

CONTEXT-SENSITIVE CORRELATION ANALYSIS FOR RACE DETECTION

Polyvios Pratikakis

Jeff Foster

Michael Hicks

University of Maryland, College Park

Data Races

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

A way to prevent races

Intuitively:

- For every dereference, correlate pointer with the acquired locks
- For every shared pointer, intersect locksets of all dereferences
- Verify that all shared pointers are protected.

A way to prevent races

Formally:

- 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*:
Location ρ is *always* correlated with lock ℓ
- Assert that every shared location ρ is *consistently correlated* with a lock ℓ

LOCKSMITH: static correlation inference

- ρ and ℓ are static approximations of run-time values
 - Sound, conservative
- Alias analysis:
 - Context-sensitive, flow-insensitive
 - May-alias for locations ρ , must-alias for locks ℓ
- Correlation $\rho \triangleright \ell$ inference
 - Every access creates a $\rho \triangleright \ell$ constraint
 - Infer all other $\rho \triangleright \ell$ relations by closing the constraints
- Consistent correlation
 - Verify consistent correlation for every shared ρ , or report a contradiction (race)

Contributions

- Static analysis for inference of *correlation* between locks and pointers
- Context sensitivity
 - Universal polymorphism for function calls
 - Existential polymorphism for data structures
- *Sound* race detection using assertion of *consistent correlation*
 - Formalised for λ_{\triangleright} , proof of soundness
- LOCKSMITH: Implementation for C

Type Based Analysis

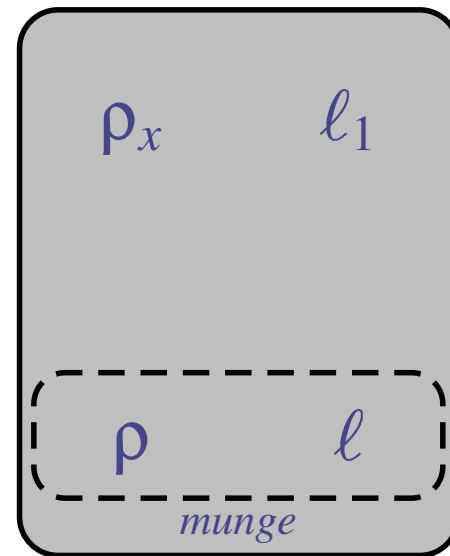
- Annotate types with labels:
 - $\text{pthread_mutex_t} \rightarrow \text{pthread_mutex_t} \langle \ell \rangle$
 - $\tau^* \rightarrow \tau^* \langle \rho \rangle$
- Create constraints among labels to capture data flow and correlation
 - Dereferencing ρ while ℓ is held: $\rho \triangleright \ell$
 - 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);
```

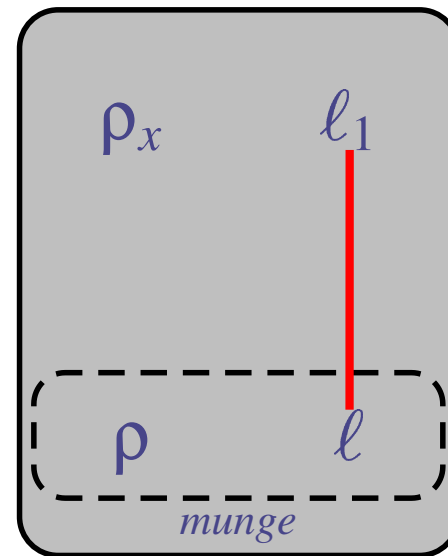
Correlation

```
pthread_mutex_t  $\langle \ell_1 \rangle$  L1 = ...;  
int x; // &x: int*  $\langle \rho_x \rangle$   
void munge(pthread_mutex_t  $\langle \ell \rangle$  *l, int *  $\langle \rho \rangle$  p) {  
    pthread_mutex_lock(l);  
    *p = 3;  
    pthread_mutex_unlock(l);  
}  
...  
munge(&L1, &x);
```



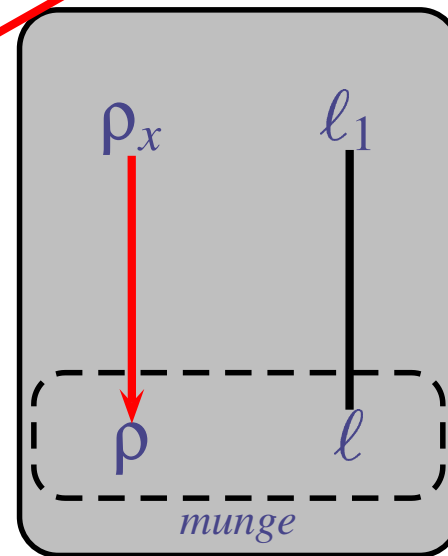
Correlation

```
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);
```



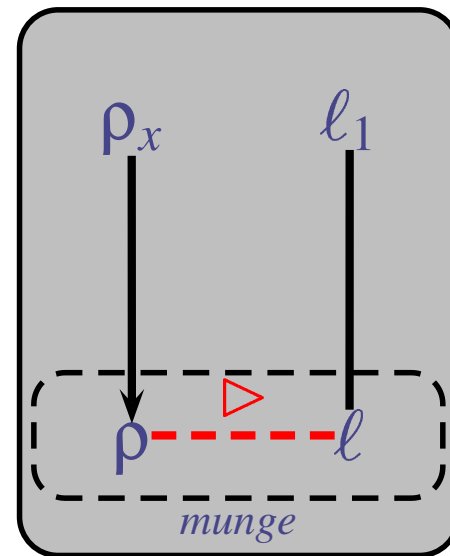
Correlation

```
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);
```



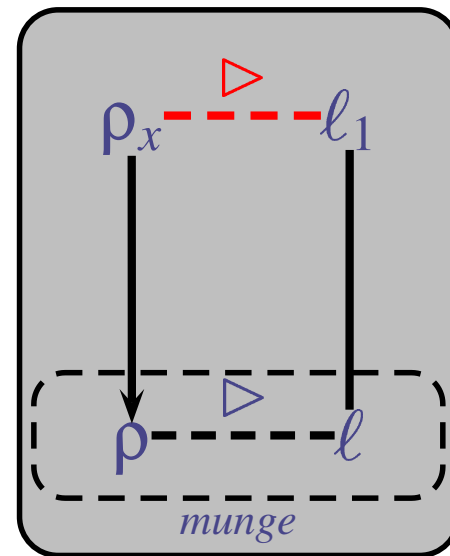
Correlation

```
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);
```



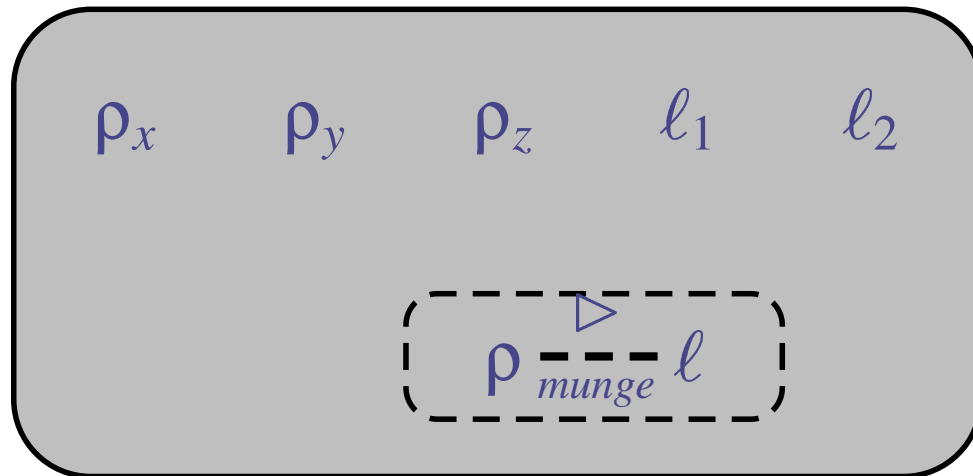
Correlation

```
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);
```



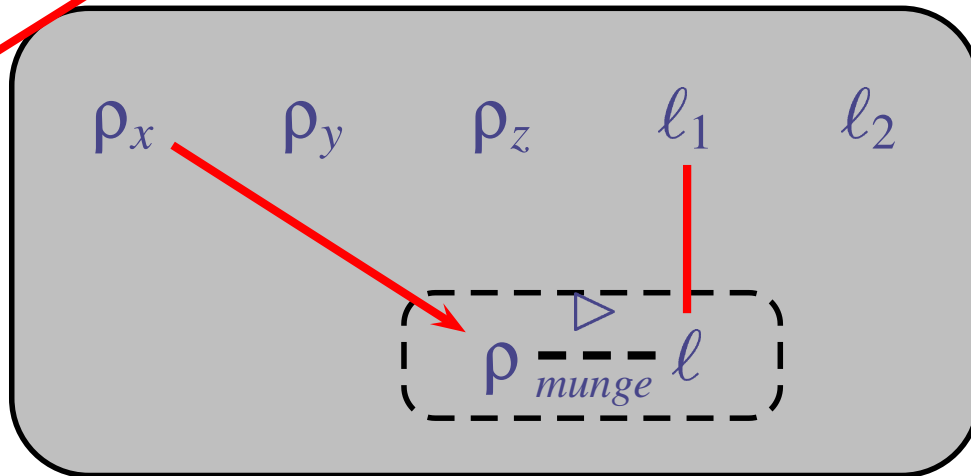
Context Sensitivity

