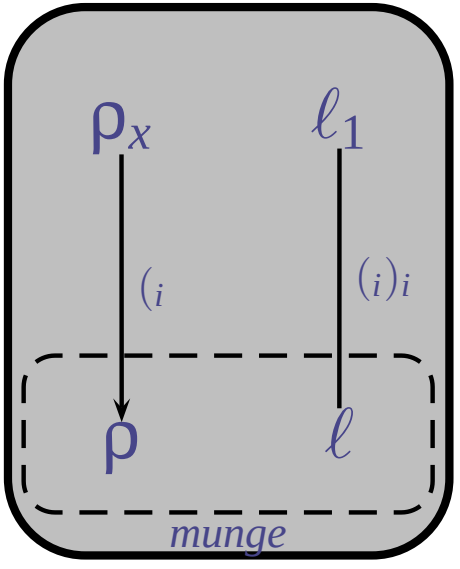

```
pthread_mutex_t⟨ℓ1⟩ L1 = ...;
int x; // &x:  int*⟨ρx⟩
void munge(pthread_mutex_t⟨ℓ⟩ *l, int *⟨ρ⟩ p) {
   pthread_mutex_lock(l);
   *p = 3;
   pthread_mutex_unlock(l);
}
...
munge^i(&L1, &x);
```



Lock State Example

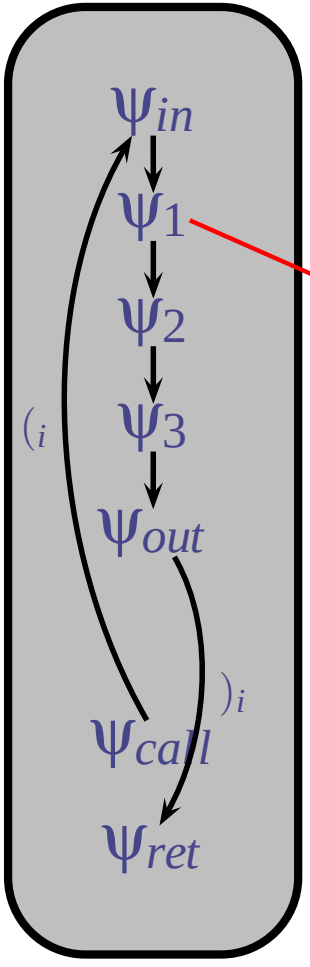


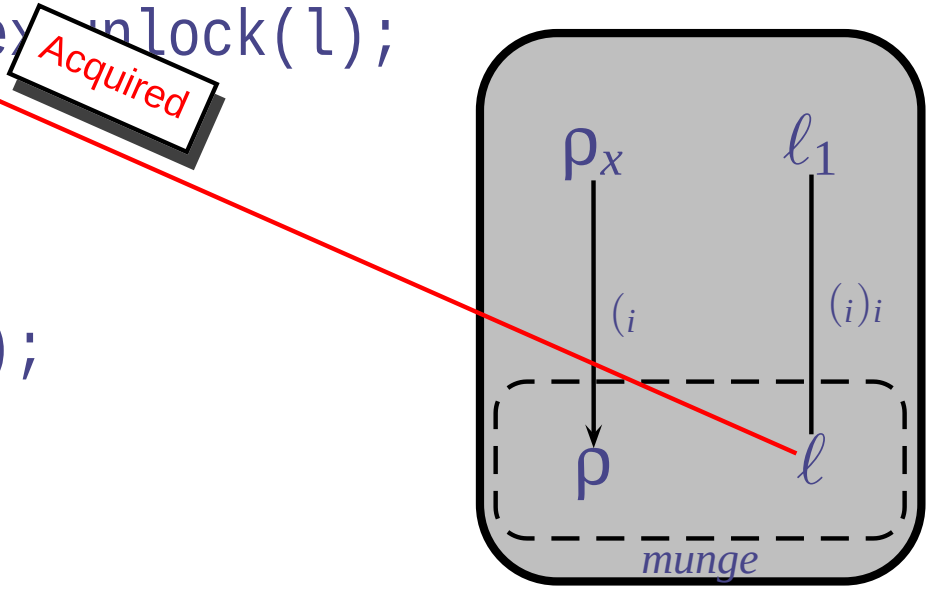

```
pthread_mutex_t⟨ℓ1⟩ L1 = ...;
int x; // &x:  int*⟨ρx⟩
void munge(pthread_mutex_t⟨ℓ⟩ *l, int *⟨ρ⟩ p) {
   pthread_mutex_lock(l);
   *p = 3;
   pthread_mutex_unlock(l);
}
...
munge^i(&L1, &x);
```



Lock State Example

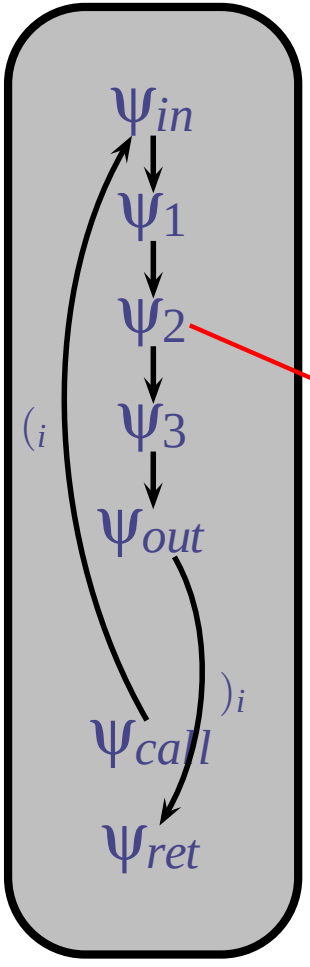
```
pthread_mutex_t⟨ℓ1⟩ L1 = ...;
int x; // &x:  int*⟨ρx⟩
void munge(pthread_mutex_t⟨ℓ⟩ *l, int *⟨ρ⟩ p) {
   pthread_mutex_lock(l);
   *p = 3;
   pthread_mutex_unlock(l);
}
...
munge^i(&L1, &x);
```



