
Sound, precise and efficient static race detection for multi-threaded programs

Polyvios Pratikakis

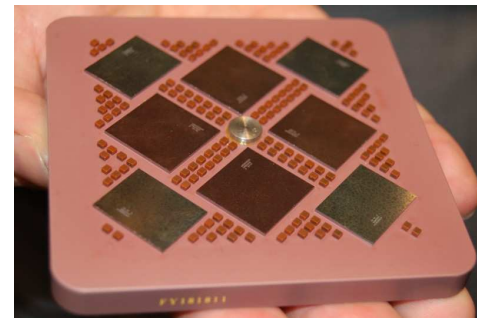
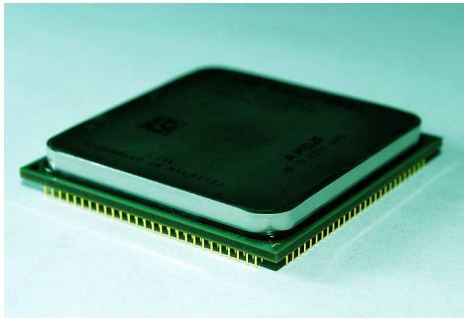
Michael Hicks Jeffrey Foster

University of Maryland, College Park

Moore's law

- 1965:
*“The complexity for minimum component costs has increased at a rate of roughly a **factor of two per year** . . . Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years.”*
- 1975: New projection: doubling every 2 years
- 2006: Doubling every 3 years, expected to slow down more

Multi-core processors



- Multi-core CPUs are increasingly available
- ClearSpeed 2004: 96 math co-processor cores on a chip
- Intel 2007: 80 simple cores on a chip
- Quad-core CPUs are everywhere, octa-core CPUs available in server market
- Demand for parallel software increasing

Current model considered harmful

- Threads and shared memory
 - Dominant model for parallel programming
- Difficult to program:
 - Hard to reason about all possible orderings
 - Subtle interactions of threads through shared memory
 - Easy to forget synchronization, introduce subtle bugs
 - Unintuitive model, implicit thread interactions and orderings

Concurrency errors

- Two kinds of synchronization errors:
 - Too much synchronization: Deadlocks, loss of parallelism
 - Too little synchronization: Data races, atomicity violations
- Parallelism introduces non-determinism
 - Errors might not be triggered by traditional testing
 - Difficult to debug and repair

Big problem: Data races

- Race: two threads access a memory location without synchronization and at least one is a write
- Well known problems caused by races:
 - August 14th 2003, Northeastern Blackout
 - 1985-1987, Therac-25 medical accelerator
- Programs with races are difficult to understand

Contributions

- LOCKSMITH: a tool that automatically detects data races in C programs
- Context-sensitive correlation analysis, formalized and proved sound
- Existential context sensitive label-flow analysis, formalized and proved sound
- Contextual effects proof of soundness, mechanized in Coq

Preventing races with locks

The most common technique:

- Shared locations ρ , locks ℓ
- Correlation $\rho \triangleright \ell$:
Lock ℓ is correlated with pointer ρ , if ℓ is held while ρ is accessed
- *Consistent correlation*:
Location ρ is *always* correlated with lock ℓ
- Ensure that every shared location ρ is *consistently correlated* with a lock ℓ

A static analysis against races

- Goal: Develop a tool for determining whether a C program is race-free
- Design criteria:
 - Be sound: complain if there is a race
 - Handle locking idioms commonly-used in C programs
 - Don't require many annotations
 - In particular, do not require the program to describe which locations are guarded by what locks
 - Scale to large programs

A static analysis against races

- Unsolvable problem in general:
 - Rices Theorem: No computer program can precisely determine anything interesting about arbitrary source code
- Static analysis: approximation solution
 - Can be conservative: When in doubt assume the worst

Static correlation inference

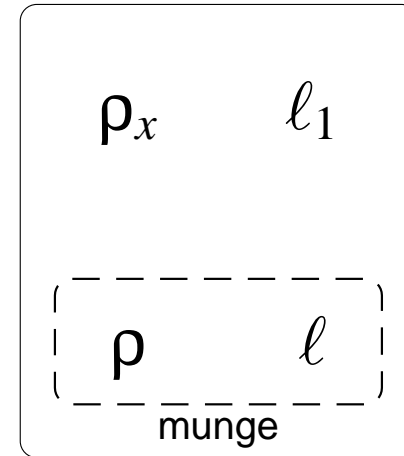
- Approximate run-time values with static ρ and ℓ
- Infer correlations $\rho \triangleright \ell$
 - Every access creates a $\rho \triangleright \ell$ constraint
 - Compute aliasing information and use it to propagate correlations
 - Infer all other $\rho \triangleright \ell$ relations by solving these constraints
- Ensure consistent correlation
 - Verify consistent correlation for every shared ρ , or report a contradiction (race)

Example: correlation inference

```
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);
```

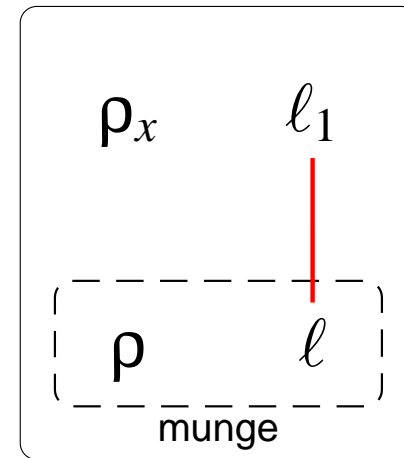
Example: correlation inference

```
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);
```



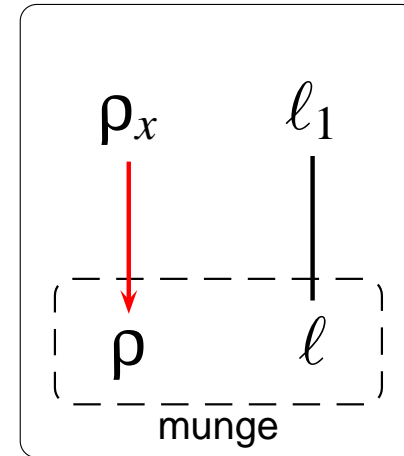
Example: correlation inference

```
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);
}
...
munge(&L1, &x);
```



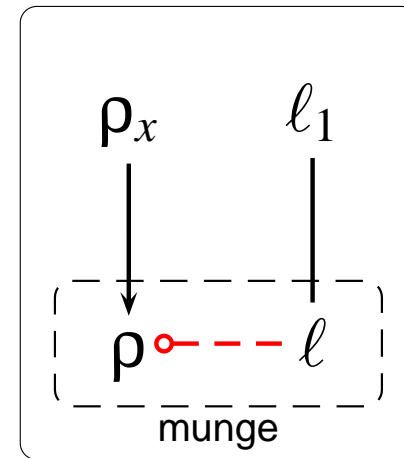
Example: correlation inference

```
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);
```



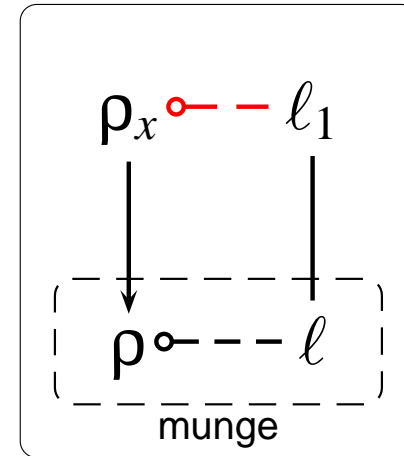
Example: correlation inference

```
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);
```



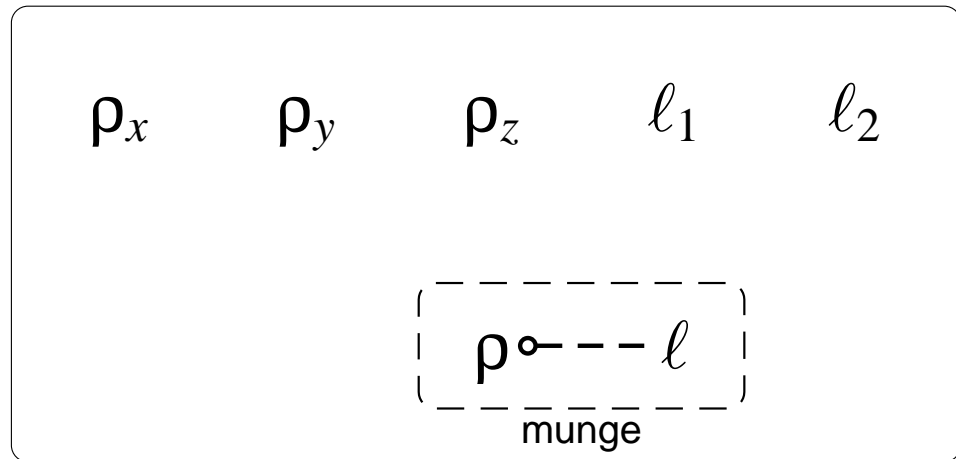
Example: correlation inference

```
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);
```



Example: 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);
```



Example: 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);
```

The diagram shows a box containing variables ρ_x , ρ_y , ρ_z , ℓ_1 , and ℓ_2 . A dashed box labeled "munge" contains variables ρ and ℓ . Red arrows indicate the mapping: one arrow points from the parameter $\langle \rho \rangle$ in the function signature to ρ_x ; another arrow points from the parameter $\langle \ell \rangle$ to ℓ_1 ; and a third arrow points from the parameter $\langle \rho \rangle$ to the ρ variable inside the "munge" box.

Example: context sensitivity

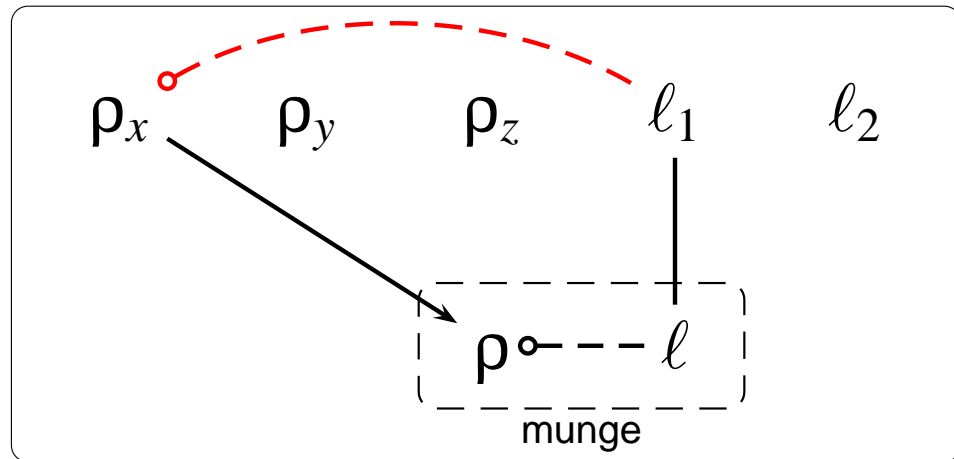
```
pthread_mutex_t<ℓ1> L1 = ..., <ℓ2> L2 = ...;
int x, y, z; // <ρx>, <ρy>, <ρz>
void munge(pthread_mutex_t<ℓ> *l, int *<ρ> p) {
    pthread_mutex_lock(l);
    *p = 3;
    pthread_mutex_unlock(l);
}
```

...