```
pthread_mutex_t  $\langle \ell_1 \rangle$  L1 = ...,  $\langle \ell_2 \rangle$  L2 = ...;
int x, y, z; //  $\langle \rho_x \rangle$ ,  $\langle \rho_y \rangle$ ,  $\langle \rho_z \rangle$ 
void munge(pthread_mutex_t  $\langle \ell \rangle$  *l, int * $\langle \rho \rangle$  p) {
    pthread_mutex_lock(l);
    *p = 3;
    pthread_mutex_unlock(l);
}
...
munge (&L1, &x);
munge (&L2, &y);
munge (&L2, &z);
```



Context Sensitivity

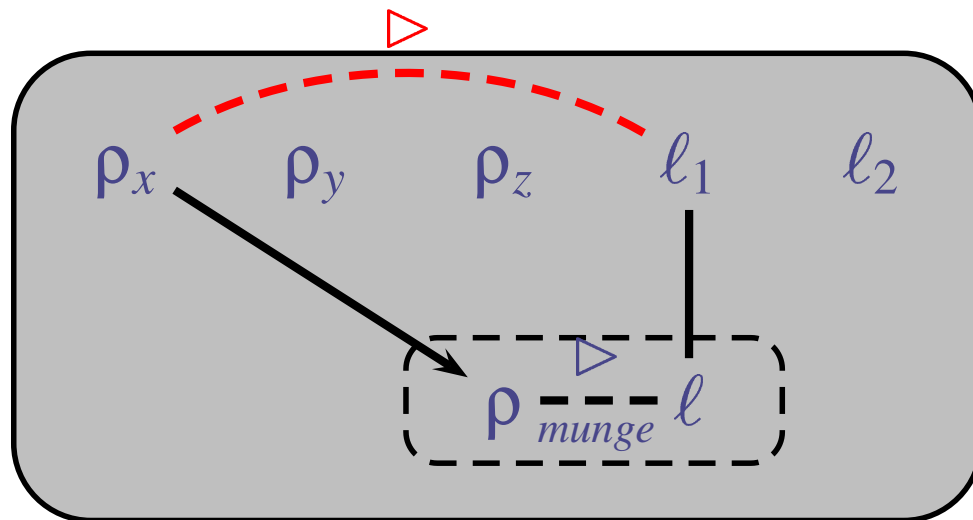
```
pthread_mutex_t<ℓ1> L1 = ..., <ℓ2> L2 = ...;  
int x, y, z; // <ρx>, <ρy>, <ρz>  
void munge(pthread_mutex_t<ℓ> *l, int *<ρ> p) {  
    pthread_mutex_lock(l);  
    *p = 3;  
    pthread_mutex_unlock(l);  
}  
...  
munge (&L1, &x);  
munge (&L2, &y);  
munge (&L2, &z);
```



Context Sensitivity

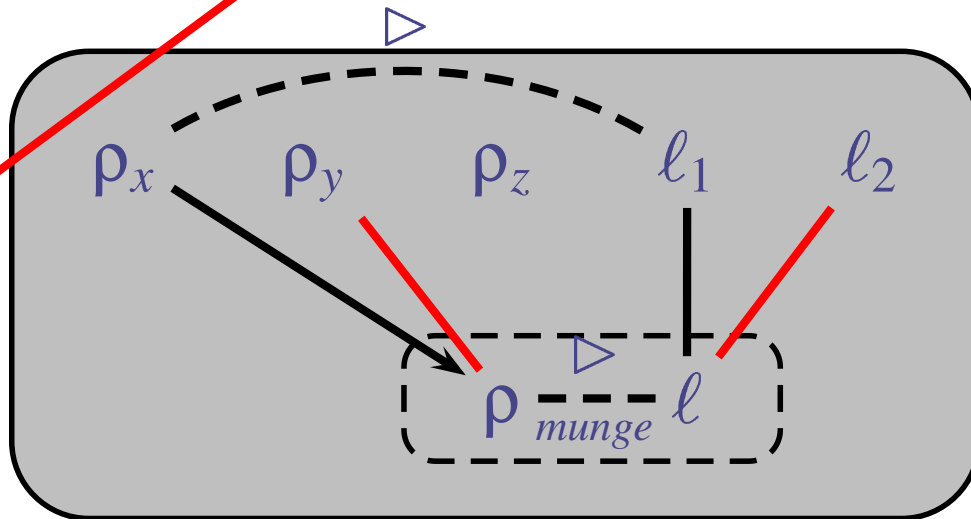
```
pthread_mutex_t  $\langle l_1 \rangle$  L1 = ...,  $\langle l_2 \rangle$  L2 = ...;  
int x, y, z; //  $\langle \rho_x \rangle$ ,  $\langle \rho_y \rangle$ ,  $\langle \rho_z \rangle$   
void munge(pthread_mutex_t  $\langle l \rangle$  *l, int * $\langle \rho \rangle$  p) {  
    pthread_mutex_lock(l);  
    *p = 3;  
    pthread_mutex_unlock(l);  
}
```

```
...  
munge (&L1, &x);  
munge (&L2, &y);  
munge (&L2, &z);
```



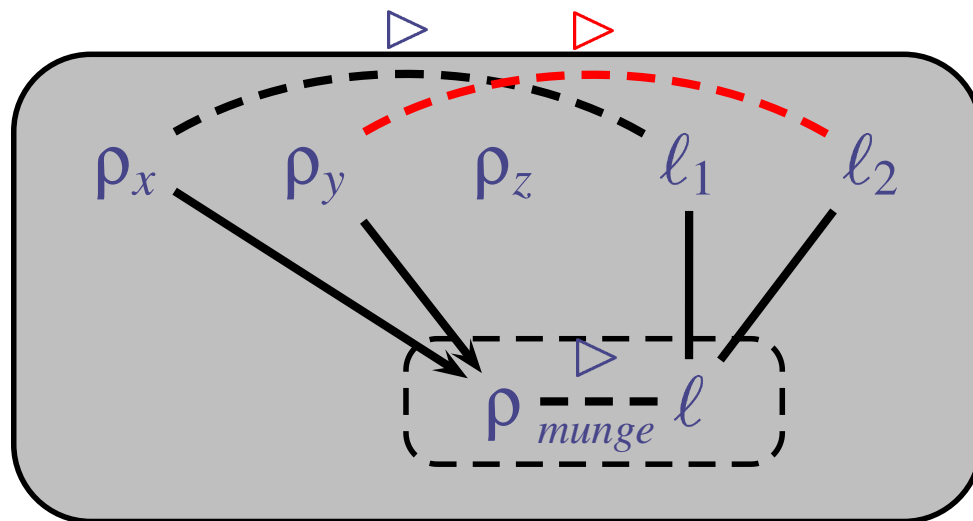
Context Sensitivity

```
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);
```



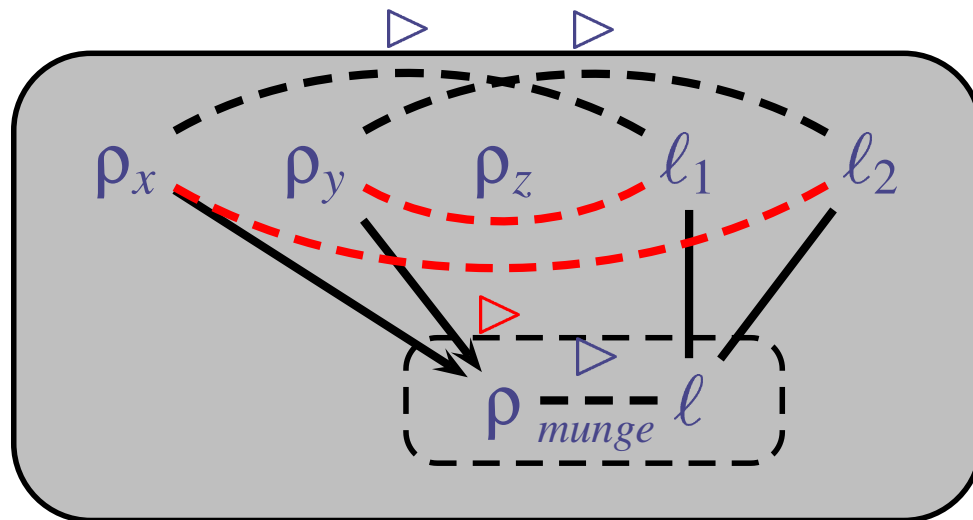
Context Sensitivity

```
pthread_mutex_t  $\langle l_1 \rangle$  L1 = ...,  $\langle l_2 \rangle$  L2 = ...;
int x, y, z; //  $\langle \rho_x \rangle$ ,  $\langle \rho_y \rangle$ ,  $\langle \rho_z \rangle$ 
void munge(pthread_mutex_t  $\langle l \rangle$  *l, int * $\langle \rho \rangle$  p) {
    pthread_mutex_lock(l);
    *p = 3;
    pthread_mutex_unlock(l);
}
...
munge (&L1, &x);
munge (&L2, &y);
munge (&L2, &z);
```



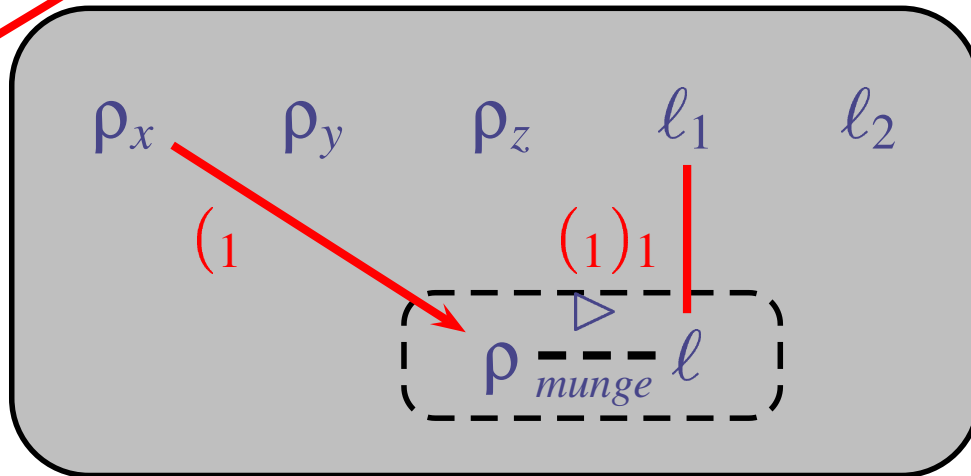
Context Sensitivity

```
pthread_mutex_t  $\langle l_1 \rangle$  L1 = ...,  $\langle l_2 \rangle$  L2 = ...;  
int x, y, z; //  $\langle \rho_x \rangle$ ,  $\langle \rho_y \rangle$ ,  $\langle \rho_z \rangle$   
void munge(pthread_mutex_t  $\langle l \rangle$  *l, int * $\langle \rho \rangle$  p) {  
    pthread_mutex_lock(l);  
    *p = 3;  
    pthread_mutex_unlock(l);  
}  
...  
munge (&L1, &x);  
munge (&L2, &y);  
munge (&L2, &z);
```



Context Sensitivity

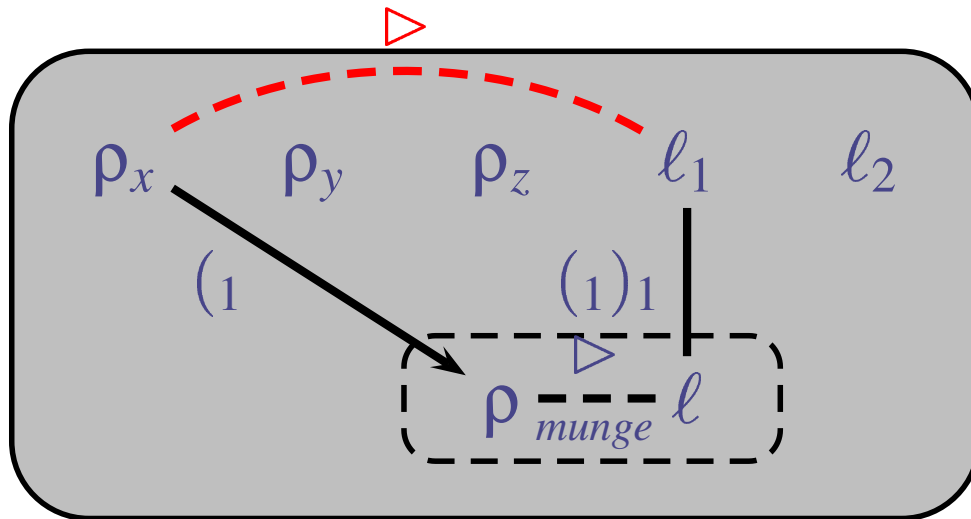
```
pthread_mutex_t<ℓ1> L1 = ..., <ℓ2> L2 = ...;  
int x, y, z; // <ρx>, <ρy>, <ρz>  
void munge(pthread_mutex_t<ℓ> *l, int *<ρ> p) {  
    pthread_mutex_lock(l);  
    *p = 3;           (1)  
    pthread_mutex_unlock(l);  
}  
...  
munge1 (&L1, &x);  
munge2 (&L2, &y);  
munge3 (&L2, &z);
```



Context Sensitivity