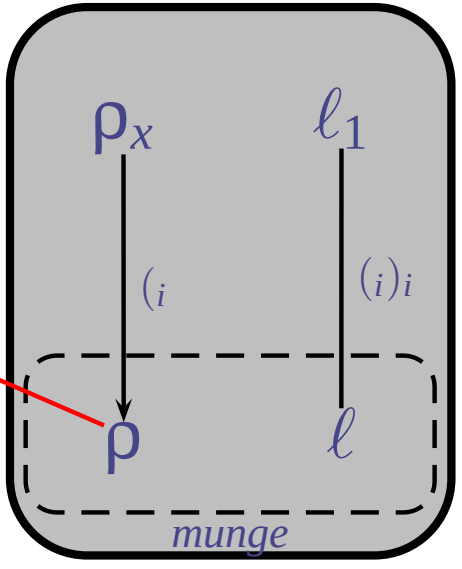


Lock State Example

```
pthread_mutex_t⟨ℓ1⟩ L1 = ...;
int x; // &x:  int*⟨ρx⟩
void munge(pthread_mutex_t⟨ℓ⟩ *l, int *⟨ρ⟩ p) {
   pthread_mutex_lock(l);
   *p = 3;
   pthread_mutex_unlock(l);
}
...
munge^i(&L1, &x);
```