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

```
munge (&L2, &y);
```

```
munge (&L2, &z);
```

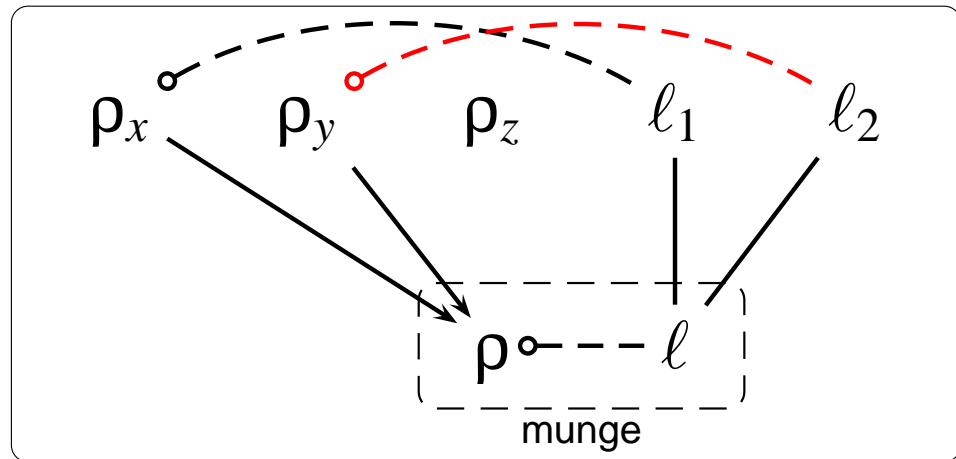


Example: context sensitivity

```
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);
```

Example: 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);
```



Example: context sensitivity

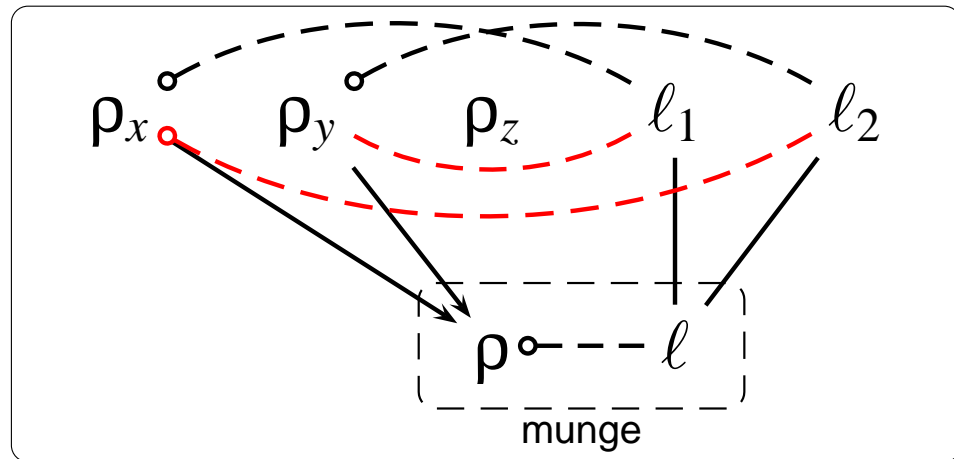
```
pthread_mutex_t<ℓ1> L1 = ..., <ℓ2> L2 = ...;
int x, y, z; // <ρx>, <ρy>, <ρz>
void munge(pthread_mutex_t<ℓ> *l, int *<ρ> p) {
    pthread_mutex_lock(l);
    *p = 3;
    pthread_mutex_unlock(l);
}
```

...

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

```
munge (&L2, &y);
```

```
munge (&L2, &z);
```



Example: context sensitivity

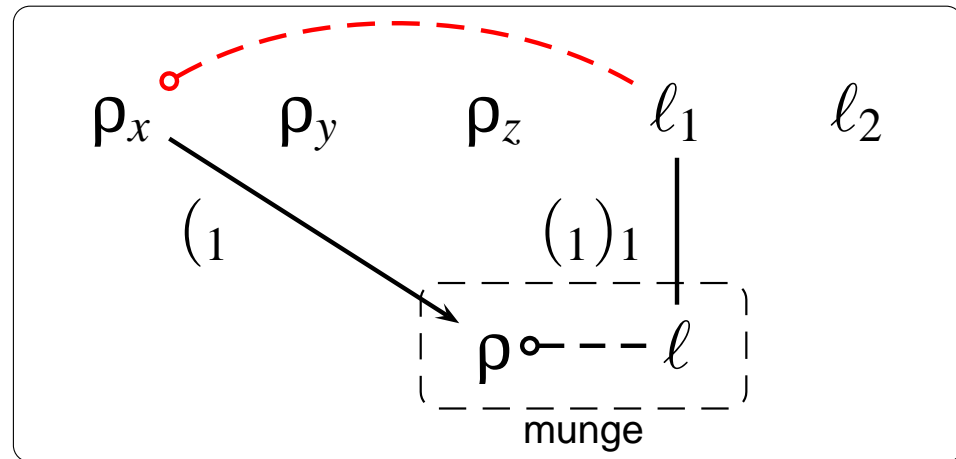
```
pthread_mutex_t<ℓ1> L1 = ..., <ℓ2> L2 = ...;
int x, y, z; // <ρx>, <ρy>, <ρz>
void munge(pthread_mutex_t<ℓ> *l, int *<ρ> p) {
    pthread_mutex_lock(l);
    *p = 3;           (1)1
    pthread_mutex_unlock(l); (1)
}
...
munge1 (&L1, &x);
munge2 (&L2, &y);
munge3 (&L2, &z);
```

The diagram illustrates context sensitivity. A box contains variables ρ_x , ρ_y , ρ_z , ℓ_1 , and ℓ_2 . A dashed box labeled "munge" contains ρ and ℓ . Red arrows show the mapping: ρ_x to the ρ parameter in the first munge call, and ℓ_1 to the ℓ parameter in the first munge call. The first munge call is annotated with $(1)_1$, and the second with (1) .

Example: 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);
}
```

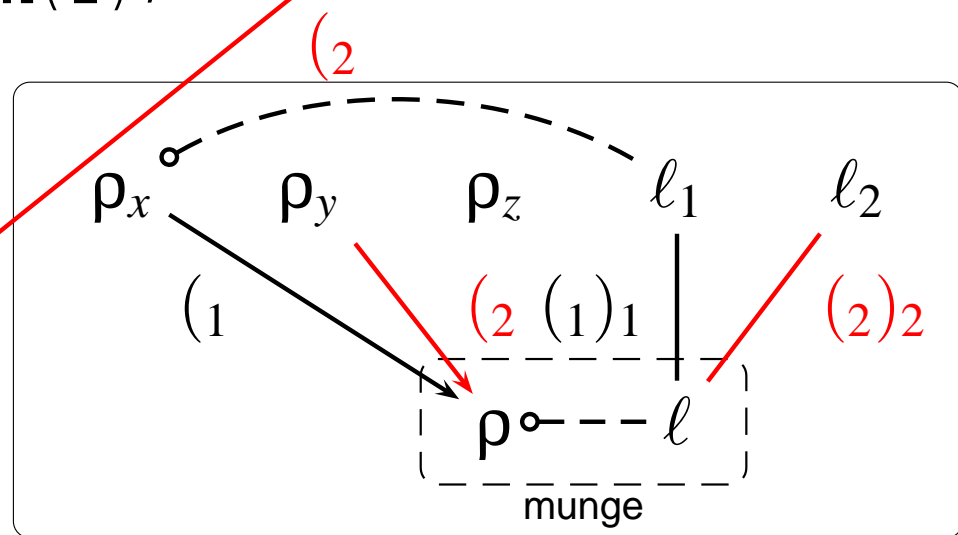
```
...
munge1 (&L1, &x);
munge2 (&L2, &y);
munge3 (&L2, &z);
```



Example: 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);
```



Example: context sensitivity

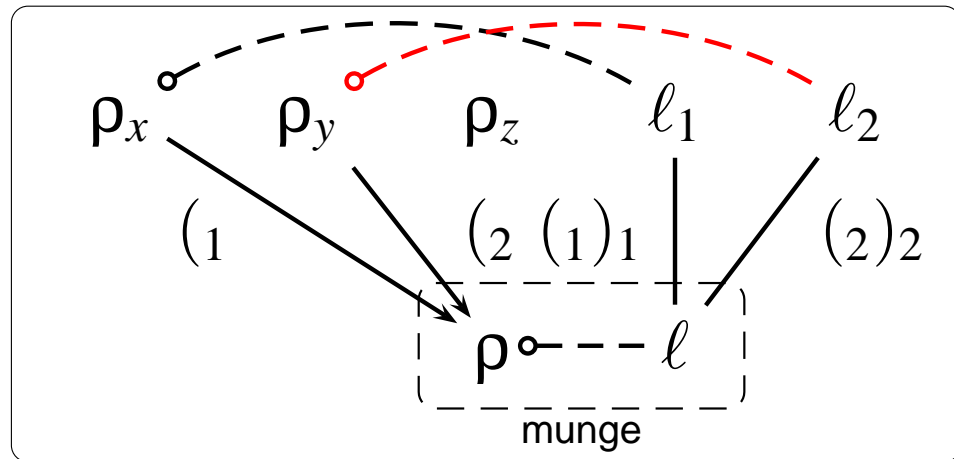
```
pthread_mutex_t<ℓ1> L1 = ..., <ℓ2> L2 = ...;
int x, y, z; // <ρx>, <ρy>, <ρz>
void munge(pthread_mutex_t<ℓ> *l, int *<ρ> p) {
    pthread_mutex_lock(l);
    *p = 3;
    pthread_mutex_unlock(l);
}
```

...

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