```
pthread_mutex_t  $\langle l_1 \rangle$  L1 = ...,  $\langle l_2 \rangle$  L2 = ...;  
int x, y, z; //  $\langle \rho_x \rangle$ ,  $\langle \rho_y \rangle$ ,  $\langle \rho_z \rangle$   
void munge(pthread_mutex_t  $\langle l \rangle$  *l, int * $\langle \rho \rangle$  p) {  
    pthread_mutex_lock(l);  
    *p = 3;  
    pthread_mutex_unlock(l);  
}
```

```
...  
munge1 (&L1, &x);  
munge2 (&L2, &y);  
munge3 (&L2, &z);
```

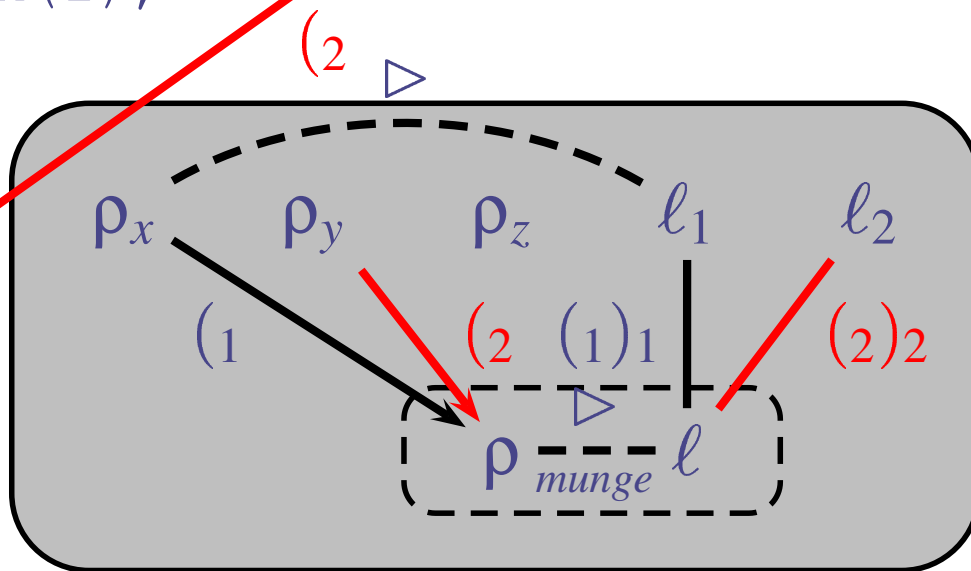


Context Sensitivity

```

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);

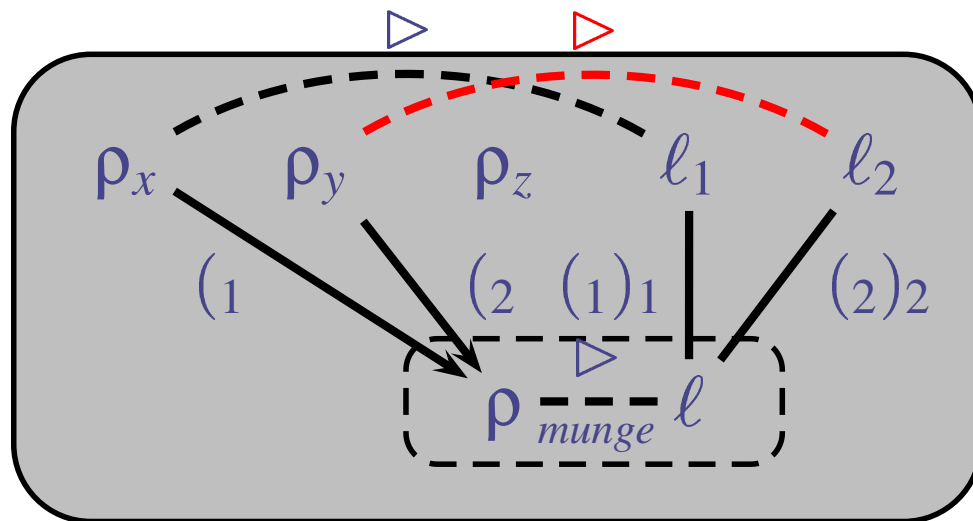
```



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);
```

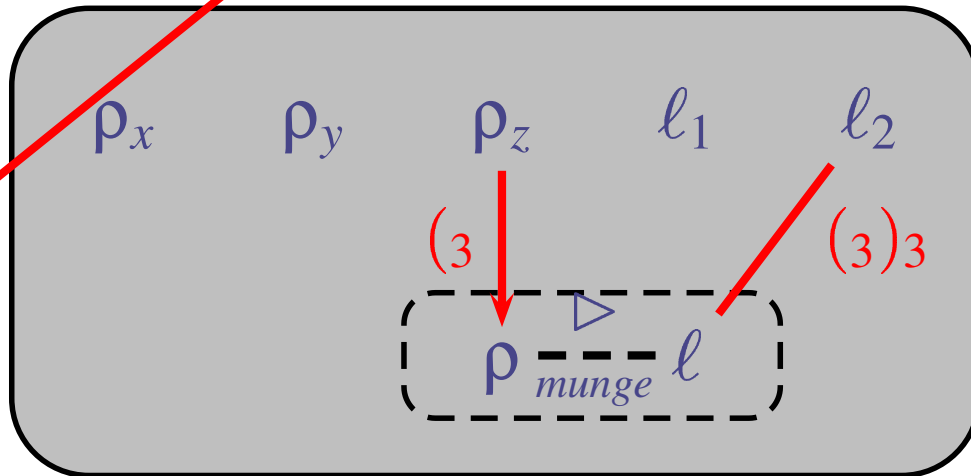


Context Sensitivity

```

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); (3)
}
...
munge1 (&L1, &x);
munge2 (&L2, &y);
munge3 (&L2, &z);

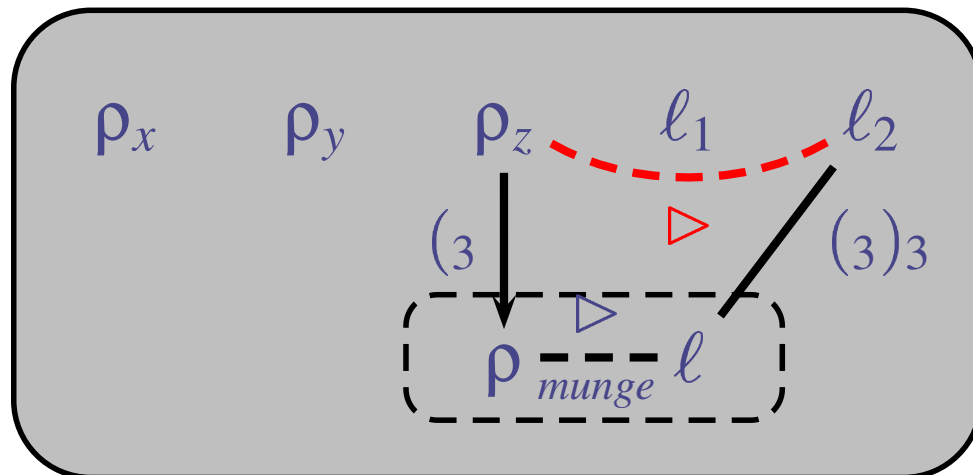
```



Context Sensitivity

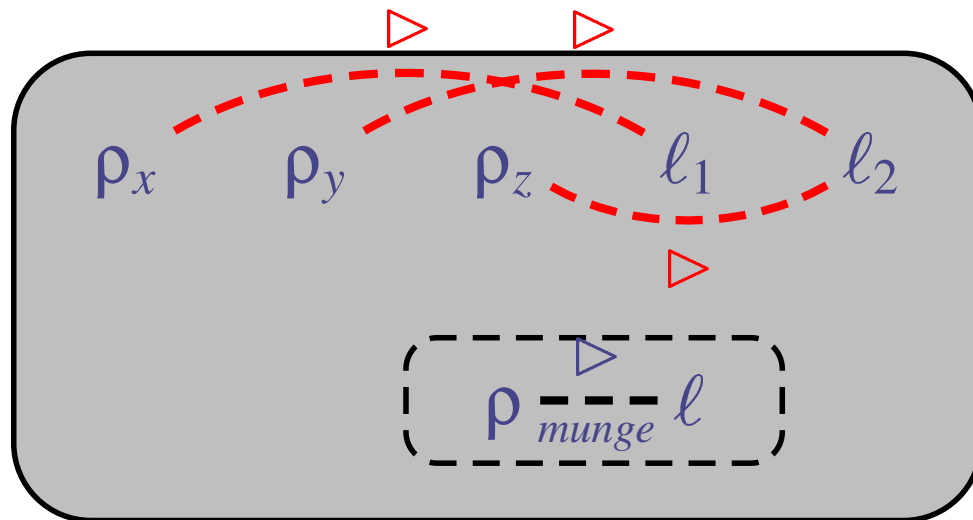
```
pthread_mutex_t  $\langle \ell_1 \rangle$  L1 = ...,  $\langle \ell_2 \rangle$  L2 = ...;  
int x, y, z; //  $\langle \rho_x \rangle$ ,  $\langle \rho_y \rangle$ ,  $\langle \rho_z \rangle$   
void munge(pthread_mutex_t  $\langle \ell \rangle$  *l, int * $\langle \rho \rangle$  p) {  
    pthread_mutex_lock(l);  
    *p = 3;  
    pthread_mutex_unlock(l);  
}
```

```
...  
munge1 (&L1, &x);  
munge2 (&L2, &y);  
munge3 (&L2, &z);
```



Context Sensitivity

```
pthread_mutex_t  $\langle \ell_1 \rangle$  L1 = ...,  $\langle \ell_2 \rangle$  L2 = ...;
int x, y, z; //  $\langle \rho_x \rangle$ ,  $\langle \rho_y \rangle$ ,  $\langle \rho_z \rangle$ 
void munge(pthread_mutex_t  $\langle \ell \rangle$  *l, int * $\langle \rho \rangle$  p) {
    pthread_mutex_lock(l);
    *p = 3;
    pthread_mutex_unlock(l);
}
...
munge1 (&L1, &x);
munge2 (&L2, &y);
munge3 (&L2, &z);
```