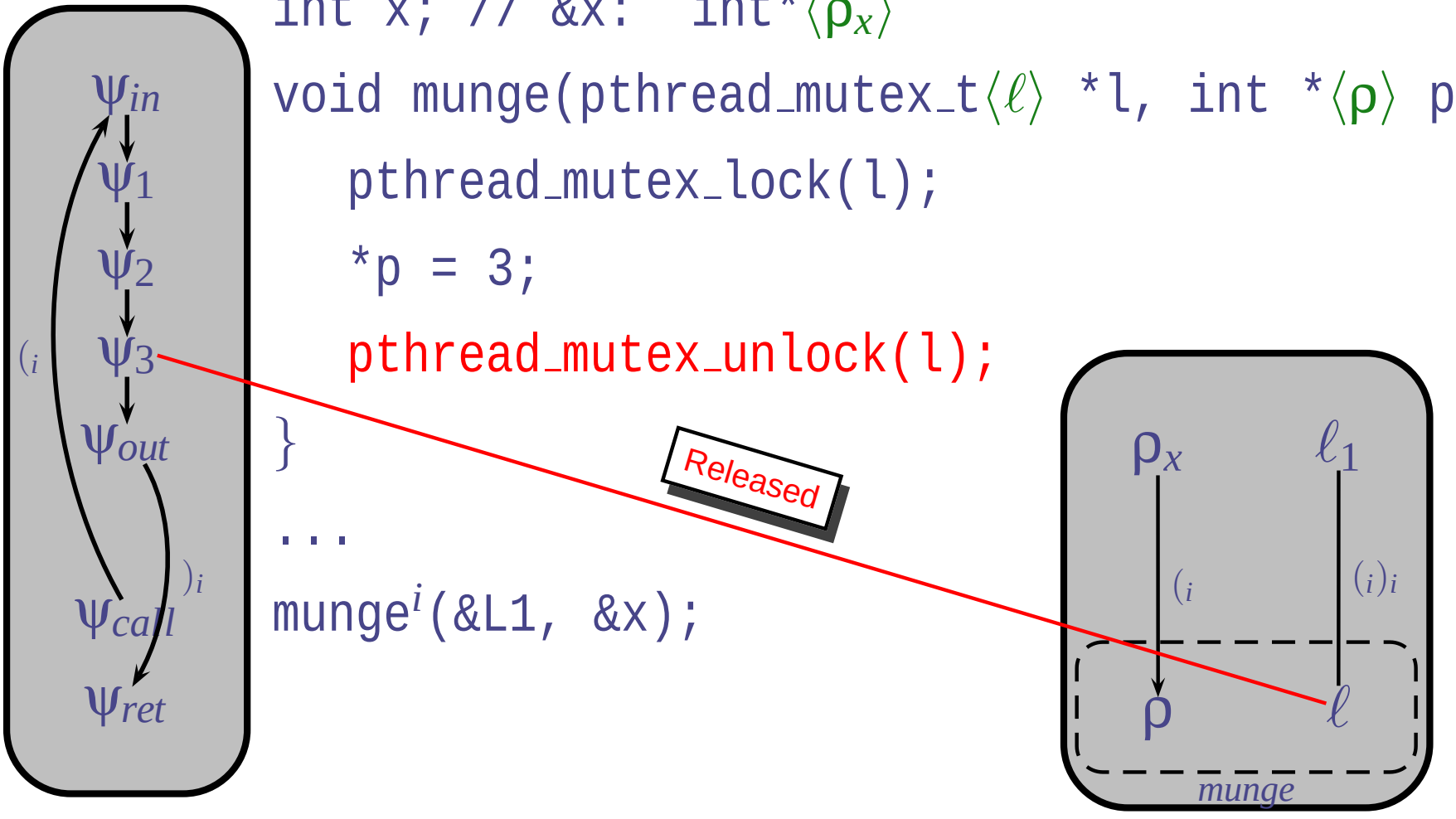


Dereferenced

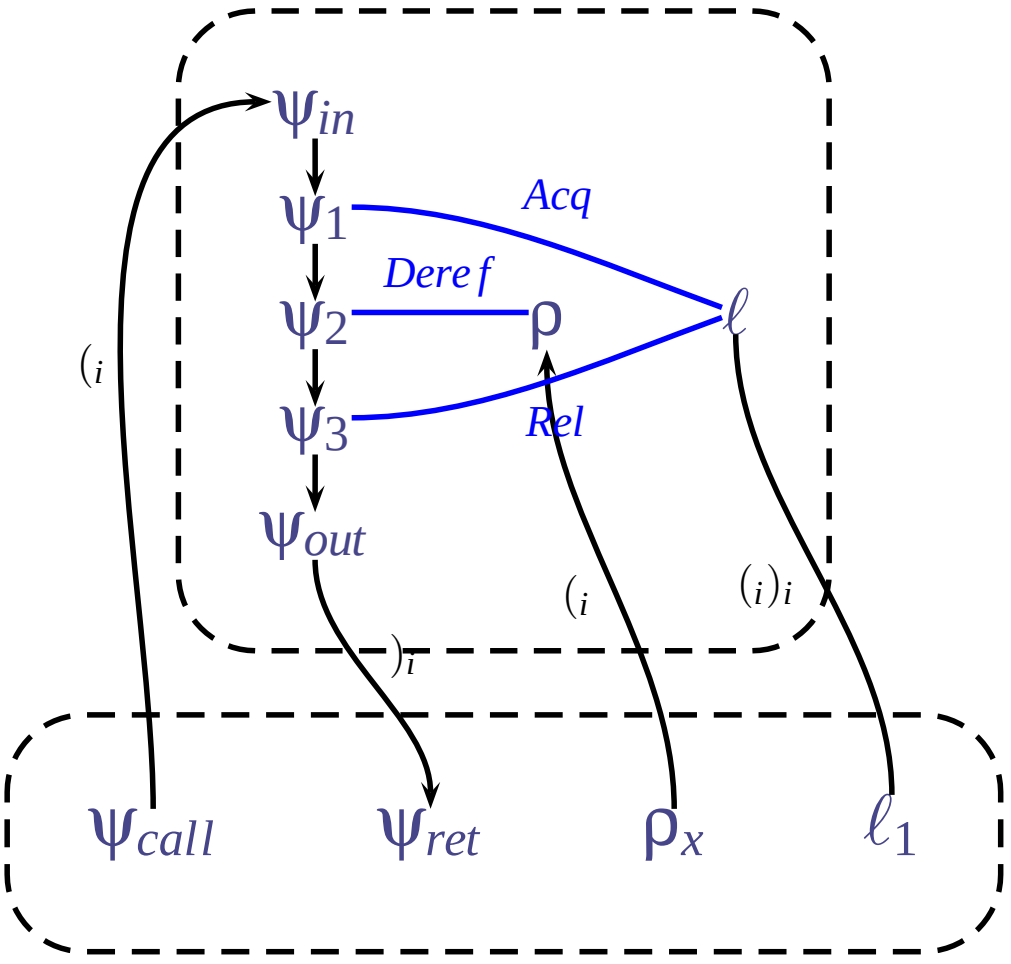


Lock State Example

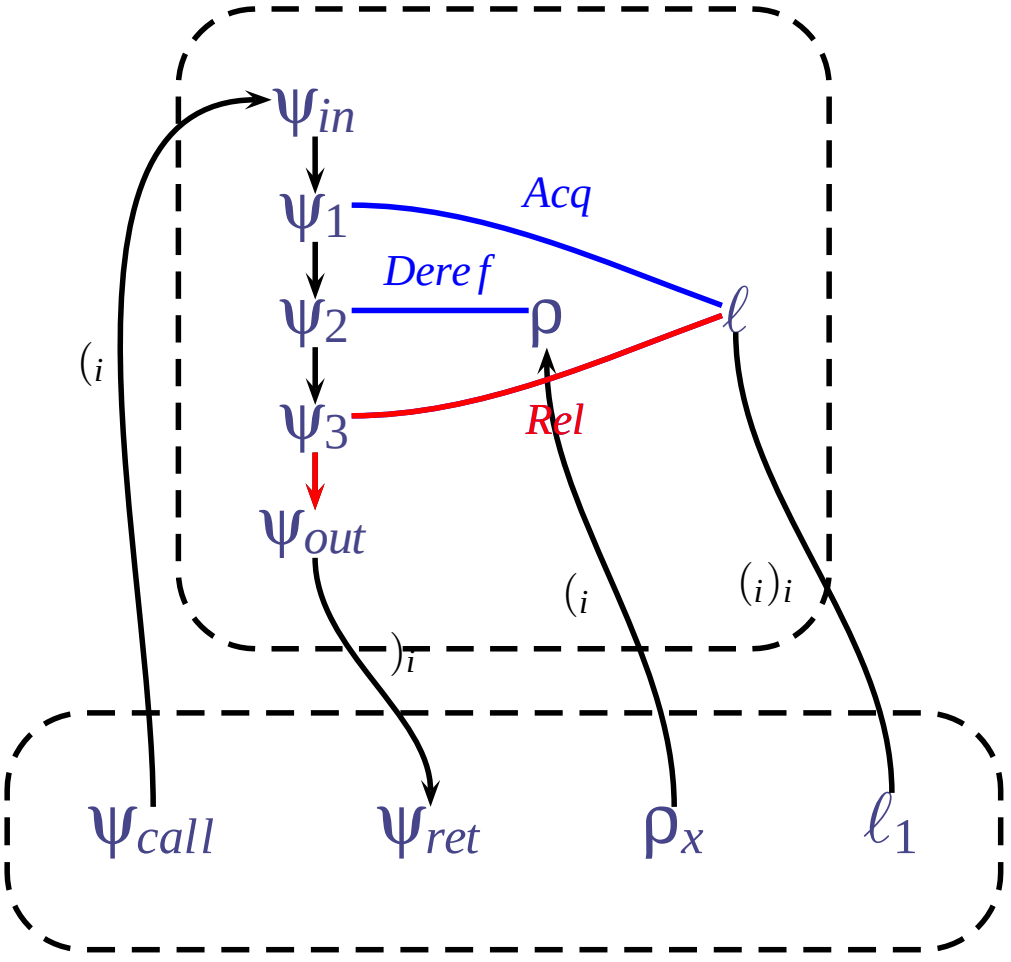
```
pthread_mutex_t⟨ℓ1⟩ L1 = ...;
int x; // &x:  int*⟨ρx⟩
void munge(pthread_mutex_t⟨ℓ⟩ *l, int *⟨ρ⟩ p) {
   pthread_mutex_lock(l);
   *p = 3;
   pthread_mutex_unlock(l);
}
...
munge^i(&L1, &x);
```