```
munge2 (&L2, &y);
```

```
munge3 (&L2, &z);
```



Example: context sensitivity

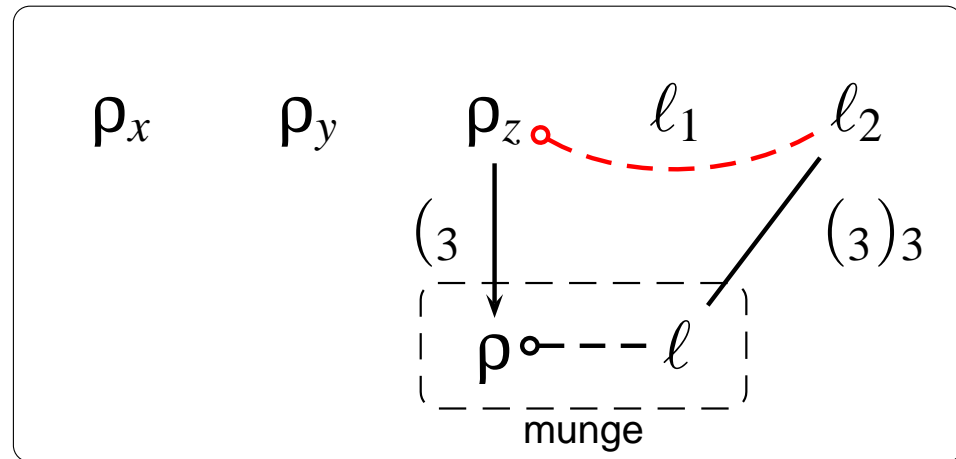
```
pthread_mutex_t<ℓ1> L1 = ..., <ℓ2> L2 = ...;
int x, y, z; // <ρx>, <ρy>, <ρz>
void munge(pthread_mutex_t<ℓ> *l, int *<ρ> p) {
    pthread_mutex_lock(l);
    *p = 3;
    pthread_mutex_unlock(l);
}
...
munge1 (&L1, &x);
munge2 (&L2, &y);
munge3 (&L2, &z);
```

The diagram illustrates context sensitivity in a program. It shows a call stack frame for the function `munge`. The parameters are ρ_x , ρ_y , ρ_z , l_1 , and l_2 . A dashed box labeled `munge` contains ρ and l . Red arrows point from the function signature in the code to the corresponding parameters in the diagram. A red arrow points from ρ_z in the diagram to ρ in the dashed box. Red annotations $(3)_3$ are placed near the function signature and the ρ_z parameter.

Example: 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);
```



Example: context sensitivity

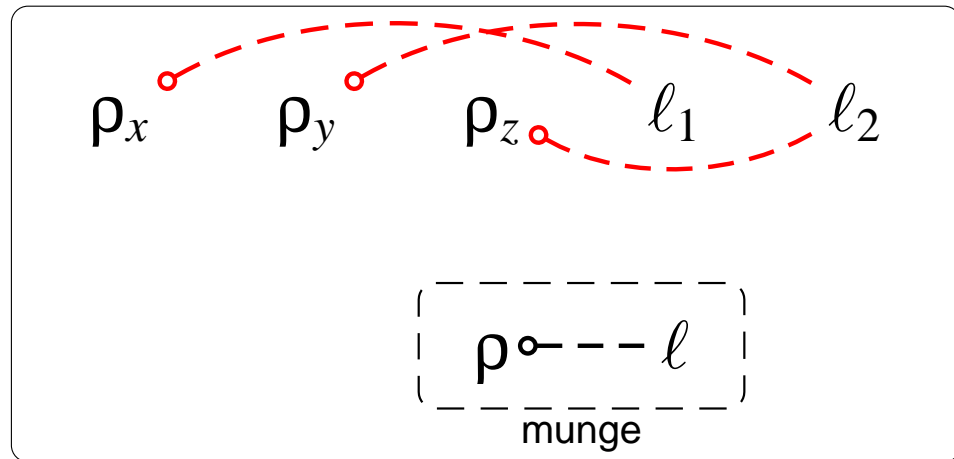
```
pthread_mutex_t<ℓ1> L1 = ..., <ℓ2> L2 = ...;
int x, y, z; // <ρx>, <ρy>, <ρz>
void munge(pthread_mutex_t<ℓ> *l, int *<ρ> p) {
    pthread_mutex_lock(l);
    *p = 3;
    pthread_mutex_unlock(l);
}
```

...

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

```
munge2 (&L2, &y);
```

```
munge3 (&L2, &z);
```



Contribution: correlation inference

- Context sensitive correlation inference
 - Formalized as type-based constraint-based analysis
 - Uses universal polymorphism for context sensitivity
 - Encoded as CFL reachability, solvable in $O(n^3)$
- Proof of soundness
 - Formalized context-copying system and proved equivalence to CFLR encoding
 - Type safety implies race freedom

Analyzing Data Structures

- Motivation: inference of “guarded by” relation between elements of a struct:

```
struct list {  
    lock_t lock;  
    int* data;  
    struct list *next;  
}
```

- Within *each* element, lock protects *data

The problem with data structures

- Actual data structure

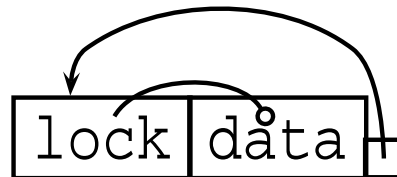


The problem with data structures

- Actual data structure



- Summarized by the analysis

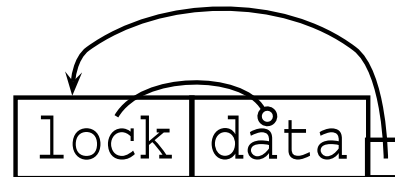


The problem with data structures

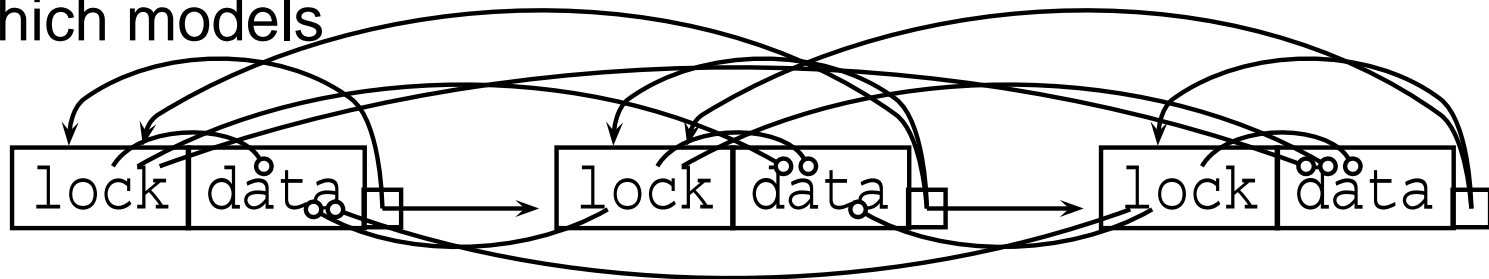
- Actual data structure



- Summarized by the analysis



- Which models

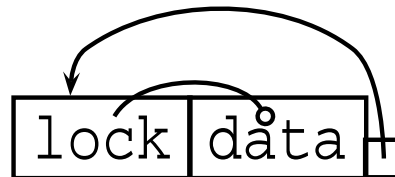


The problem with data structures

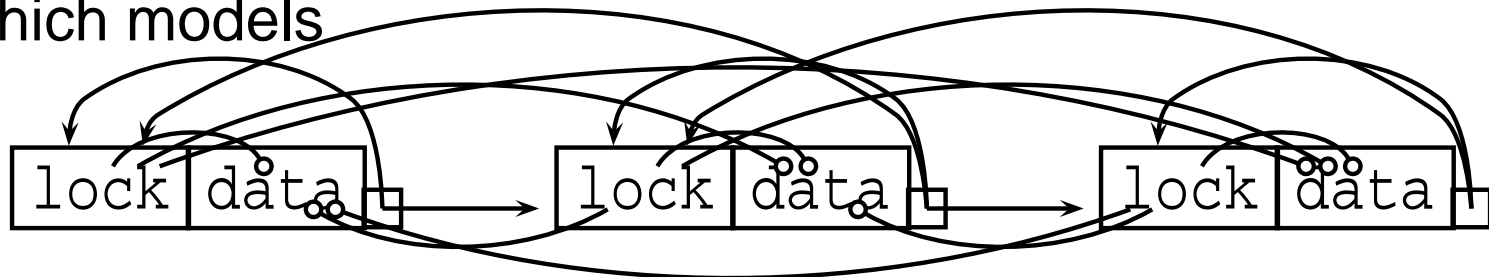
- Actual data structure



- Summarized by the analysis



- Which models



...a “blob”

Contribution: existential label flow

- Label flow analysis with support for flow within data structure elements
 - Formalized as type-based label flow analysis
 - Flow within data structure elements is existentially quantified
- Proof of soundness
 - Type-based formulation allows us to use type-system proof techniques
 - Type-soundness implies sound analysis
- Encoded as a CFL reachability problem
 - Solvable in $O(n^3)$

Sharing analysis

Inefficiency:

- Each thread accesses many memory locations
- Only a few are shared between threads

Solution:

- Find shared memory locations
- Ignore thread-local memory accesses

Find shared locations

- Find *contextual effects* (Iulian Neamtiu)
 - $[\alpha; \varepsilon; \omega]; \Gamma \vdash e : \tau$
 - α : prior effect
 - ε : normal effect
 - ω : future effect
 - Formalization and proofs in Coq
- Sharing with contextual effects
 - Locations used in child thread (ε_c)
 - Locations accessed in the parent after fork (ω_p)
 - Shared locations $\varepsilon_c \cap \omega_p$

Example: shared locations

```
int x, y;
main() {
    x = 1;
    pthread_create(&thread1);
    y = 2;
}
thread1() {
    x = 42;
    y = 43;
}
```

Example: shared locations

ω_1

ω_2

ω_3

ω_4

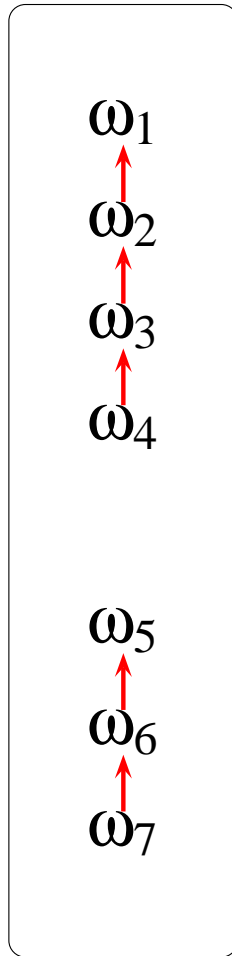
ω_5

ω_6

ω_7

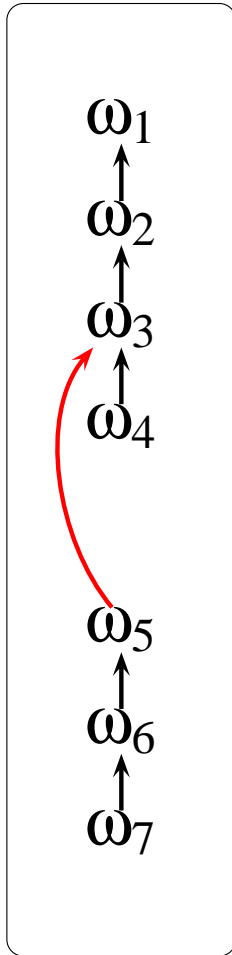
```
int x, y;
main() {
    x = 1;
    pthread_create(&thread1);
    y = 2;
}
thread1() {
    x = 42;
    y = 43;
}
```