Linearity of locks

- Each lock label ℓ might represent more than one run-time locks.
- Then:
 - Which one is correlated with ρ in $\rho \triangleright \ell$?
 - Which one gets acquired by `pthread_mutex_lock`?
- So, locks ℓ have to be linear (must alias)
- Challenges:
 - Dynamic allocation of locks
 - Want to avoid being overly conservative in loops

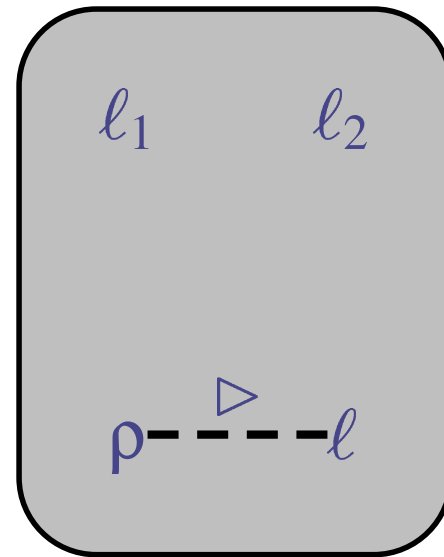
Linearity Effects

- 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* of effects
- Filter effects to remove any ℓ that does not escape

Linearity Example

```
pthread_mutex_t L1<math>\ell_1>, L2<math>\ell_2>, *l<math>\ell>;  
int x; // &x: int*<math>\rho_x>
```

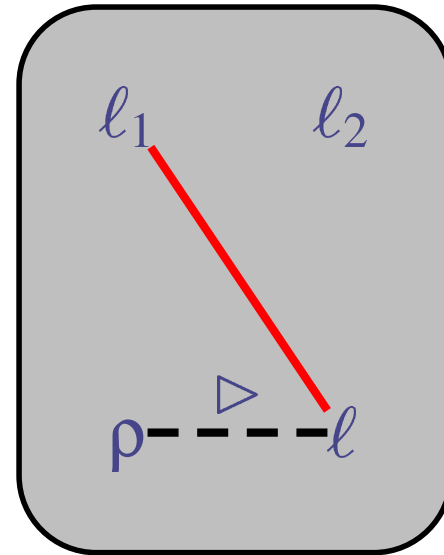
```
pthread_mutex_init(&L1);  
pthread_mutex_init(&L2);  
if(...) l = &L1;  
else l = &L2;  
pthread_mutex_lock(&l);  
x = 3;  
pthread_mutex_unlock(&l);
```



Linearity Example

```
pthread_mutex_t L1<math>l_1</math>, L2<math>l_2</math>, *l<math>l</math>;  
int x; // &x: int*<math>\rho_x</math>
```

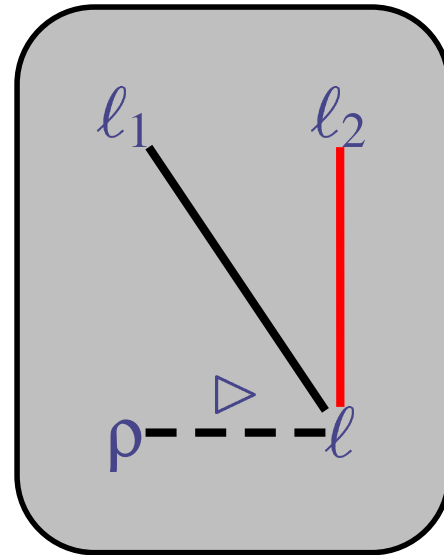
```
pthread_mutex_init(&L1);  
pthread_mutex_init(&L2);  
if(...) l = &L1;  
else l = &L2;  
pthread_mutex_lock(&l);  
x = 3;  
pthread_mutex_unlock(&l);
```



Linearity Example

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

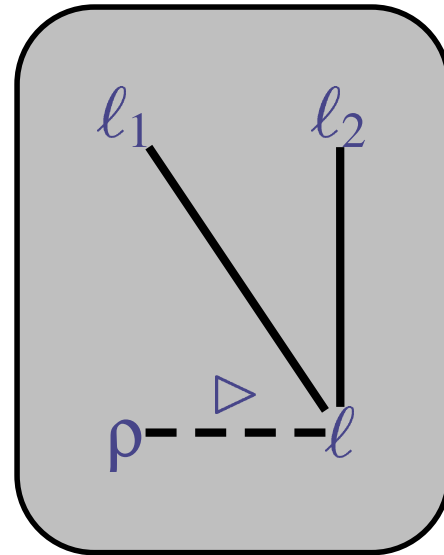
```
pthread_mutex_init(&L1);  
pthread_mutex_init(&L2);  
if(...) l = &L1;  
else l = &L2;  
pthread_mutex_lock(&l);  
x = 3;  
pthread_mutex_unlock(&l);
```



Linearity Example

```
pthread_mutex_t L1<math>l_1</math>, L2<math>l_2</math>, *l<math>l</math>;  
int x; // &x: int*<math>\rho_x</math>
```

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

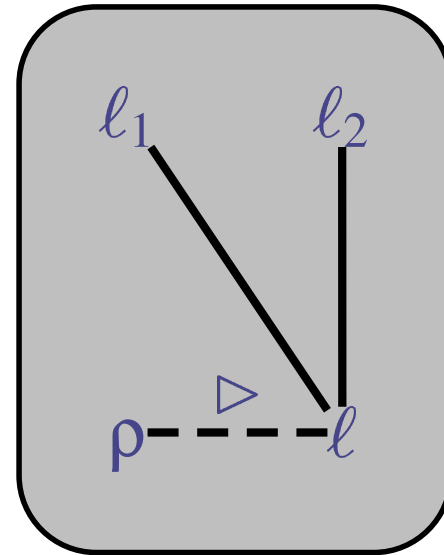


Linearity Example

```
pthread_mutex_t L1⟨ $l_1$ ⟩, L2⟨ $l_2$ ⟩, *l⟨ $l$ ⟩;  
int x; // &x: int*⟨ $\rho_x$ ⟩
```

```
pthread_mutex_init(&L1); { $l_1$ }  
pthread_mutex_init(&L2); { $l_2$ }  
if(...) l = &L1; 0  
else l = &L2; 0  
pthread_mutex_lock(&l); 0  
x = 3;  
pthread_mutex_unlock(&l); 0
```

$\{l_1\} \uplus \{l_2\}$

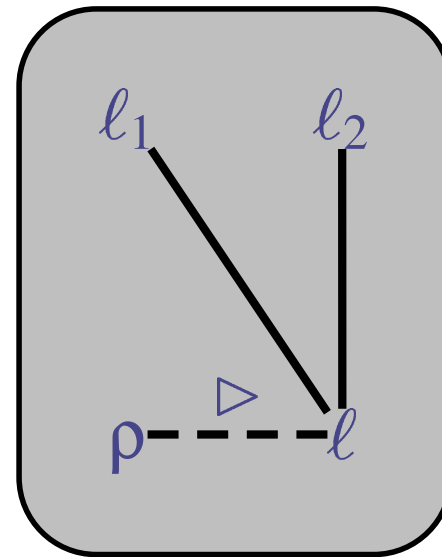


Linearity Example

```
pthread_mutex_t L1⟨ $l_1$ ⟩, L2⟨ $l_2$ ⟩, *l⟨ $l$ ⟩;  
int x; // &x: int*⟨ $\rho_x$ ⟩
```

```
pthread_mutex_init(&L1); { $l_1$ }  
pthread_mutex_init(&L2); { $l_2$ }  
if(...) l = &L1; 0  
else l = &L2; 0  
pthread_mutex_lock(&l); 0  
x = 3;  
pthread_mutex_unlock(&l); 0
```

$l_1 \neq l_2$



Soundness

- Formal system for a functional language: λ_{\triangleright}
- Proof: type safety in λ_{\triangleright} implies race freedom

- Correlation constraints have other applications:
 - Pointers correlated with allocation regions
 - Arrays correlated with integer lengths

LOCKSMITH: Implementation for C

- Apply consistent correlation inference to the full C language
- Challenges:
 - Infer the acquired set at every program point
 - Locks in data structures
 - Increase precision using `void *` inference
 - Thread locality (can be flow sensitive)
 - 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

Flow sensitive lock state

- Which locks are acquired at each program point?
- 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 call site i
 - $\psi \xrightarrow{)i} \psi'$: function returns control at call site i
 - Solve using data-flow analysis

Example: generating constraints

```
pthread_mutex_t  $\langle l_1 \rangle$  L1 = ...;
```

```
int x; // &x: int*  $\langle \rho_x \rangle$ 
```

```
void munge(pthread_mutex_t  $\langle l \rangle$  *l, int *  $\langle \rho \rangle$  p) {
```

```
    pthread_mutex_lock(l);
```

```
    *p = 3;
```

```
    pthread_mutex_unlock(l);
```

```
}
```

```
...
```

```
munge1(&L1, &x);
```

Ψ_{in}

Ψ_1

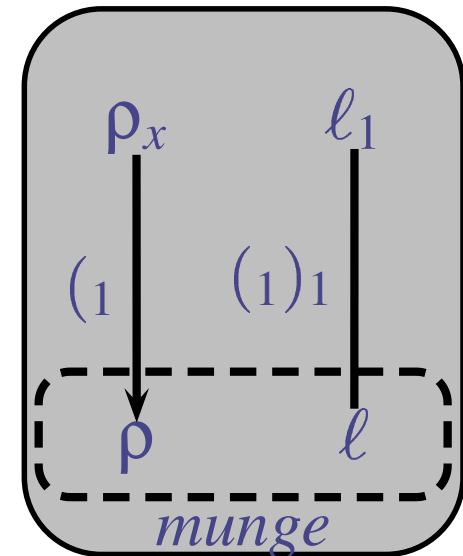
Ψ_2

Ψ_3

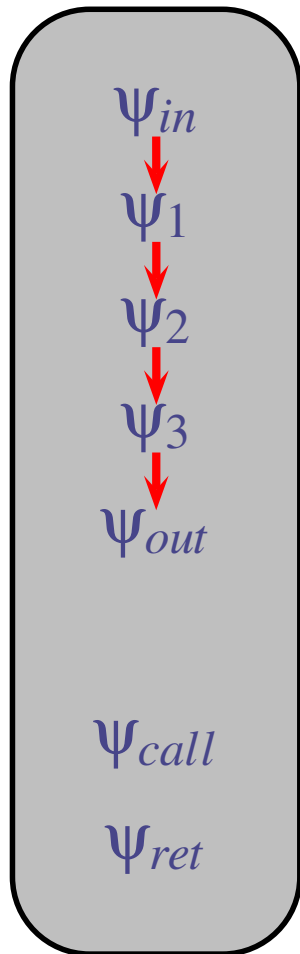
Ψ_{out}

Ψ_{call}

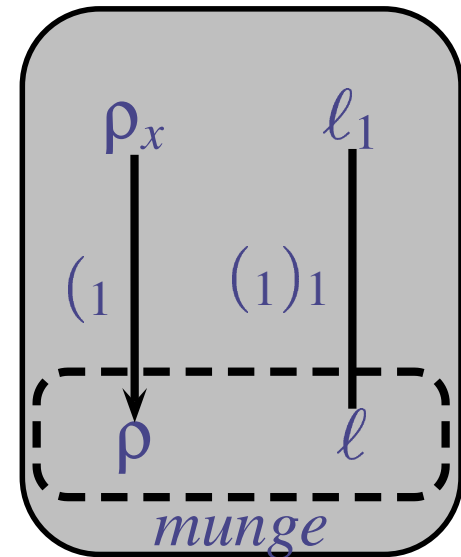
Ψ_{ret}



Example: generating constraints



```
pthread_mutex_t  $\langle l_1 \rangle$  L1 = ...;  
int x; // &x: int*  $\langle \rho_x \rangle$   
void munge(pthread_mutex_t  $\langle l \rangle$  *l, int *  $\langle \rho \rangle$  p) {  
    pthread_mutex_lock(l);  
    *p = 3;  
    pthread_mutex_unlock(l);  
}  
...  
munge1(&L1, &x);
```