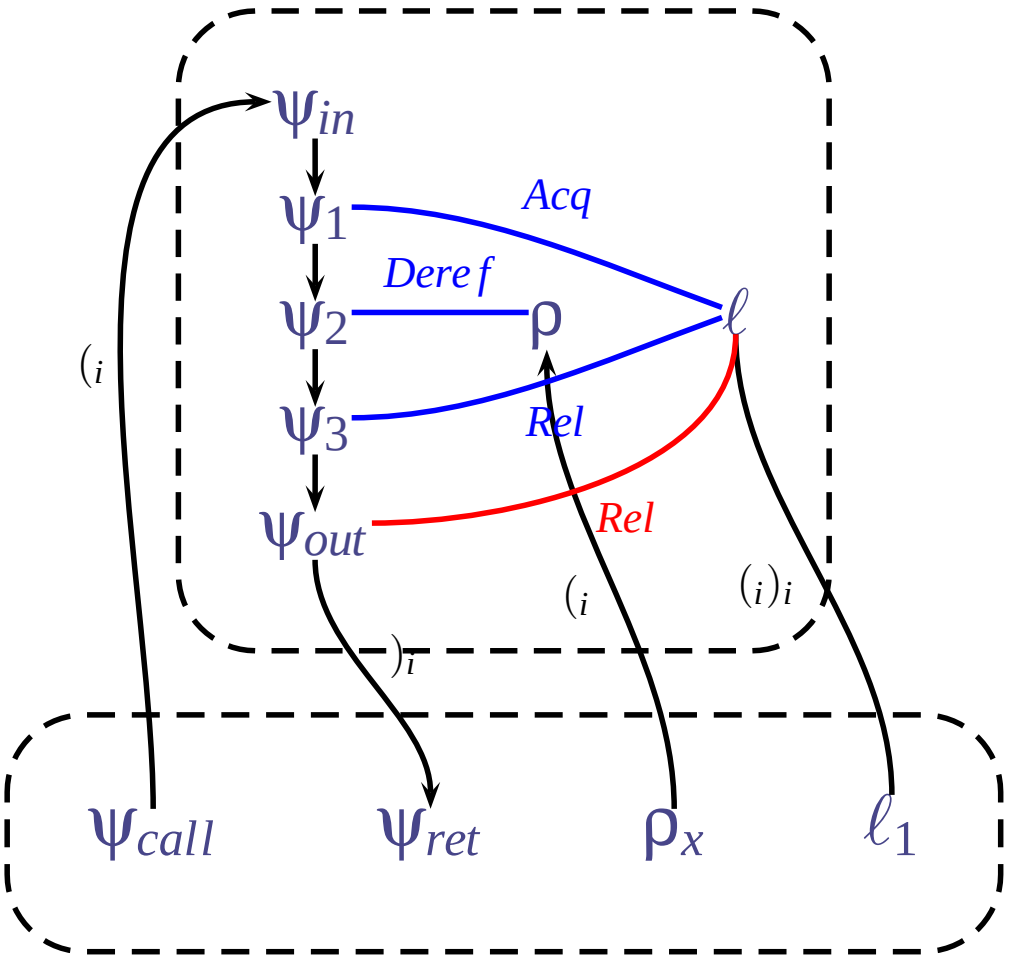
Lock State Example



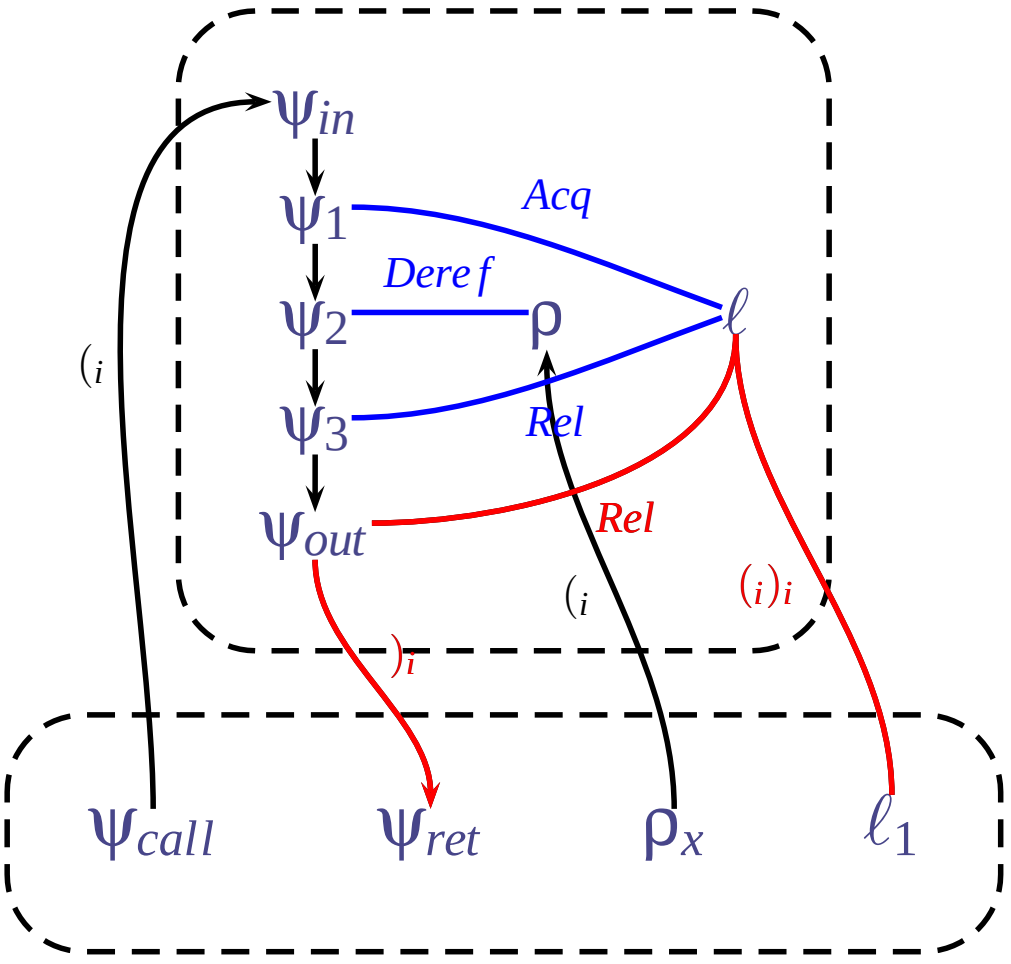
Lock State Example



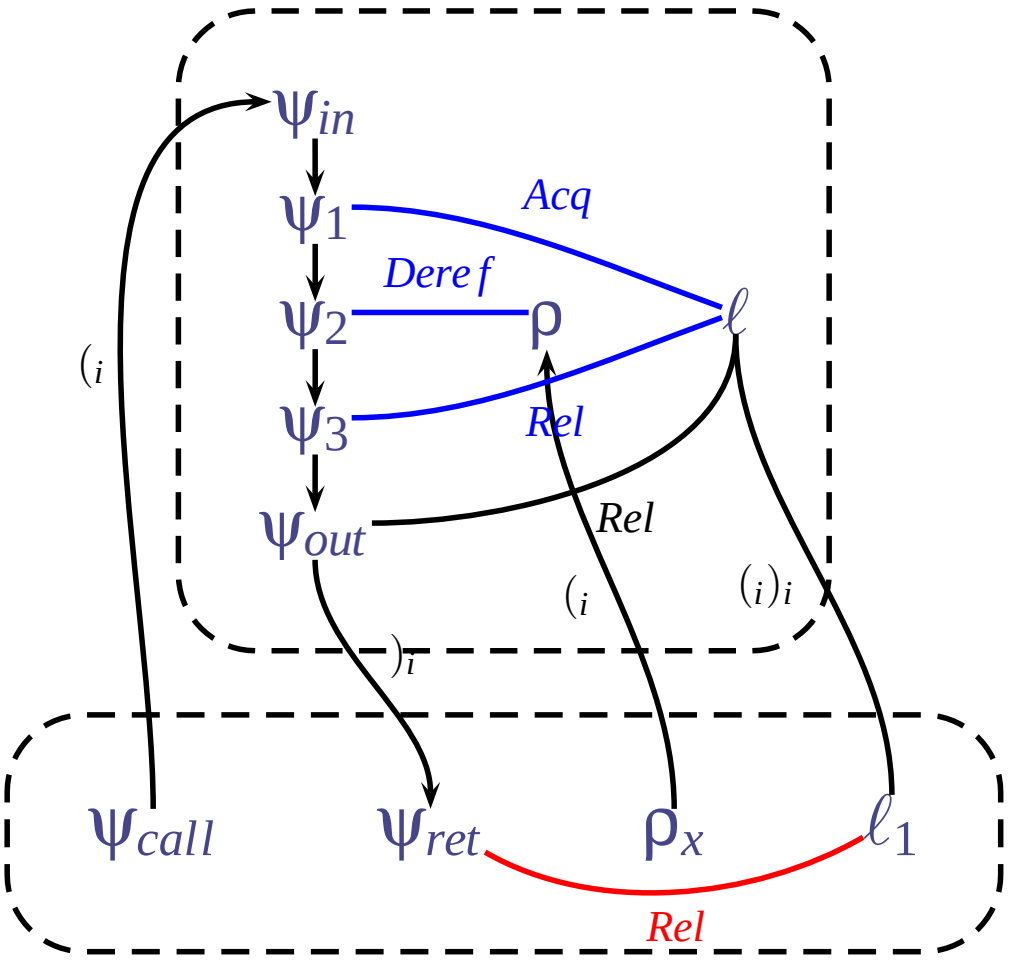
Lock State Example



Lock State Example



Lock State Example



Implementation

- LOCKSMITH: implementation for C
- Increase precision of the analysis using `void *` inference
- Reduce memory footprint with lazy `struct` field expansion
- Liveness/uniqueness analysis reduce false positives when an (eventually) shared location variable is provably still thread-local
- Continuation read/write effects used to precisely find shared locations at fork points

`void *`

- Each occurence of a `void *` pointer is annotated with a type list
- Any time a type τ is cast to or from a `void *`, we add τ to the list
- Track flow and instantiations of `void *` types
- Simple worklist algorithm:
  - For every list with > 2 elements, *conflate* all labels
  - Unify type lists if there is flow between `void *` types
  - For `void *` instantiation, unify with an instance of the type list
- For every *singleton type* `void *`, treat it as τ

Lazy `struct` fields

- Annotate every `struct` type with an empty list of fields
- Whenever a field is used, add it to the list
- Worklist solving algorithm:
  - Unify field lists if there is flow between `struct` types
  - Track instantiations of `struct` types and instantiate field lists

Uniqueness

```
int* shared; /* shared global pointer */
f() {
   int* x = (int *) malloc(sizeof(int));
   *x = 2; /* x is not yet shared */
   shared = x; /* x becomes shared */
}
```

- Very simple uniqueness analysis “filters” some unnecessary checks
- Reduced number of false positives
- Still sound

Experiments

Benchmark	Size (KLOC)	Time	Warn.	Unguarded	Races
aget	1.6	2s	7	2	2
knot	1.4	5s	8	8	8
ctrace	1.4	6s	3	3	1
freshclam	55	51m	11	0	0
esp	16.2	91s	19	1	1