Example: shared locations



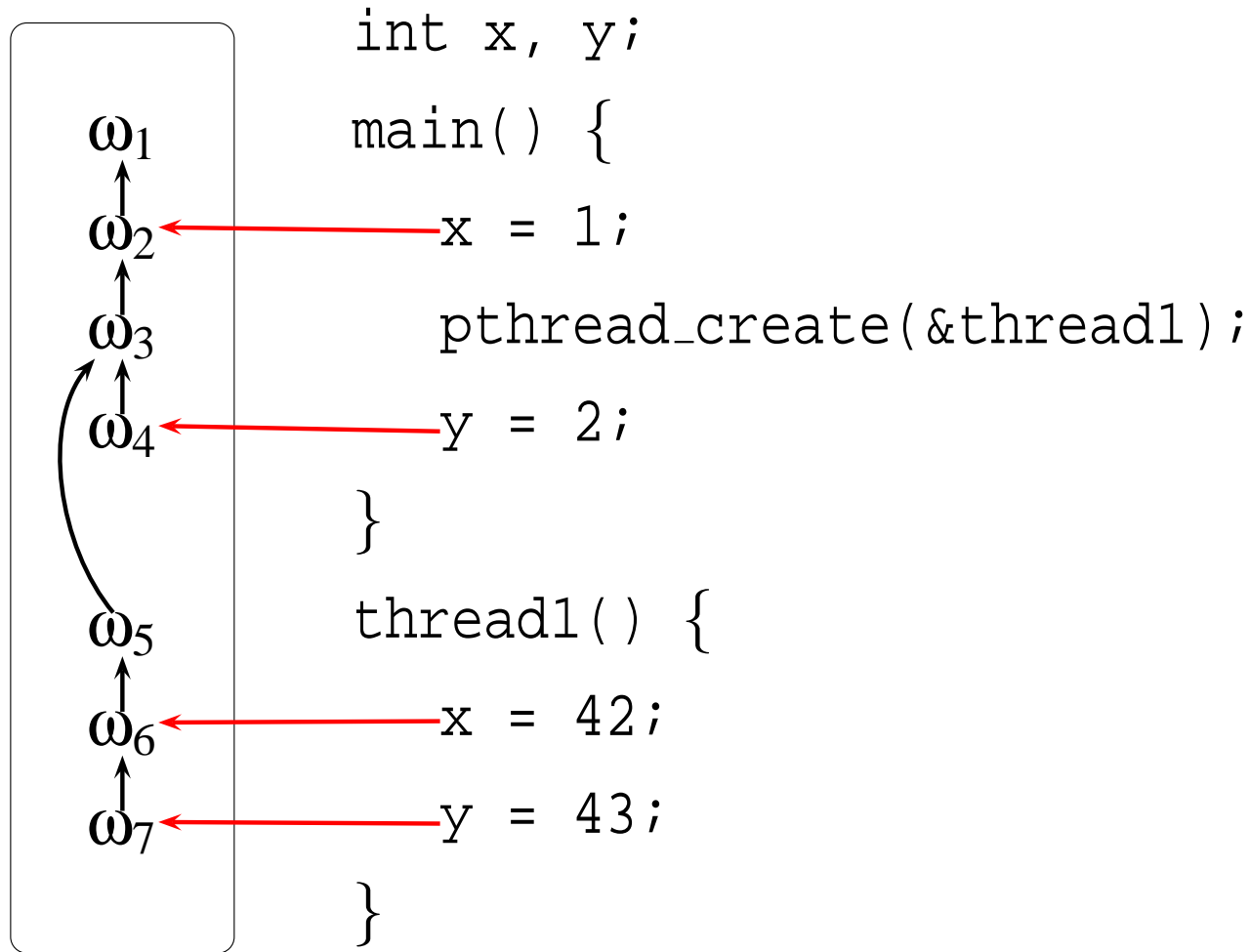
```
int x, y;
main() {
    x = 1;
    pthread_create(&thread1);
    y = 2;
}
thread1() {
    x = 42;
    y = 43;
}
```

Example: shared locations

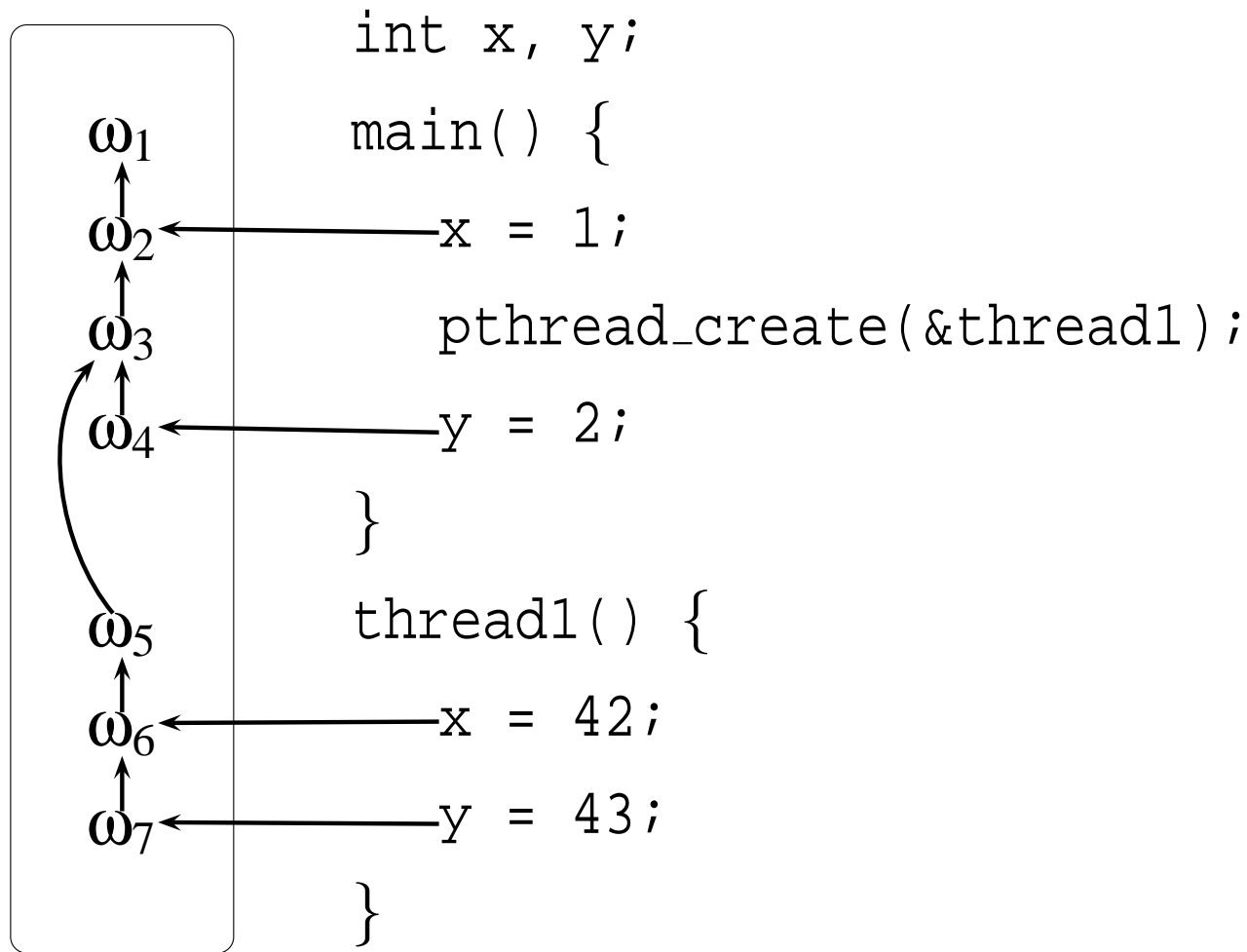


```
int x, y;
main() {
    x = 1;
    pthread_create(&thread1);
    y = 2;
}
thread1() {
    x = 42;
    y = 43;
}
```

Example: shared locations



Example: shared locations



$$\text{Shared} = \omega_4 \cap \omega_5 = \{y\} \cap \{x, y\} = \{y\}$$

Contribution: cont. eff. soundness

- Continuation effects for shared locations
- Generalized to contextual effects
 - Formally defined contextual effects using operational semantics
- Proof of soundness
 - Stated and proved that type-soundness implies soundness of inferred effects
 - Encoded and verified proof in the Coq proof assistant

LOCKSMITH: Implementation for C

- Apply consistent correlation inference to the full C language
- Some challenges:
 - Infer the acquired locks at every program point
 - Lock linearity
 - Increase precision using `void*` inference
 - More precise sharing analysis with thread-locality
 - Reduce memory footprint with lazy `struct` field expansion

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 ϕ
 - ϕ nodes have kinds (Acquire, Release, Newlock, Deref, etc.)
 - $\phi \longrightarrow \phi'$: control flow
 - $\phi \xrightarrow{(i)} \phi'$: control enters function at call site i
 - $\phi \xrightarrow{)i} \phi'$: function returns control at call site i
 - Solve using data-flow analysis

Example: generating constraints

```
pthread_mutex_t  $\langle \ell_1 \rangle$  L1 = ...;
```

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

Φ_{in}

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

Φ_1

```
    pthread_mutex_lock(l);
```

Φ_2

```
    *p = 3;
```

Φ_3

```
    pthread_mutex_unlock(l);
```

Φ_{out}

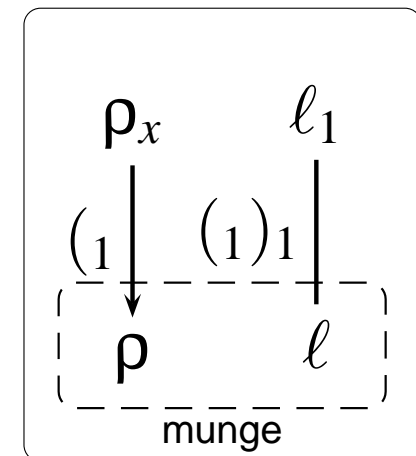
```
}
```

```
...
```

Φ_{call}

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

Φ_{ret}



Example: generating constraints

```
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);
```

```
}
```

```
...
```