Example: generating constraints

```
pthread_mutex_t  $\langle l_1 \rangle$  L1 = ...;
```

```
int x; // &x: int*  $\langle \rho_x \rangle$ 
```

```
void munge(pthread_mutex_t  $\langle l \rangle$  *l, int *  $\langle \rho \rangle$  p) {
```

```
    pthread_mutex_lock(l);
```

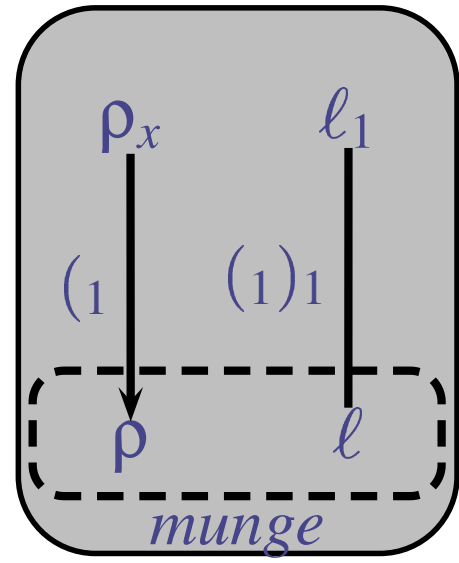
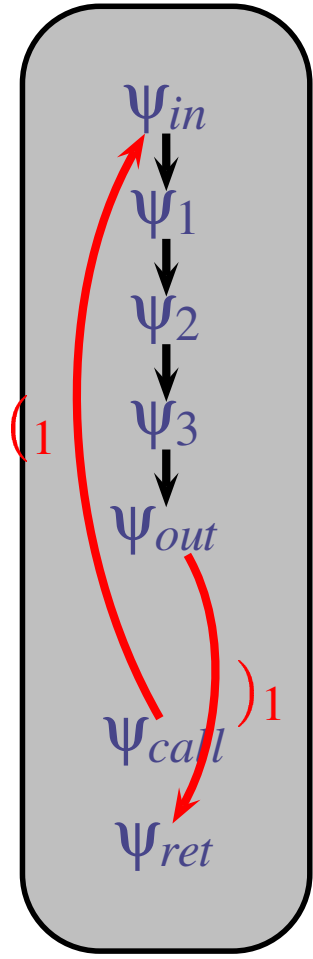
```
    *p = 3;
```

```
    pthread_mutex_unlock(l);
```

```
}
```

```
...
```

```
munge1(&L1, &x);
```



Example: generating constraints

```
pthread_mutex_t  $\langle l_1 \rangle$  L1 = ...;
```

```
int x; // &x: int*  $\langle \rho_x \rangle$ 
```

```
void munge(pthread_mutex_t  $\langle l \rangle$  *l, int *  $\langle \rho \rangle$  p) {
```

```
    pthread_mutex_lock(l);
```

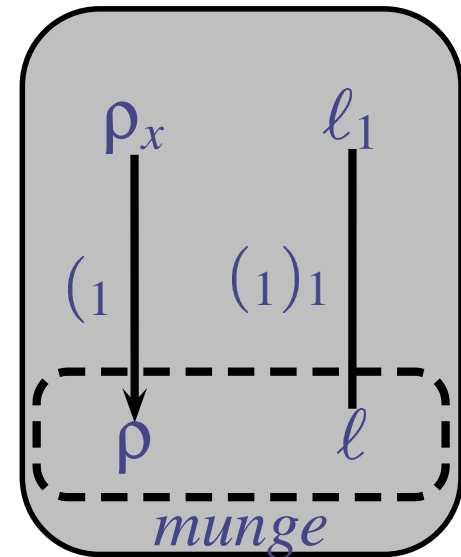
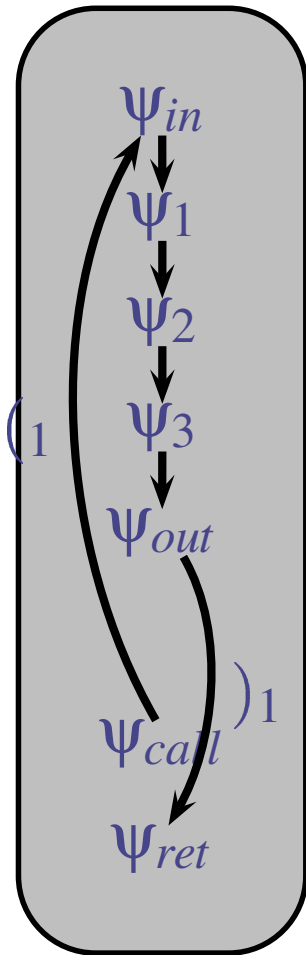
```
    *p = 3;
```

```
    pthread_mutex_unlock(l);
```

```
}
```

```
...
```

```
munge1(&L1, &x);
```



Example: generating constraints

```

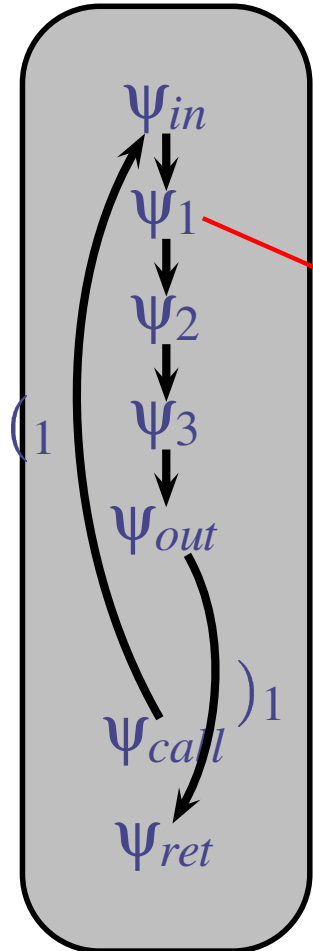
pthread_mutex_t<ℓ1> L1 = ...;
int x; // &x: int*<ρx>

void munge(pthread_mutex_t<ℓ> *l, int *<ρ> p) {
    pthread_mutex_lock(l);

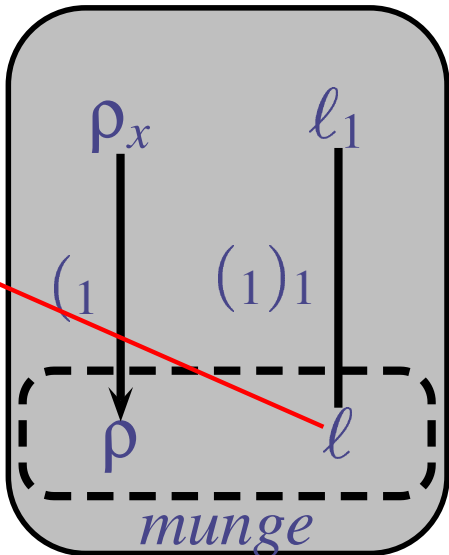
    *p = 3;
    pthread_mutex_unlock(l);
}

...
munge1(&L1, &x);

```



Acquired



Example: generating constraints

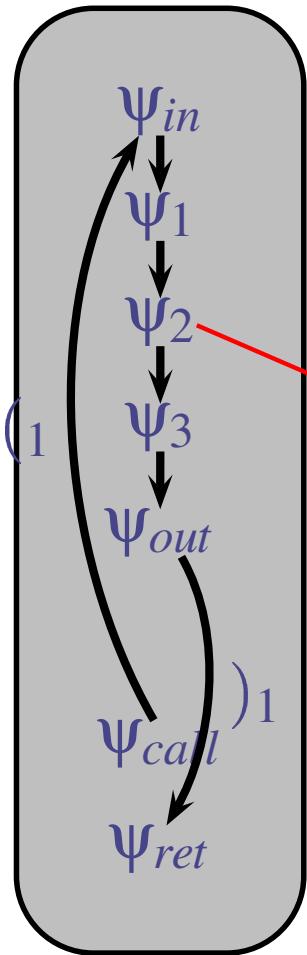
```

pthread_mutex_t  $\langle l_1 \rangle$  L1 = ...;
int x; // &x: int*  $\langle \rho_x \rangle$ 

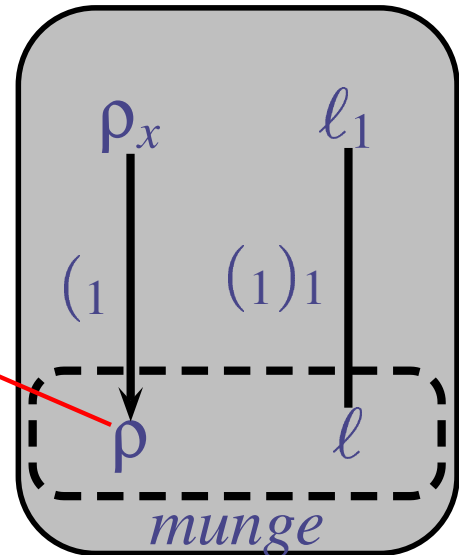
void munge(pthread_mutex_t  $\langle l \rangle$  *l, int *  $\langle \rho \rangle$  p) {
    pthread_mutex_lock(l);
    *p = 3;
    pthread_mutex_unlock(l);
}

...
munge1(&L1, &x);

```



Dereferenced



Example: generating constraints

```
pthread_mutex_t  $\langle l_1 \rangle$  L1 = ...;
```

```
int x; // &x: int*  $\langle \rho_x \rangle$ 
```

```
void munge(pthread_mutex_t  $\langle l \rangle$  *l, int *  $\langle \rho \rangle$  p) {
```

```
    pthread_mutex_lock(l);
```

```
    *p = 3;
```