plip	18.2	60s	1	0	0
synclink	23.6	16m	2	0	0
consolemap	14.1	1s	1	0	0
serial_core	15.2	26s	0	0	0
ide-disk	18.9	7s	0	0	0

Experiments

Benchmark	All off	Unique	`void*`	Exists
aget	9	9	7	—
knot	31	26	22	8
ctrace	4	4	3	—
freshclam	37	21	19	11
esp	37	21	19	—
plip	1	1	1	—
synclink	2	2	2	—
consolemap	1	1	1	—
serial_core	0	0	0	—
ide-disk	0	0	0	—

Conclusions - Future Work

Contribution:

- Formalized correlation inference system with universal and existential context sensitivity
- Proof of soundness
- Implementation for C

Future work:

- Apply correlation inference to other relations:
  - pointers correlated with allocation regions
  - arrays correlated with integer lengths
- Infer synchronization for programs with atomic regions

Formal language

$$
\begin{array}{rcl}
e & ::= & x \mid v \mid e_1\ e_2 \mid \texttt{if0}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\
 & \mid & (e_1, e_2) \mid e.j \mid \texttt{let}\ f = v\ \texttt{in}\ e_2 \mid \texttt{fix}\ f.v \mid f^i \\
 & \mid & \texttt{newlock} \mid \texttt{ref}\ e \mid !^{e_2} e_1 \mid e_1 :=^{e_3} e_2 \\
v & ::= & n \mid \lambda x.e \mid (v_1, v_2)
\end{array}
$$

Types

$$
\begin{array}{rcl}
\tau & ::= & \mathit{int} \mid \tau \times \tau \mid \tau \rightarrow^{\varepsilon} \tau' \\
 & \mid & \mathit{lock}\ \ell \mid \mathit{ref}^{\rho}\ \tau \\
l & ::= & \ell \mid \rho \\
\varepsilon & ::= & \emptyset \mid \{\ell\} \mid \chi \mid \varepsilon \uplus \varepsilon' \mid \varepsilon \cup \varepsilon' \\
\sigma & ::= & (\forall.\tau, \vec{l}) \\
C & ::= & \emptyset \mid \{c\} \mid C \cup C \\
c & ::= & \ell = \ell' \mid \rho \leq \rho' \mid \rho \rhd \ell \\
 & \mid & \varepsilon \leq \chi \mid \varepsilon \leq_{\vec{l}} \chi \mid \mathit{effect}(\tau) = \emptyset \\
 & \mid & \ell \preceq^{i} \ell' \mid \rho \preceq^{i}_{p} \rho' \mid \varepsilon \preceq^{i} \chi
\end{array}
$$