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

Φ_{in}

Φ_1

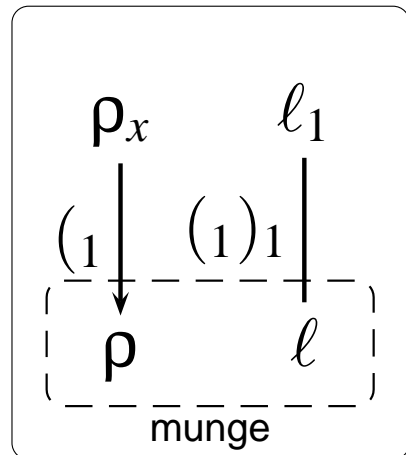
Φ_2

Φ_3

Φ_{out}

Φ_{call}

Φ_{ret}



Example: generating constraints

```
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);
```

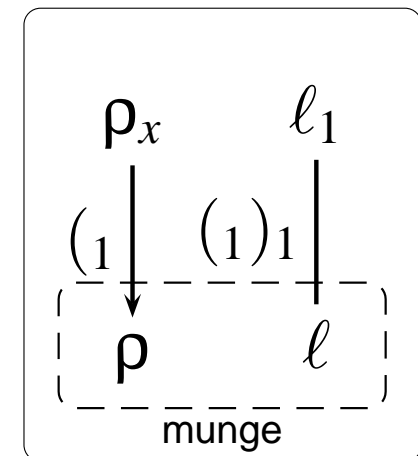
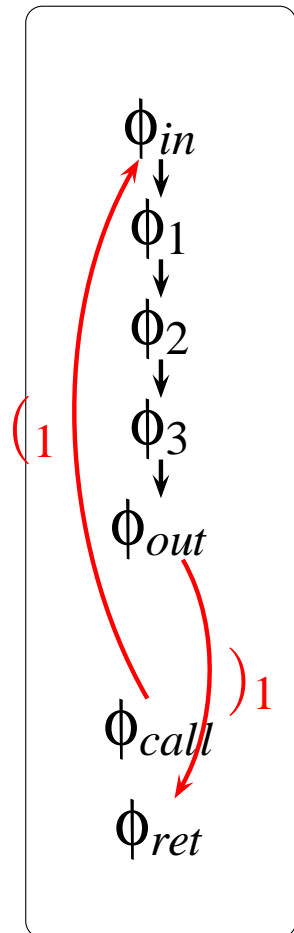
```
    *p = 3;
```

```
    pthread_mutex_unlock(l);
```

```
}
```

```
...
```

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



Example: generating constraints

```
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);
```

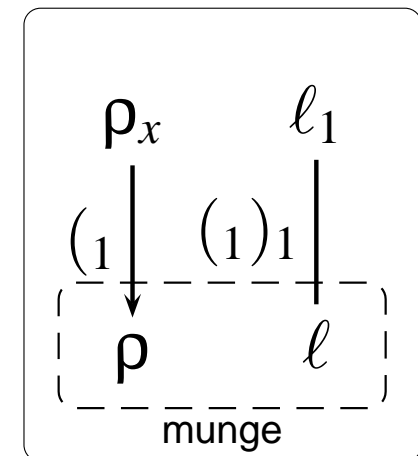
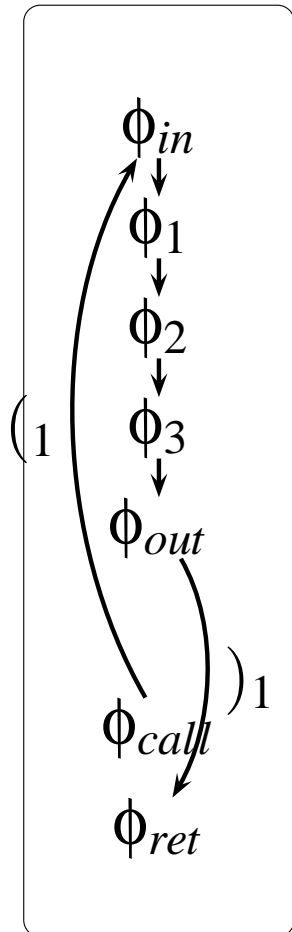
```
    *p = 3;
```

```
    pthread_mutex_unlock(l);
```

```
}
```

```
...
```

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



Example: generating constraints

```
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);
```

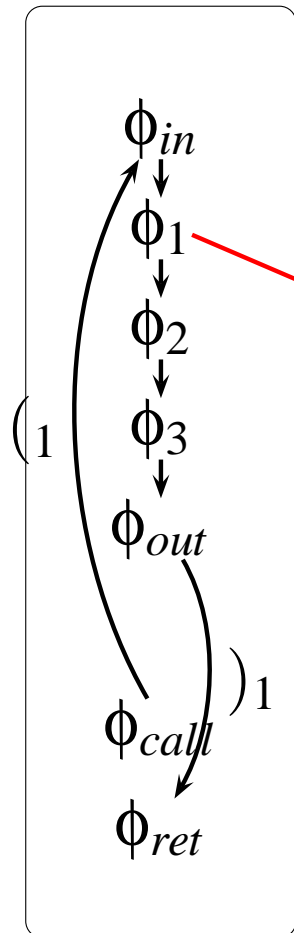
```
    *p = 3;
```

```
    pthread_mutex_unlock(l);
```

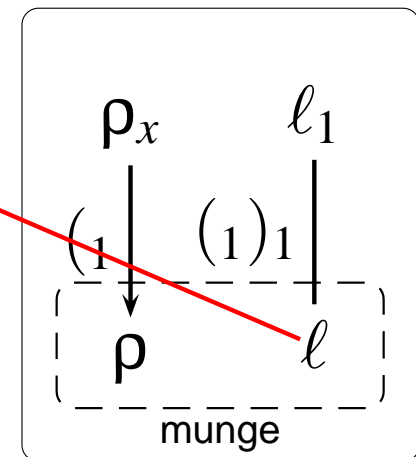
```
}
```

```
...
```

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



Acquired



Example: generating constraints

```
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);
```

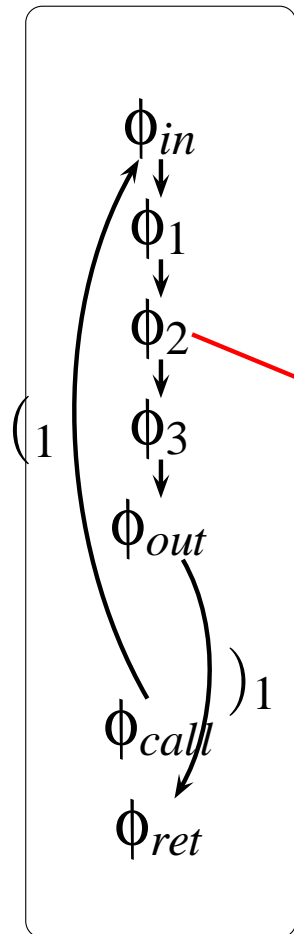
```
    *p = 3;
```

```
    pthread_mutex_unlock(l);
```

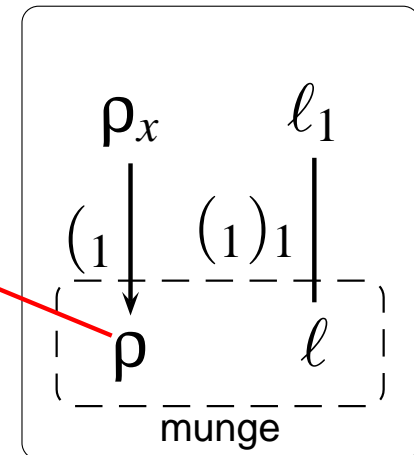
```
}
```

```
...
```

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



Dereferenced



Example: generating constraints

```
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);
```

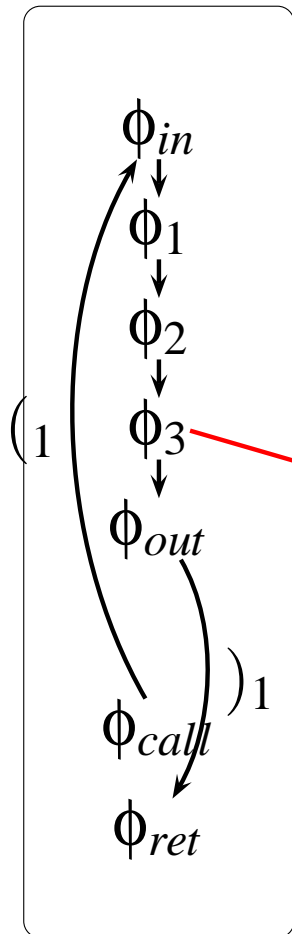
```
    *p = 3;
```

```
    pthread_mutex_unlock(l);
```