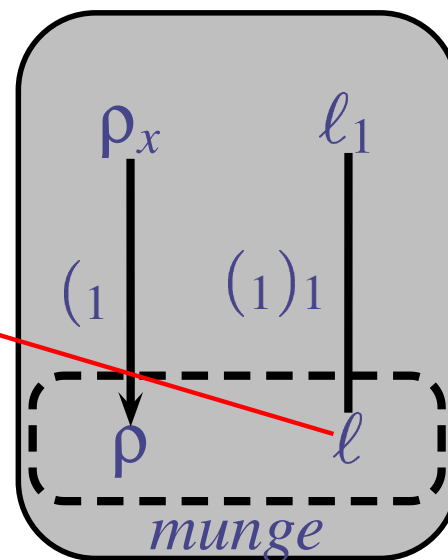
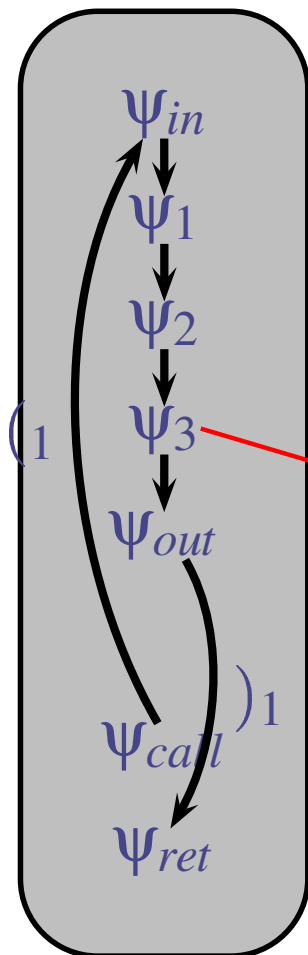
```
    pthread_mutex_unlock(l);
```

```
}
```