Type rules

$$\text{[Lam]}\ \frac{C;\Gamma,x:\tau \vdash e:\tau';\varepsilon \qquad C \vdash \varepsilon \leq \chi \qquad \chi \text{ fresh}}{C;\Gamma \vdash \lambda x.e:\tau \rightarrow^{\chi} \tau';\emptyset}$$

$$\text{[App]}\ \frac{\begin{array}{c} C;\Gamma \vdash e_1:\tau \rightarrow^{\varepsilon} \tau';\varepsilon_1 \\ C;\Gamma \vdash e_2:\tau;\varepsilon_2 \end{array}}{C;\Gamma \vdash e_1\ e_2:\tau';\varepsilon_1 \uplus \varepsilon_2 \uplus \varepsilon}$$

$$\text{[Newlock]}\ \frac{\ell \text{ fresh}}{C;\Gamma \vdash \texttt{newlock} : \mathit{lock}\ \ell;\{\ell\}}$$

Type rules

$$\text{[Ref]}\ \frac{C;\Gamma \vdash e : \tau;\varepsilon \qquad \rho \text{ fresh}}{C;\Gamma \vdash \mathsf{ref}\ e : \mathit{ref}^{\rho}\ \tau;\varepsilon}$$

$$\text{[Deref]}\ \frac{C;\Gamma \vdash e_1 : \mathit{ref}^{\rho}\ \tau;\varepsilon_1 \qquad C;\Gamma \vdash e_2 : \mathit{lock}\ \ell;\varepsilon_2 \qquad C \vdash \rho \rhd \ell}{C;\Gamma \vdash !^{e_2} e_1 : \tau;\varepsilon_1 \uplus \varepsilon_2}$$

$$\text{[Assign]}\ \frac{C;\Gamma \vdash e_1 : \mathit{ref}^{\rho}\ \tau;\varepsilon_1 \qquad C;\Gamma \vdash e_2 : \tau;\varepsilon_2 \qquad C;\Gamma \vdash e_3 : \mathit{lock}\ \ell;\varepsilon_3 \qquad C \vdash \rho \rhd \ell}{C;\Gamma \vdash e_1 :=^{e_3} e_2 : \tau;\varepsilon_1 \uplus \varepsilon_2 \uplus \varepsilon_3}$$

Polymorphic Type Rules

$$\text{[Let]}\ \frac{C;\Gamma \vdash v_1 : \tau_1;\emptyset \qquad \vec{l} = fl(\Gamma) \qquad C;\Gamma, f : (\forall.\tau_1, \vec{l}) \vdash e_2 : \tau_2;\varepsilon}{C;\Gamma \vdash \texttt{let}\ f = v_1\ \texttt{in}\ e_2 : \tau_2;\varepsilon}$$

$$\text{[Inst]}\ \frac{C \vdash \tau \preceq^i_+ \tau' \qquad C \vdash \vec{l} \preceq^i_\pm \vec{l}}{C;\Gamma, f : (\forall.\tau, \vec{l}) \vdash f^i : \tau';\emptyset}$$

Polymorphic Type Rules

$$\text{[Fix]}\ \frac{\begin{array}{c} C;\Gamma, f : (\forall.\tau, \vec{l}) \vdash v : \tau';\emptyset \qquad \vec{l} = \mathit{fl}(\Gamma) \\ C \vdash \tau' \leq \tau \qquad\qquad C \vdash \tau \preceq^{i}_{+} \tau'' \\ C \vdash \vec{l} \preceq^{i}_{\pm} \vec{l} \qquad\qquad C \vdash \mathit{effect}(\tau) = \emptyset \end{array}}{C;\Gamma \vdash \mathtt{fix}\, f.v : \tau'';\emptyset}$$

$$\text{[Down]}\ \frac{\begin{array}{c} C;\Gamma \vdash e : \tau;\varepsilon \qquad \vec{l} = \mathit{fl}(\Gamma) \cup \mathit{fl}(\tau) \\ C \vdash \varepsilon \leq_{\vec{l}} \chi \qquad\qquad \chi \text{ fresh} \end{array}}{C;\Gamma \vdash e : \tau;\chi}$$

Inst

$$\text{[Inst-Ref]}\ \frac{C \vdash \rho \preceq^{i}_{p} \rho' \qquad C \vdash \tau \preceq^{i}_{\pm} \tau'}{C \vdash \mathit{ref}^{\rho}\ \tau \preceq^{i}_{p} \mathit{ref}^{\rho'}\ \tau'}$$

$$\text{[Inst-Fun]}\ \frac{C \vdash \tau_1 \preceq^{i}_{\bar{p}} \tau_2 \qquad C \vdash \tau'_1 \preceq^{i}_{p} \tau'_2 \qquad C \vdash \varepsilon_1 \preceq^{i} \varepsilon_2}{C \vdash \tau_1 \rightarrow^{\varepsilon_1} \tau'_1 \preceq^{i}_{p} \tau_2 \rightarrow^{\varepsilon_2} \tau'_2}$$

Sub

$$\text{[Sub-Lock]}\ \frac{C \vdash \ell = \ell'}{C \vdash \mathit{lock}\ \ell \leq \mathit{lock}\ \ell'}$$

$$\text{[Sub-Ref]}\ \frac{C \vdash \rho \leq \rho' \quad C \vdash \tau \leq \tau' \quad C \vdash \tau' \leq \tau}{C \vdash \mathit{ref}^{\rho}\ \tau \leq \mathit{ref}^{\rho'}\ \tau'}$$

$$\text{[Sub-Fun]}\ \frac{C \vdash \tau_2 \leq \tau_1 \quad C \vdash \tau_1' \leq \tau_2' \quad C \vdash \varepsilon_1 \leq \varepsilon_2}{C \vdash \tau_1 \rightarrow^{\varepsilon_1} \tau_1' \leq \tau_2 \rightarrow^{\varepsilon_2} \tau_2'}$$