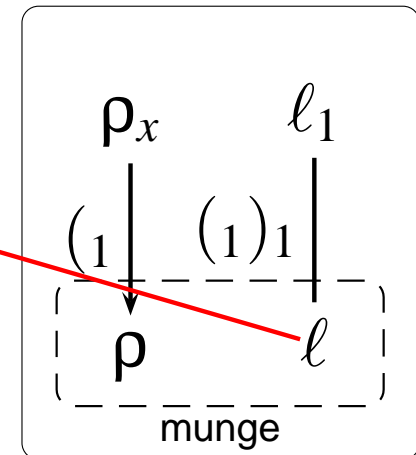
```
}
```

```
...
```

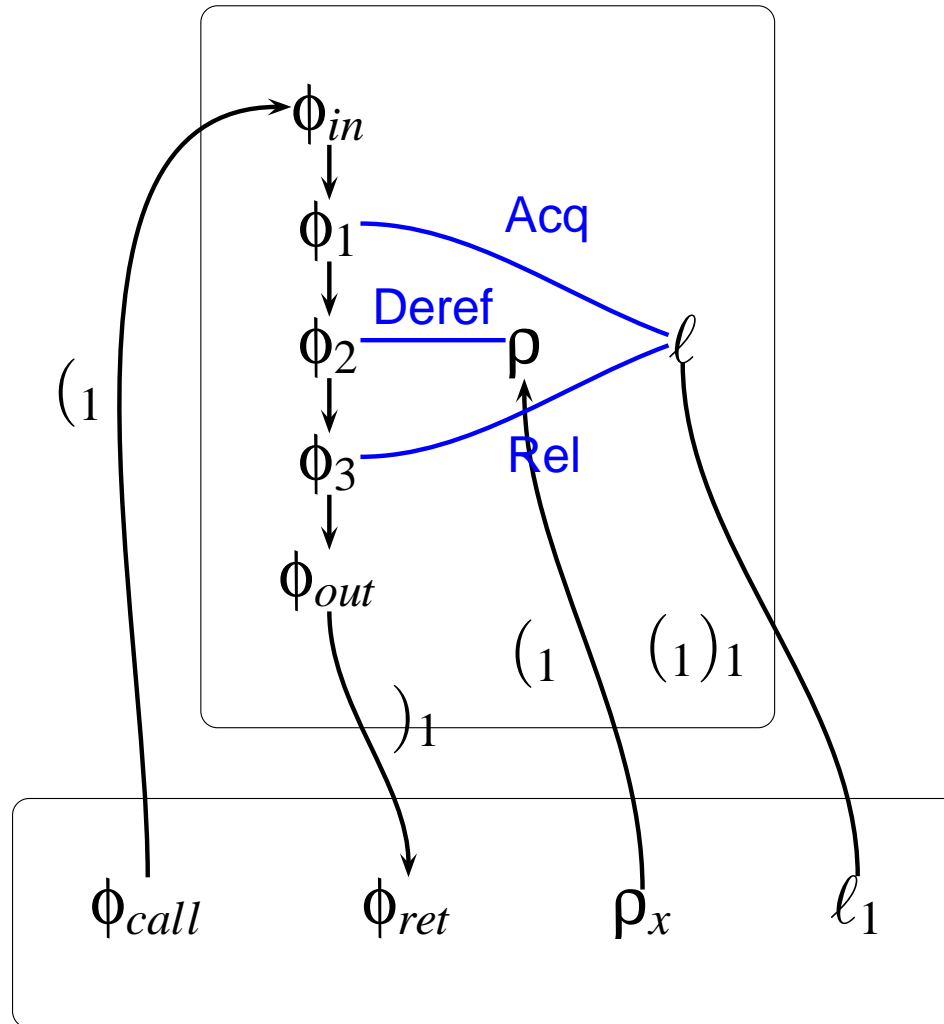
```
munge1(&L1, &x);
```



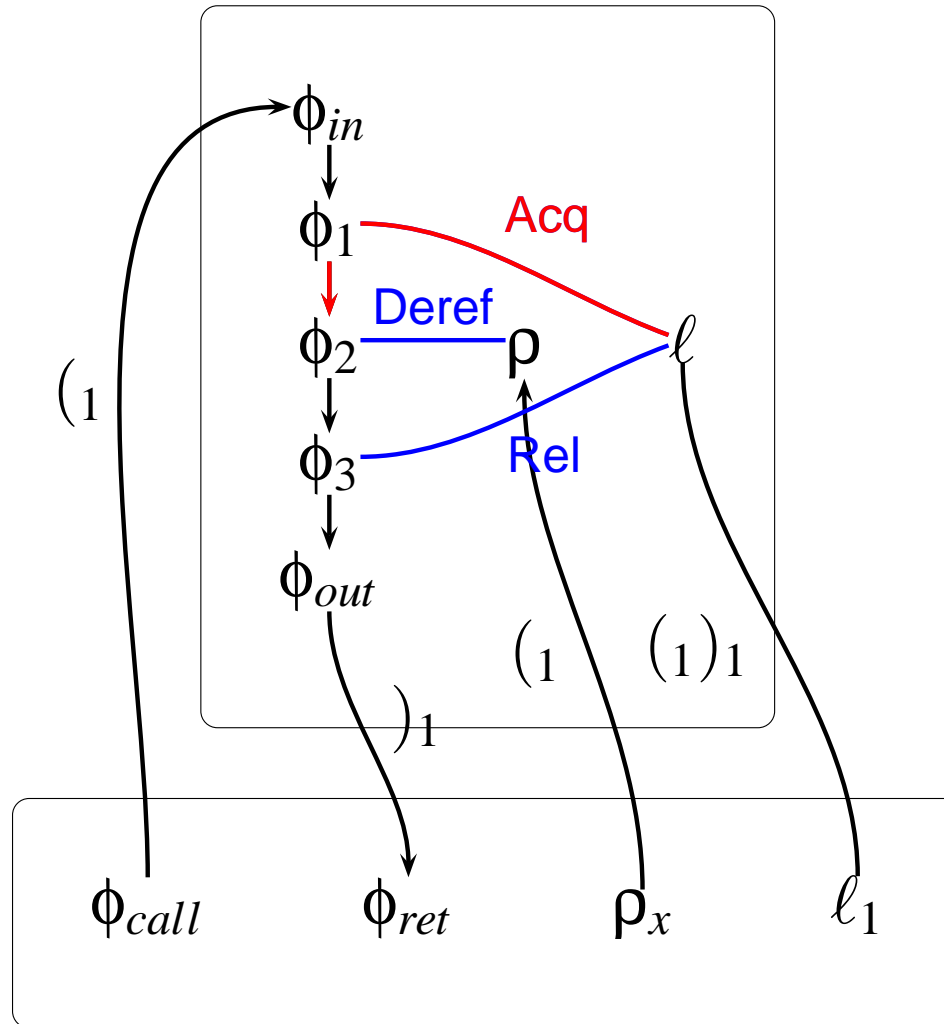
Released



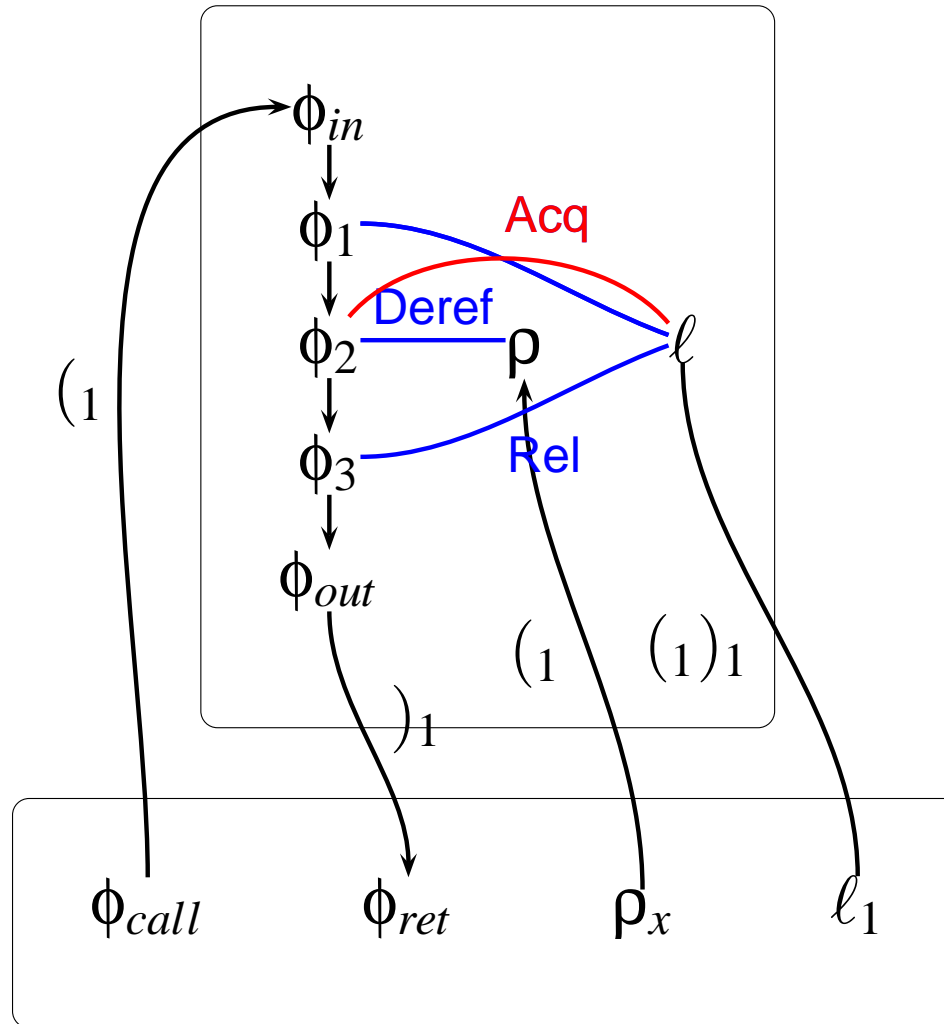
Example: solving constraints



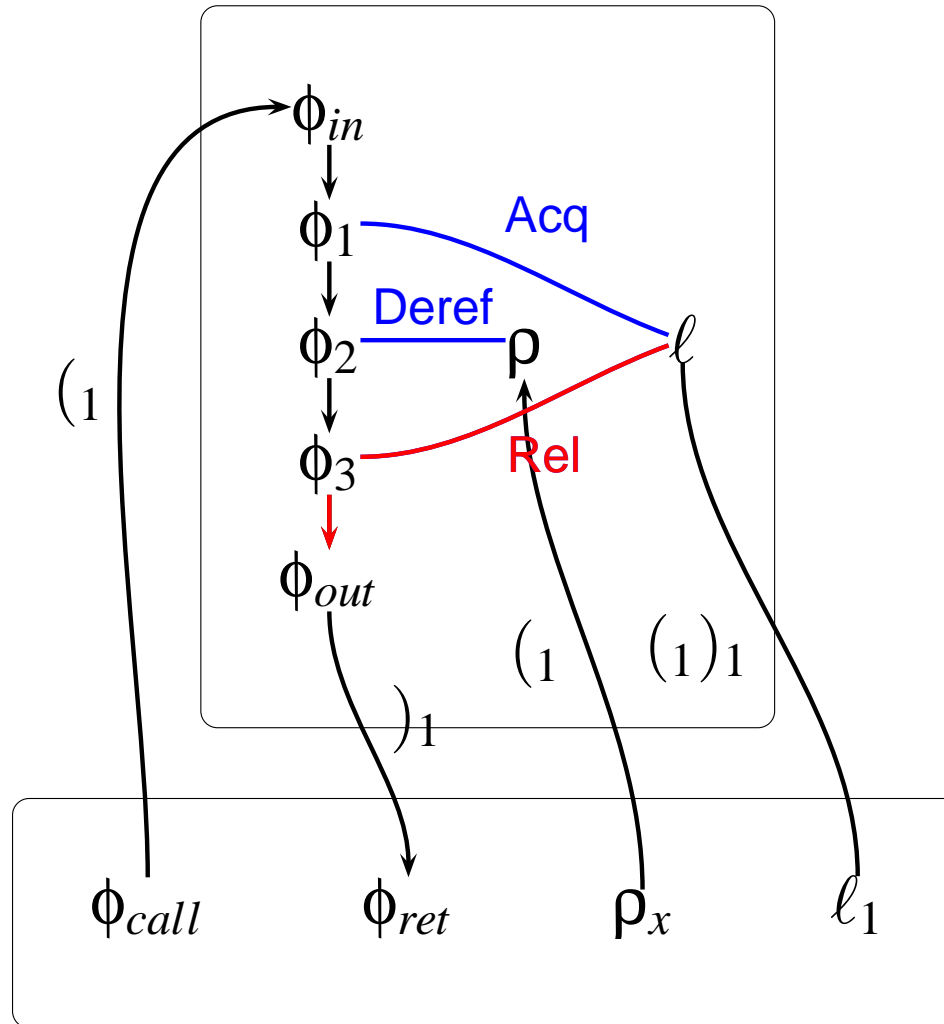
Example: solving constraints



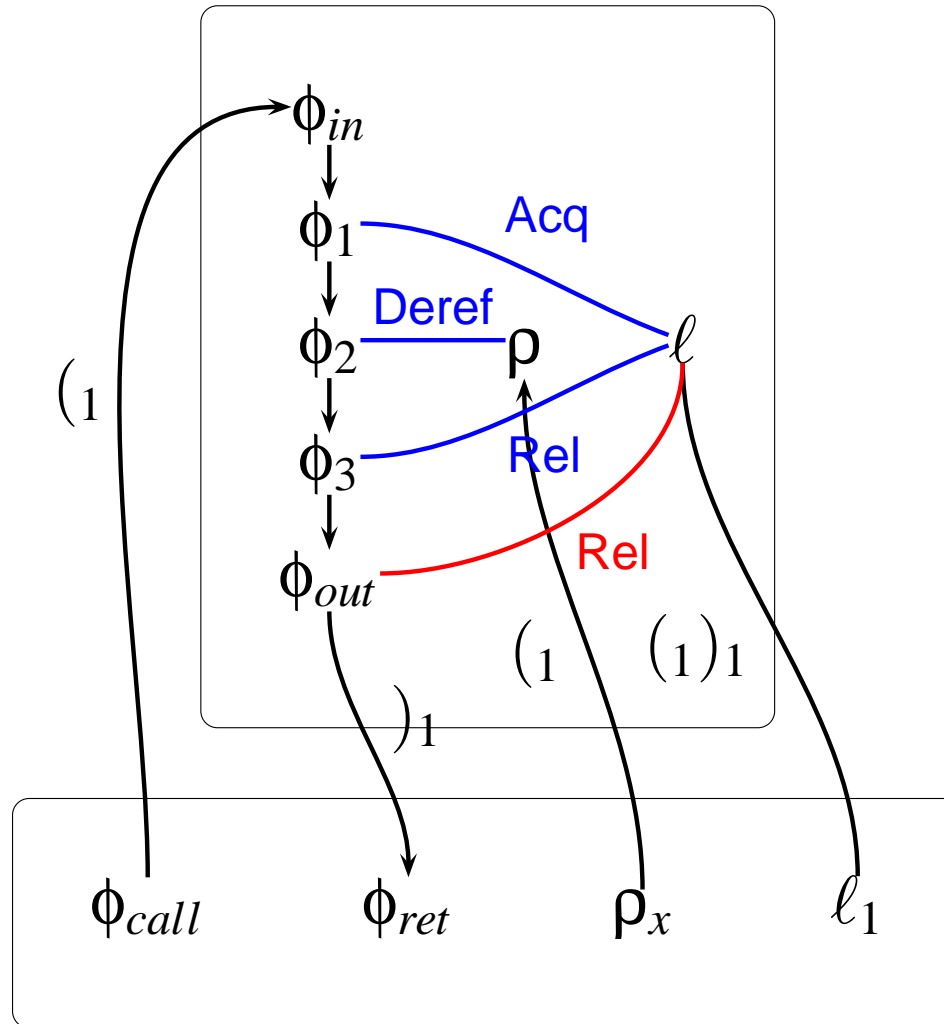
Example: solving constraints



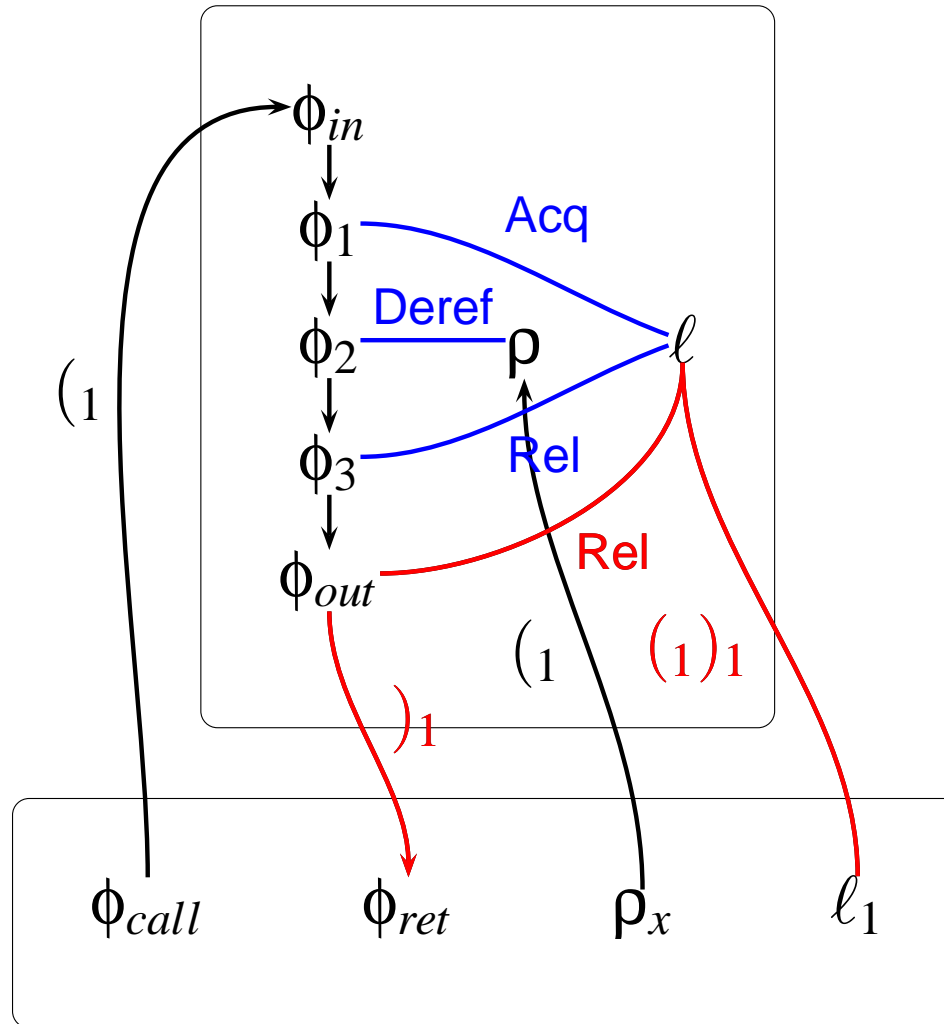
Example: solving constraints



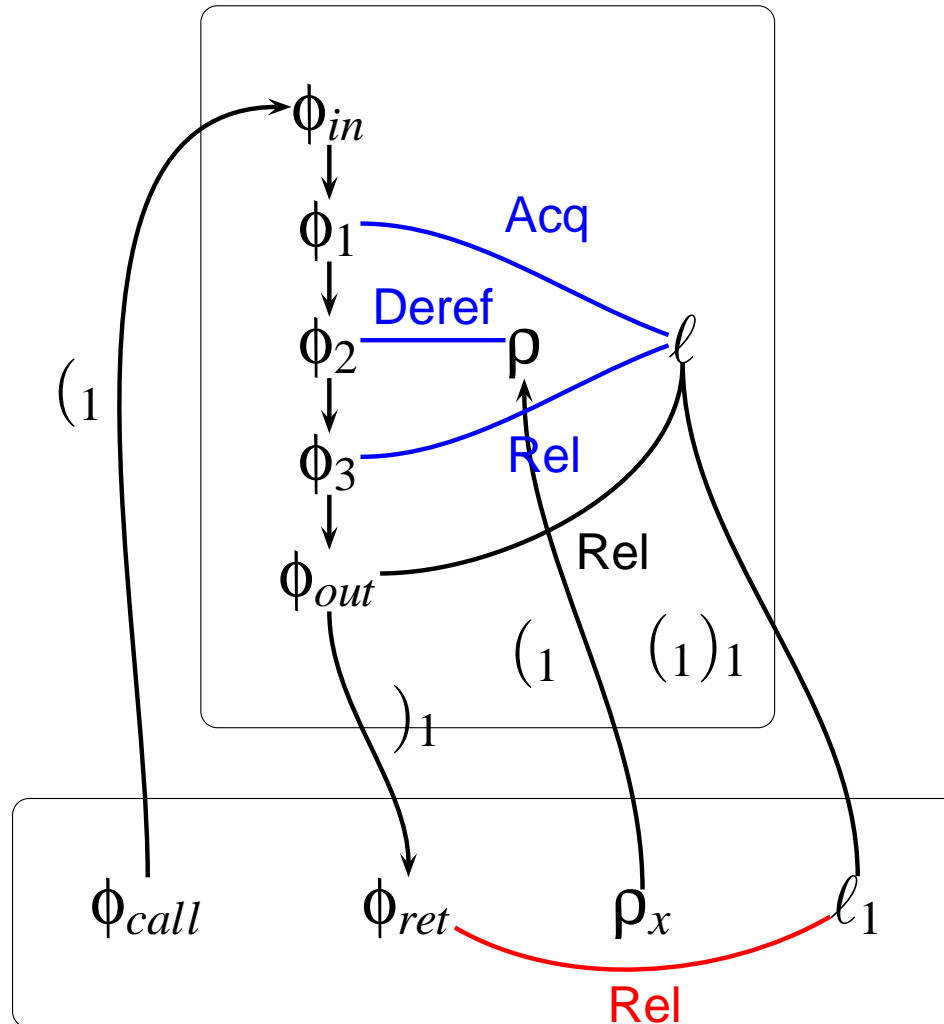
Example: solving constraints



Example: solving constraints



Example: solving constraints



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

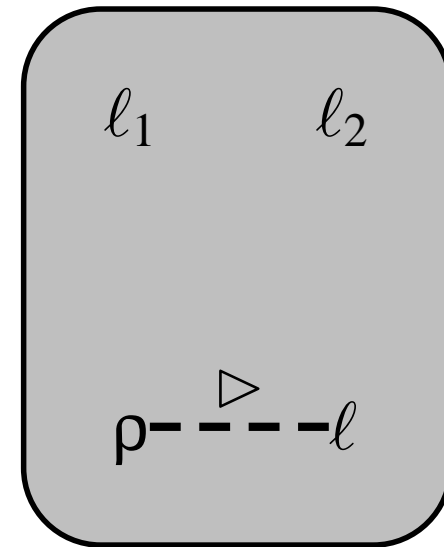
Linearity Effects

- 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>
```

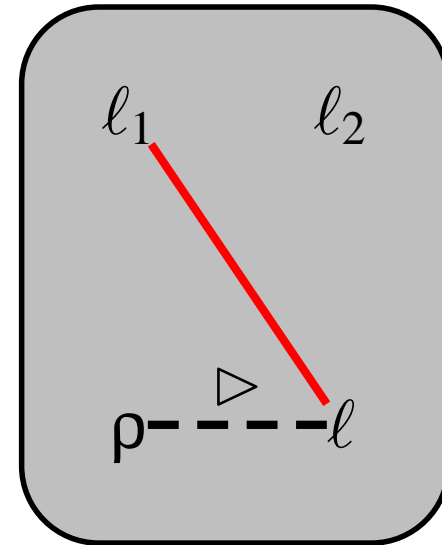
```
pthread_mutex_init(&L1);  
pthread_mutex_init(&L2);  
if(...) l = &L1;  
else l = &L2;  
pthread_mutex_lock(&l);  
x = 42;  
pthread_mutex_unlock(&l);
```



Linearity Example

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

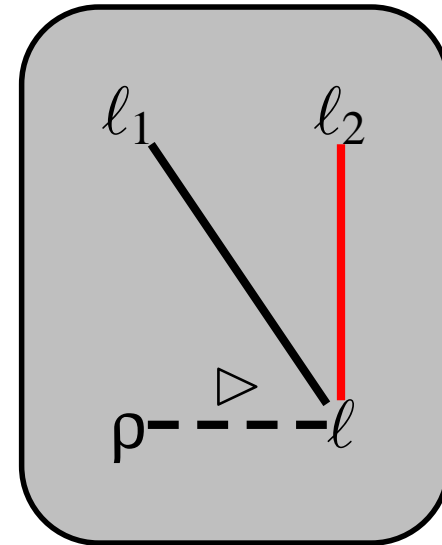
```
pthread_mutex_init(&L1);  
pthread_mutex_init(&L2);  
if(...) l = &L1;  
else l = &L2;  
pthread_mutex_lock(&l);  
x = 42;  
pthread_mutex_unlock(&l);
```



Linearity Example

```
pthread_mutex_t L1<math>\ell_1\ell_2\ellint x; // &x: int*<math>\rho_x
```

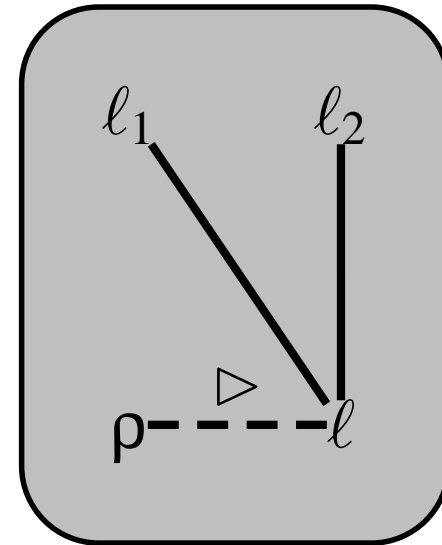
```
pthread_mutex_init(&L1);  
pthread_mutex_init(&L2);  
if(...) l = &L1;  
else l = &L2;  
pthread_mutex_lock(&l);  
x = 42;  
pthread_mutex_unlock(&l);
```



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 = 42;  
pthread_mutex_unlock(&l); 0
```

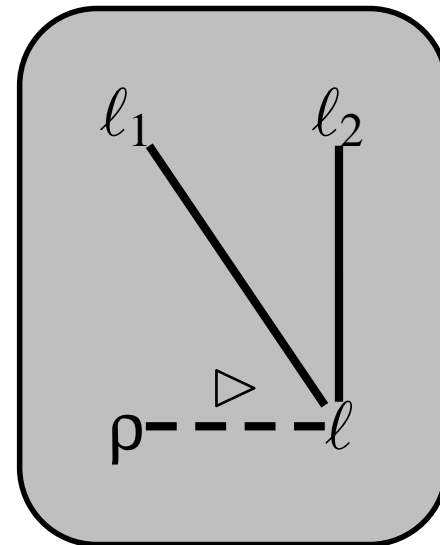


Linearity Example

```
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 = 42;  
pthread_mutex_unlock(&l); 0
```

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

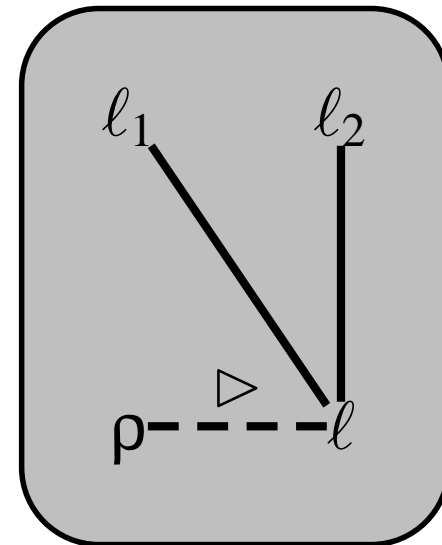


Linearity Example

```