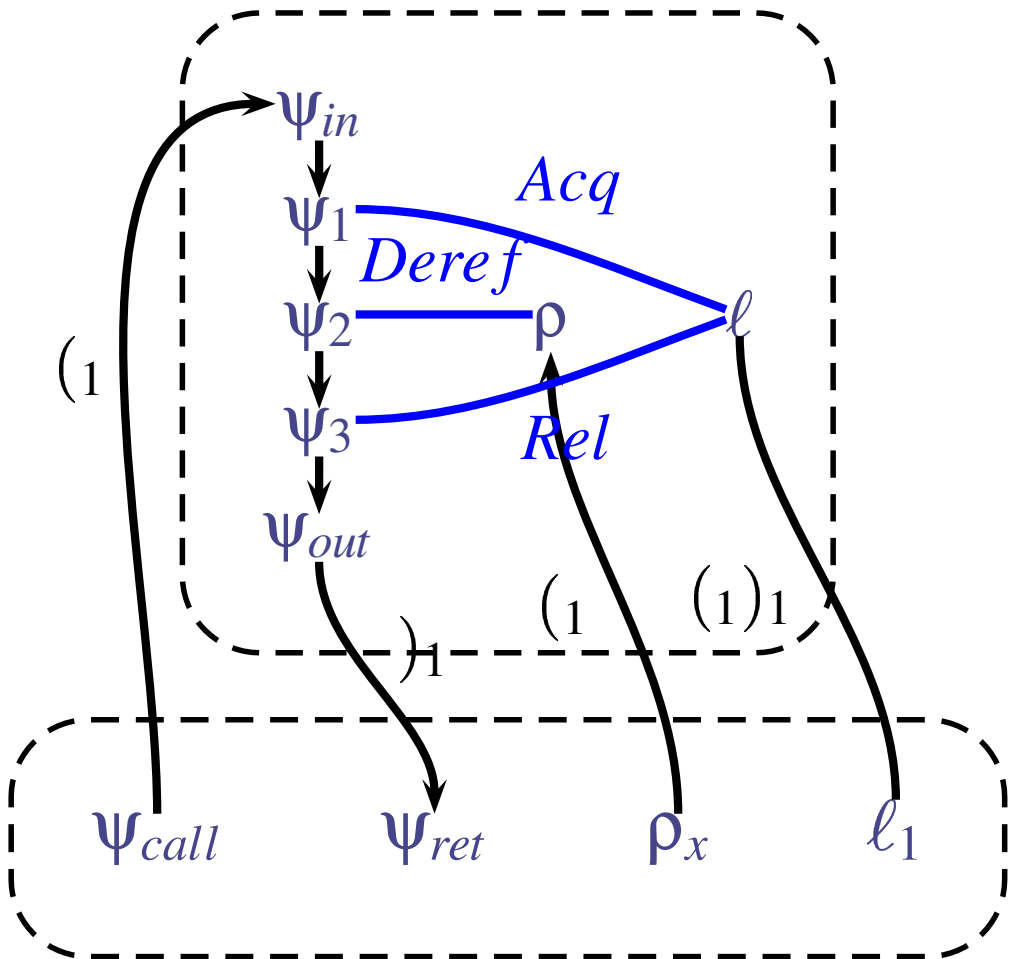
```
...
```

```
munge1(&L1, &x);
```

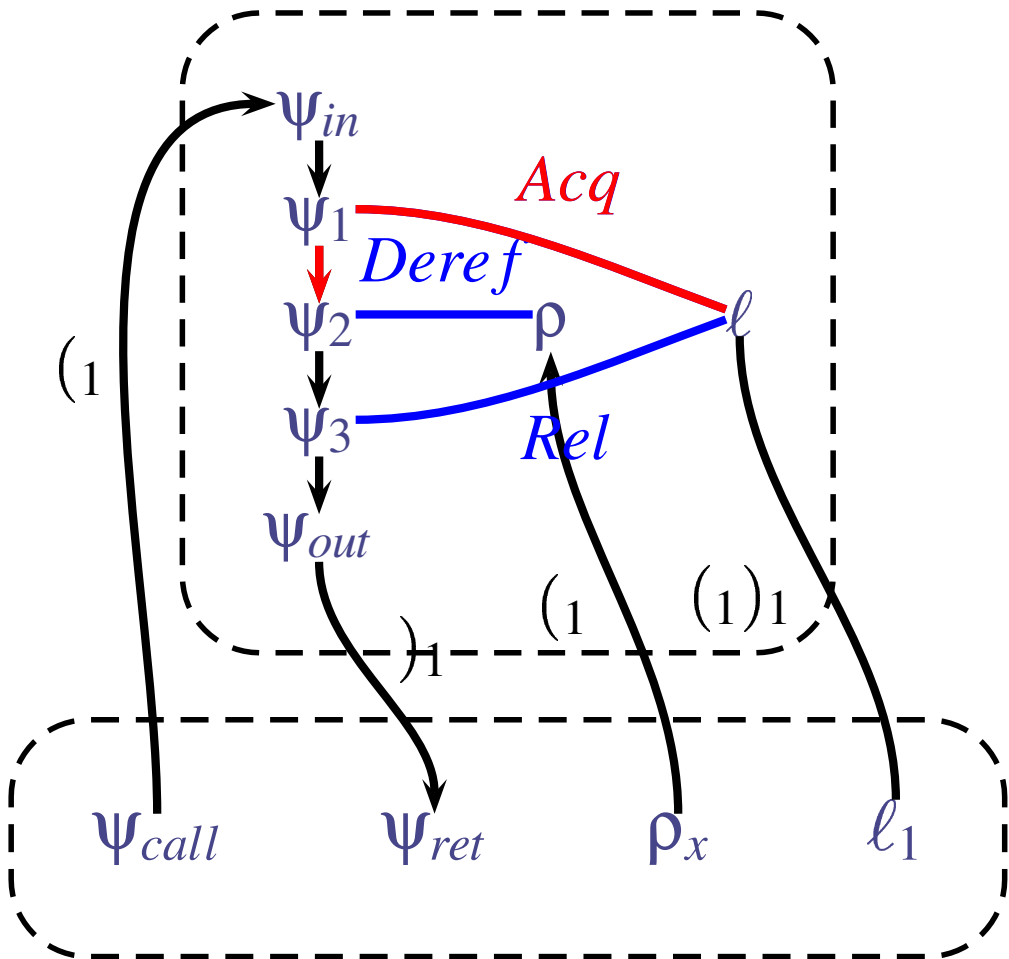
Released



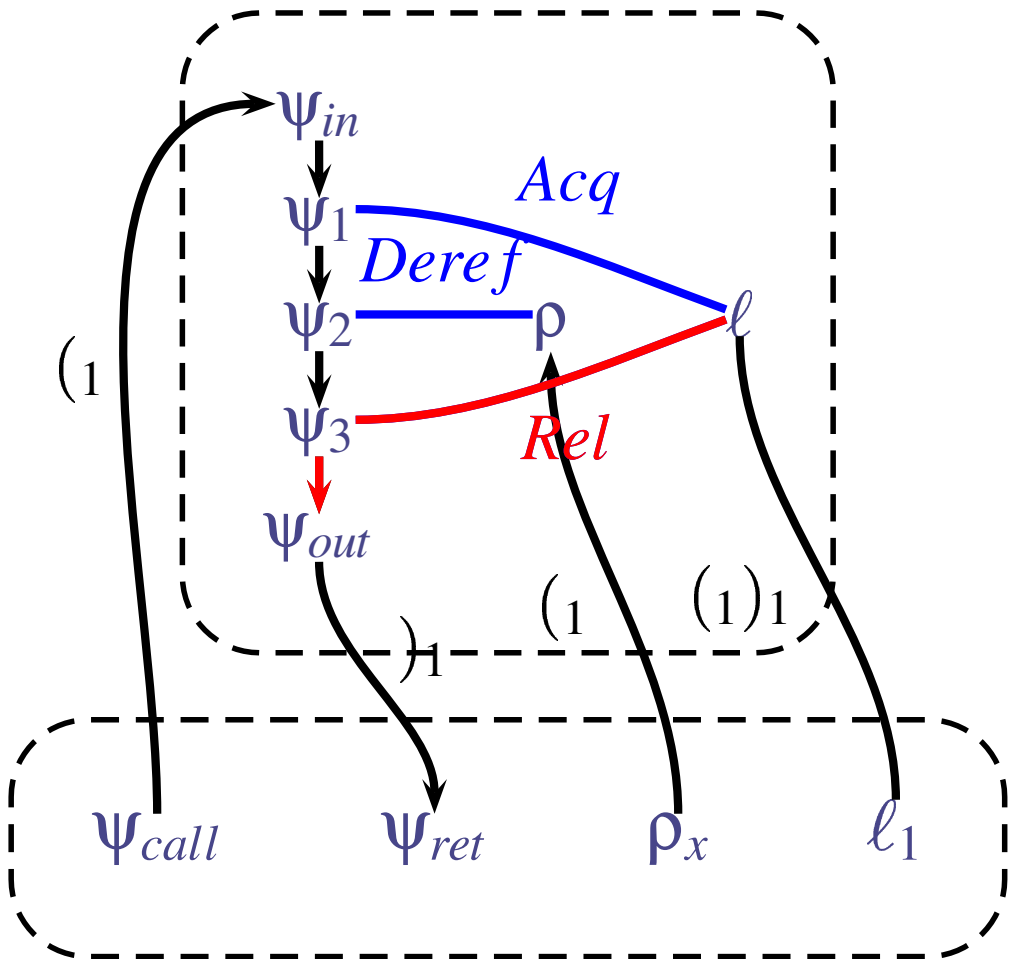
Example: solving constraints



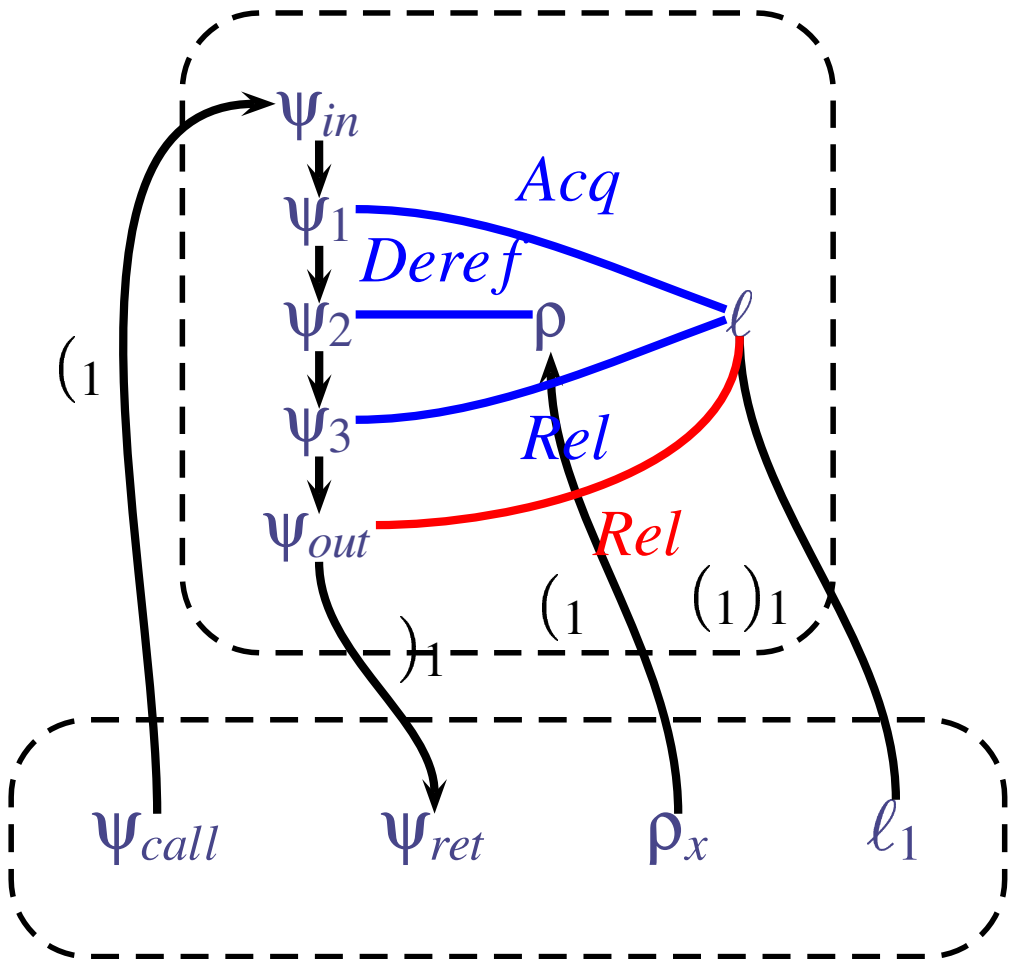
Example: solving constraints



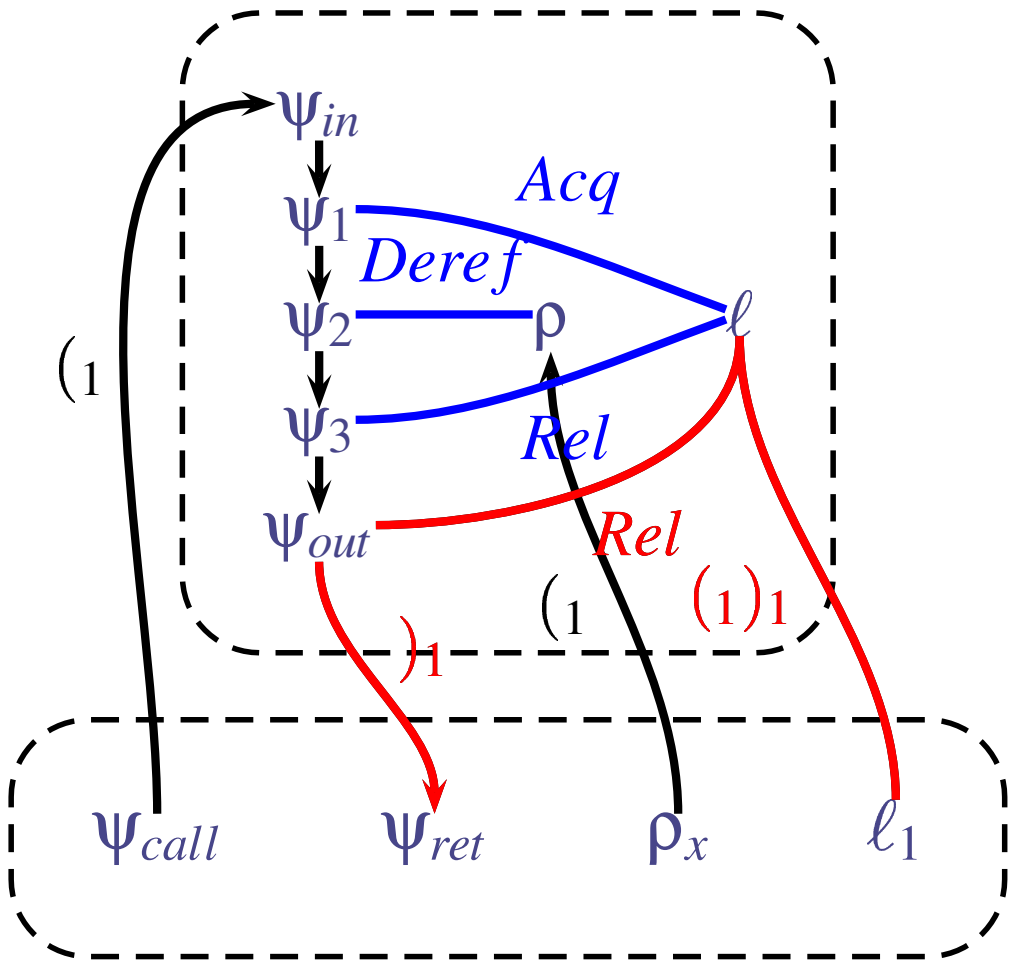
Example: solving constraints



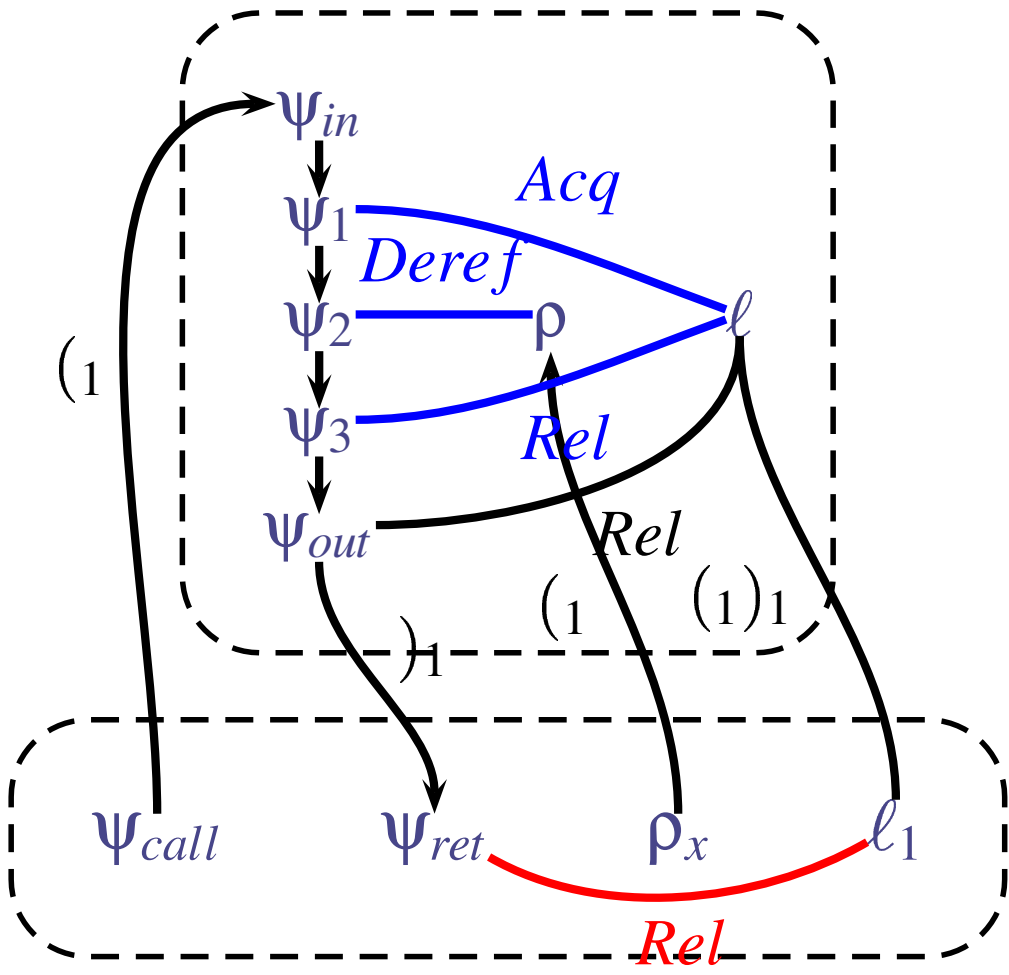
Example: solving constraints



Example: solving constraints



Example: solving constraints



Existential Context Sensitivity

- Often, locks exist in data structures:

```
struct foo {  
    pthread_mutex_t<ℓ> lock;  
    int*<ρ> data;  
    struct foo* next;  
};
```

- Alias analysis conflates nodes in data structures
- Can recover precise correlation within individual elements
- Programmer writes existential annotations

Existential Context Sensitivity

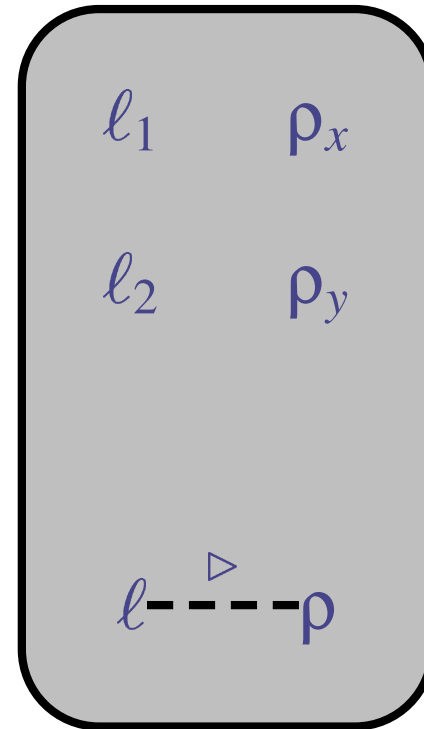
- Often, locks exist in data structures:

```
struct foo {  $\exists p, l. p \triangleright l$   
    pthread_mutex_t  $\langle l \rangle$  lock;  
    int*  $\langle p \rangle$  data;  
    struct foo* next;  
};
```