Constraint Resolution

$$C \cup \{\ell = \ell'\} \quad \Rightarrow \quad C[\ell \mapsto \ell']$$

$$C \cup \{\rho_0 \leq \rho_1\} \cup \{\rho_1 \leq \rho_2\} \quad \cup \Rightarrow \quad \{\rho_0 \leq \rho_2\}$$

$$C \cup \{\ell_0 \preceq^i \ell_1\} \cup \{\ell_0 \preceq^i \ell_2\} \quad \Rightarrow \quad C[\ell_2 \mapsto \ell_1] \cup \{\ell_0 \preceq^i \ell_1\}$$

$$C \cup \{\rho_1 \preceq^i_- \rho_0\} \cup \{\rho_1 \leq \rho_2\} \cup \{\rho_2 \preceq^i_+ \rho_3\} \quad \cup \Rightarrow \quad \{\rho_0 \leq \rho_3\}$$

$$C \cup \{\rho \leq \rho'\} \cup \{\rho' \rhd \ell\} \quad \cup \Rightarrow \quad \{\rho \rhd \ell\}$$

$$C \cup \{\rho \preceq^i_p \rho'\} \cup \{\rho \rhd \ell\} \cup \{\ell \preceq^i \ell'\} \quad \cup \Rightarrow \quad \{\rho' \rhd \ell'\}$$

Constraint Resolution

$$
\begin{aligned}
C \cup \{\emptyset \leq \chi\} &\Rightarrow C \\
C \cup \{\varepsilon \cup \varepsilon' \leq \chi\} &\Rightarrow C \cup \{\varepsilon \leq \chi\} \cup \{\varepsilon' \leq \chi\} \\
C \cup \{\varepsilon \leq \chi\} \cup \{\chi \leq \chi'\} &\cup\Rightarrow \{\varepsilon \leq \chi'\} \\
C \cup \{\varepsilon \leq \chi\} \cup \{\chi \leq_{\vec{l}} \chi'\} &\cup\Rightarrow \{\varepsilon \leq_{\vec{l}} \chi'\} \\
C \cup \{\varepsilon \leq \chi\} \cup \{\chi \preceq^{i} \chi'\} &\cup\Rightarrow \{\varepsilon \preceq^{i} \chi'\}
\end{aligned}
$$

Constraint Resolution

$$
\begin{array}{rcl}
C \cup \{\emptyset \preceq^i \chi\} & \Rightarrow & C \\
C \cup \{\{\ell\} \preceq^i \chi\} & \Rightarrow & C \cup \{\ell \preceq^i \ell'\} \cup \{\ell' \leq \chi\} \\
 & & \ell' \text{ fresh} \\
C \cup \{\varepsilon \uplus \varepsilon' \preceq^i \chi_0\} & \Rightarrow & C \cup \{\varepsilon \preceq^i \chi\} \cup \{\varepsilon' \preceq^i \chi'\} \\
 & & \cup \{\chi \uplus \chi' \leq \chi_0\} \\
 & & \chi, \chi' \text{ fresh} \\
C \cup \{\varepsilon \cup \varepsilon' \preceq^i \chi\} & \Rightarrow & C \cup \{\varepsilon \preceq^i \chi\} \cup \{\varepsilon' \preceq^i \chi\} \\
C \cup \{\chi \preceq^i \chi'\} \cup \{\chi \preceq^i \chi''\} & \Rightarrow & C[\chi' \mapsto \chi''] \cup \{\chi \preceq^i \chi'\}
\end{array}
$$

Constraint Resolution

$$
\begin{aligned}
C \cup \{\emptyset \leq_{\vec{l}} \chi\} &\Rightarrow C \\
C \cup \{\{\ell\} \leq_{\vec{l}} \chi\} &\Rightarrow C \cup \{\{\ell\} \leq \chi\} \\
&\qquad \text{if } C \vdash \mathit{escapes}(\ell, \vec{l}) \\
C \cup \{\varepsilon \uplus \varepsilon' \leq_{\vec{l}} \chi_0\} &\Rightarrow C \cup \{\varepsilon \leq_{\vec{l}} \chi\} \cup \{\varepsilon' \leq_{\vec{l}} \chi'\} \\
&\qquad \cup \{\chi \uplus \chi' \leq \chi_0\} \\
C \cup \{\varepsilon \cup \varepsilon' \leq_{\vec{l}} \chi\} &\Rightarrow C \cup \{\varepsilon \leq_{\vec{l}} \chi\} \cup \{\varepsilon' \leq_{\vec{l}} \chi\}
\end{aligned}
$$

void * results

Benchmark	Total	Empty	Single Type	Multiple Types
aget	148	56	78	14
knot	181	87	80	14
ctrace	115	57	58	0
freshclam	5745	1025	2365	2355
esp	754	476	188	90
plip	1310	1187	106	17
synclink	1100	701	151	248
consolemap	110	18	37	55
serial_core	1206	1001	77	128
ide-disk	447	252	97	98

Lazy `struct` fields results

Benchmark	Total Structs	Total Fields	Used Fields
aget	159	1955	252
knot	210	1418	619
ctrace	162	1113	389
freshclam	6982	65582	32822
esp	2439	37286	16462
plip	4119	89343	17706
synclink	7721	197872	111368
consolemap	145	3819	301
serial_core	7463	114149	29238
ide-disk	2352	72923	11011