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 = 42;  
pthread_mutex_unlock(&l); 0
```

$l_1 \neq l_2$



Experiments

- Two sets of benchmarks:
 - Stand-alone C applications, use pthreads library
 - Linux device drivers, use kernel threads and spinlocks
- Measured:
 - Overall results—scalability, running time, races found, false positives
 - Precision/performance tradeoff of individual techniques
 - Sharing analysis
 - Field sensitivity
 - Context sensitivity

Results

Benchmark	Size (LOC)	Time (s)	Warnings	Unguarded	Races
aget	1,914	0.85	62	31	31
ctrace	2,212	0.59	10	9	2
engine	2,608	0.88	7	0	0
knot	1,985	0.78	12	8	8
pfscan	1,948	0.46	6	0	0
smtprc	8,624	5.37	46	1	1
3c501	17,443	9.18	15	5	4
eql	16,568	21.38	35	0	0
hp100	20,370	143.23	14	9	8
plip	19,141	19.14	42	11	11
sis900	20,428	71.03	6	0	0
slip	22,693	16.99	3	0	0
sundance	19,951	106.79	5	1	1
synclink	24,691	1521.07	139	2	0
wavelan	20,099	19.70	10	1	1

Precision of sharing analysis

Benchmark	Pointers			Allocation Sites		
	Total	Shared	In scope	Total	Shared	In scope
aget	1,411	258	235	352	64	62
ctrace	1,089	129	116	311	12	12
engine	1,441	60	17	410	11	7
knot	1,238	338	238	321	30	15
pfscan	987	53	48	240	8	7
smtprc	4,275	196	67	1079	74	46
3c501	10,020	954	913	408	20	20
eql	4,572	2,377	2,168	273	43	35
hp100	19,401	5,268	5,210	497	15	15
plip	13,249	2,867	2,823	466	49	49
sis900	38,624	2,648	2,594	779	11	9
slip	13,748	1,338	1,281	382	20	19
sundance	34,142	3,313	3,267	753	9	9
synclink	51,147	11,621	11,472	1,298	155	139
wavelan	18,799	2,535	2,125	695	128	10

Field sensitivity: applications

Benchmark	C.Gen. (s)	Total (s)	Labels	Shared	Warnings
aget	0.55	0.85	5,634	62	62
	0.50	0.67	5,490	62	62
ctrace	0.40	0.59	4,351	12	10
	0.38	0.53	4,285	15	13
engine	0.76	0.88	5,051	7	7
	0.79	0.91	4,989	59	59
knot	0.55	0.78	4,752	15	12
	0.52	0.83	4,566	24	21
pfscan	0.36	0.46	4,143	7	6
	0.36	0.46	4,139	15	14
smtprc	3.09	5.37	14,815	46	46
	3.08	5.14	14,917	97	97

Field sensitive / Field insensitive

Field sensitivity: Linux drivers