- Alias analysis conflates nodes in data structures
- Can recover precise correlation within individual elements
- Programmer writes existential annotations

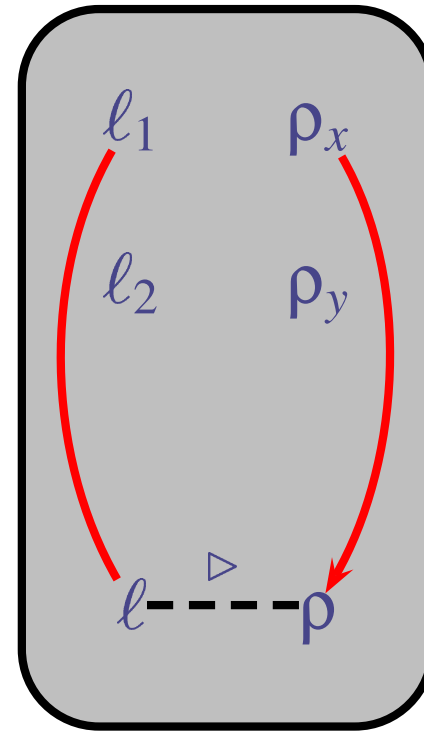
Existential Context Sensitivity

```
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);
```



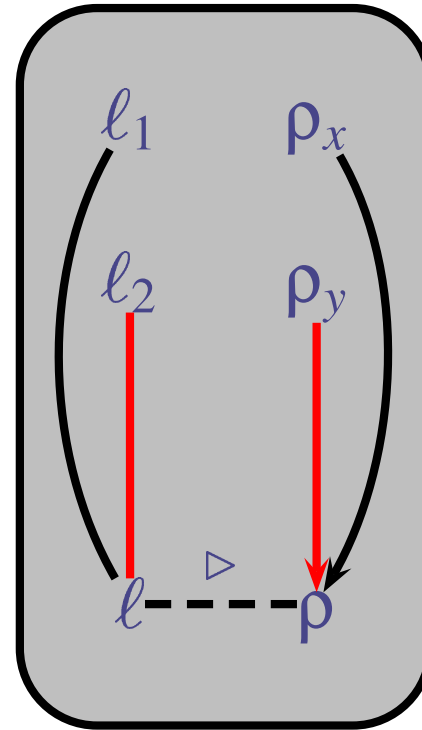
Existential Context Sensitivity

```
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);
```



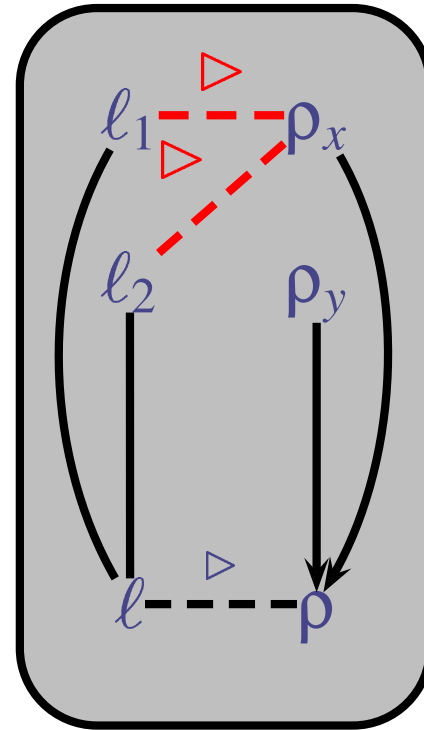
Existential Context Sensitivity

```
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);
```



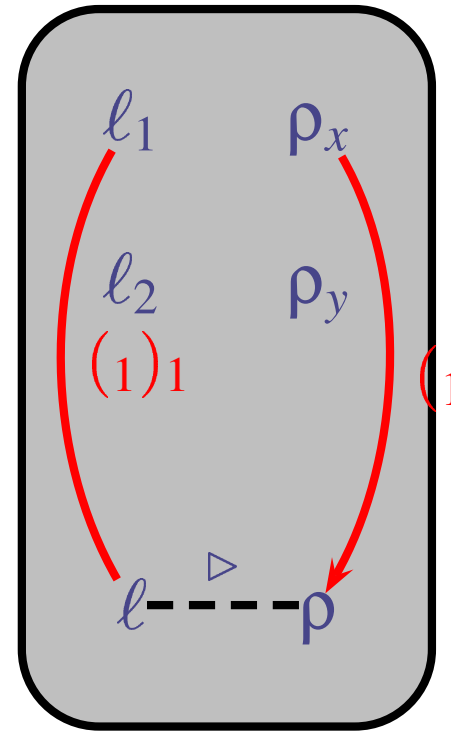
Existential Context Sensitivity

```
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);
```



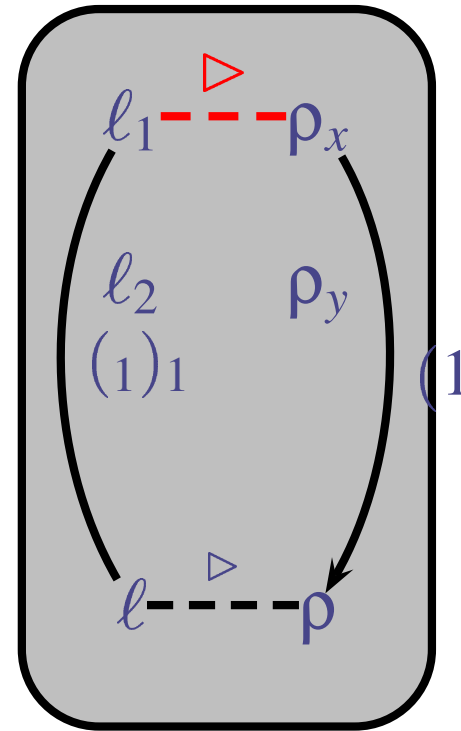
Existential Context Sensitivity

```
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);
```



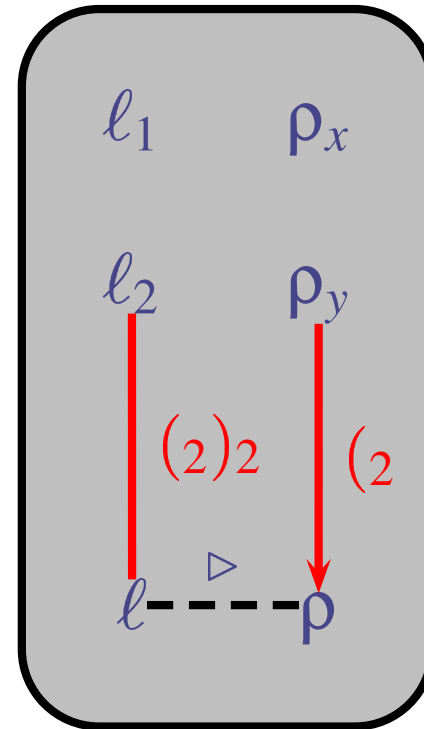
Existential Context Sensitivity

```
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);
```



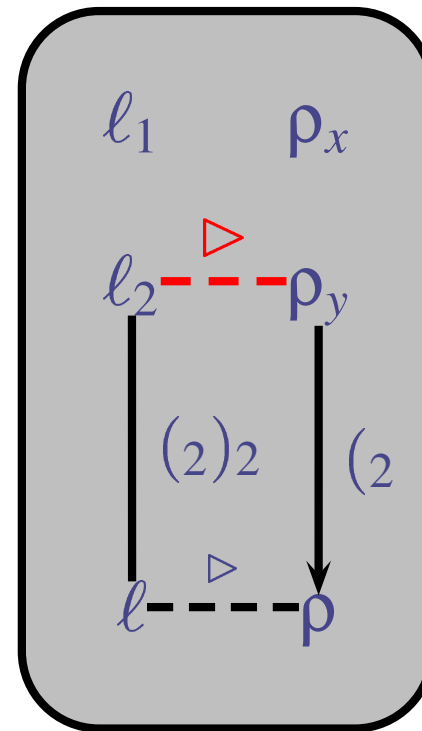
Existential Context Sensitivity

```
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);
```



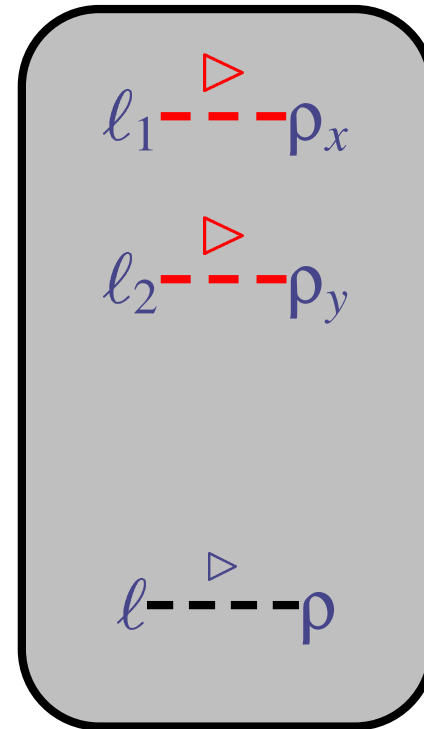
Existential Context Sensitivity

```
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);
```



void *

- Annotate each occurrence of a `void *` pointer 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

- Standalone C programs
- Linux device drivers
- Experiments on a dual core Xeon processor, at 2.8MHz, 3.5GB RAM

Standalone programs

Program	Size (KLOC)	Time	Warn.	Unguarded	Races
aget	1.6	0.8s	15	15	15
ctrace	1.8	0.9s	8	8	2
pfscan	1.7	0.7s	5	0	0
engine	1.5	1.2s	7	0	0
smtprc	6.1	6.0s	46	1	1
knot	1.7	1.5s	12	8	8

Linux Drivers

Driver	Size (KLOC)	Time	Warn.	Unguarded	Races
plip	19.1	24.9s	11	2	1
eql	16.5	3.2s	3	0	0
3c501	17.4	240.1s	24	2	2
sundance	19.9	98.2s	3	1	0
sis900	20.4	61.0s*	8	2	1
slip	22.7	16.5s*	19	1	0
hp100	20.3	31.8s*	23	2	0

(*) Run without lock linearity analysis

Conclusions

Contribution:

- Discover races automatically by inferring consistent correlation
- Formalized correlation inference system with universal and existential context sensitivity
- Proof of soundness
- LOCKSMITH: Implementation for C
 - Requires no annotations (minimal annotations when using existential context sensitivity)
 - Found races in existing programs and Linux drivers

LOCKSMITH is available

- Download LOCKSMITH at <http://www.cs.umd.edu/~polyvios/locksmith>
- Analyses are modular, easy to reuse