Benchmark	C.Gen. (s)	Total (s)	Labels	Shared	Warnings
3c501	7.92	9.18	25,905	20	15
	7.60	18.56	22,976	42	42
eql	2.72	21.38	8,954	35	35
	2.39	17.99	7,484	42	42
hp100	35.92	143.23	31,609	15	14
	34.18	976.12	22,214	41	41
plip	16.41	19.14	24,124	49	42
	17.82	103.21	18,969	60	60
sis900	65.66	71.03	84,797	9	6
	60.45	132.18	71,630	42	42
slip	15.11	16.99	25,371	19	3
	15.44	33.24	18,333	56	31
sundance	96.72	106.79	73,552	9	5
	81.44	timeout	61,570	44	n/a
synclink	1433.56	1521.07	68,643	139	139
	1232.05	timeout	35,542	171	n/a
wavelan	17.89	19.70	30,052	10	10
	16.90	40.19	21,071	43	44

Field sensitive / Field insensitive

Context sensitivity

Benchmark	Context-sensitive		Context-insensitive	
	Time (s)	Warnings	Time (s)	Warnings
aget	0.85	62	0.64	77
ctrace	0.59	10	0.42	21
engine	0.88	7	0.60	15
knot	0.78	12	0.60	31
pfscan	0.46	6	0.50	26
smtprc	5.37	46	4.16	128
3c501	9.18	15	0.75	20
eql	21.38	35	0.86	41
hp100	143.23	14	2.76	25
plip	19.14	42	1.39	46
sis900	71.03	6	Out of Mem.	n/a
slip	16.99	3	Out of Mem.	n/a
sundance	106.79	5	1.32	20
synclink	1521.07	139	23.42	227
wavelan	19.70	10	9.59	143

Contribution: LOCKSMITH

- Discovers races automatically by inferring consistent correlation
- LOCKSMITH: Implementation for C
 - Practical, fast, precise; bounded by memory usage in aliasing analysis
 - Requires no annotations (minimal annotations when using existential context sensitivity)
 - Found races in existing programs and Linux drivers
- Extensive evaluation
 - Measured the effect each analysis has on LOCKSMITH
 - Explored precision/performance tradeoff

Conclusions

- LOCKSMITH automatically finds data races in C programs
- Uses novel static analyses
 - Context-sensitive correlation analysis to infer “guarded-by” relation
 - Existential context sensitive label flow analysis for precision with data structures
 - Contextual effects for inferring shared locations
 - Mechanized proof of soundness in Coq
- Scales to the full C language

<http://www.cs.umd.edu/projects/PL/locksmith>