LOCKSMITH: Context-sensitive Correlation Analysis for Detecting Races

Polyvios Pratikakis
Jeffrey S. Foster
Michael Hicks

University of Maryland, College Park

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

A way to prevent races

- Shared locations ρ, locks ℓ
- Correlation $\rho \rhd \ell$:
  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

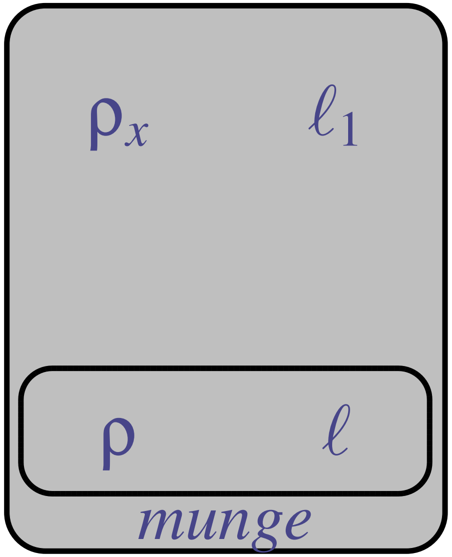
- Static analysis for inference of *correlation* between locks and pointers
- Context sensitivity
  - Universal polymorphism for function calls
  - Existential polymorphism for data structures
- *Sound* race detection using assertion of *consistent correlation*
  - Formalised for $\lambda_{\triangleright}$,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

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);
```

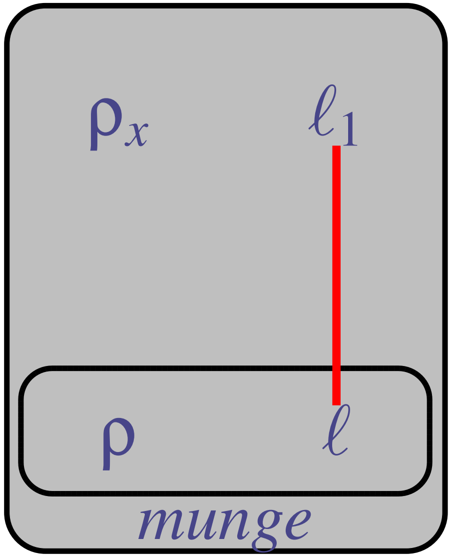
Correlation

```
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);
```



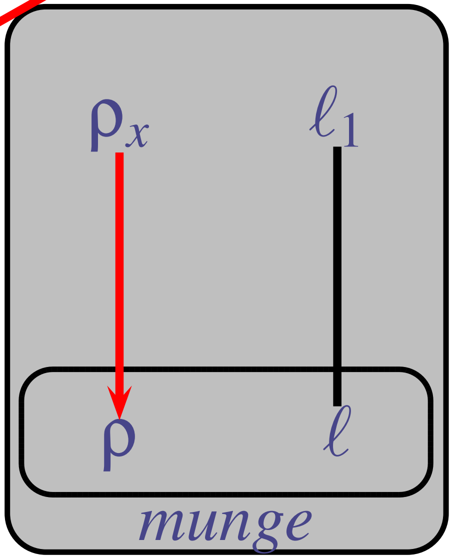
Correlation

```
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);
```



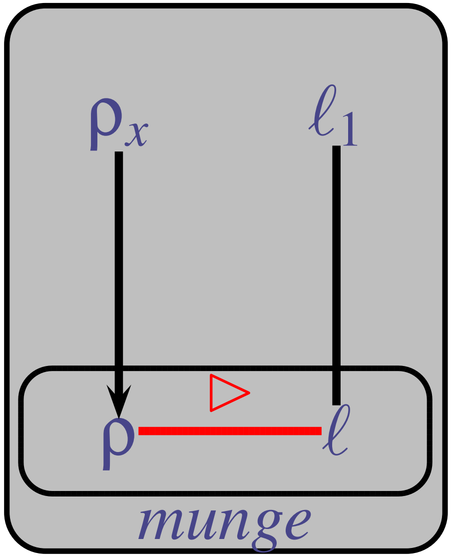
Correlation

```
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);
```



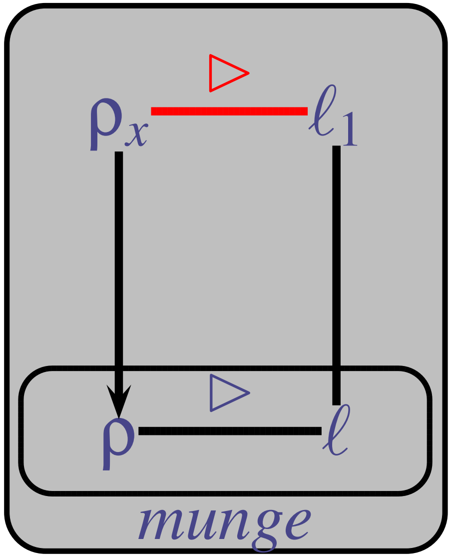
Correlation

```
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);
```



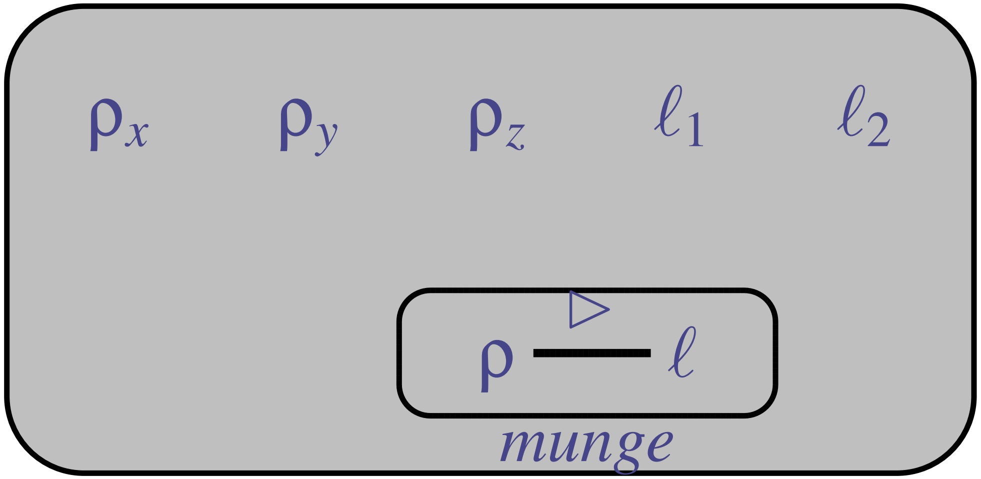
Correlation

```
pthread_mutex_t⟨ℓ_1⟩ L1 = ...;
int x; // &x:  int*⟨ρ_x⟩
void munge(pthread_mutex_t⟨ℓ⟩ *l, int *⟨ρ⟩ p) {
   pthread_mutex_lock(l);
   *p = 3;
   pthread_mutex_unlock(l);
}
...
munge(&L1, &x);
```



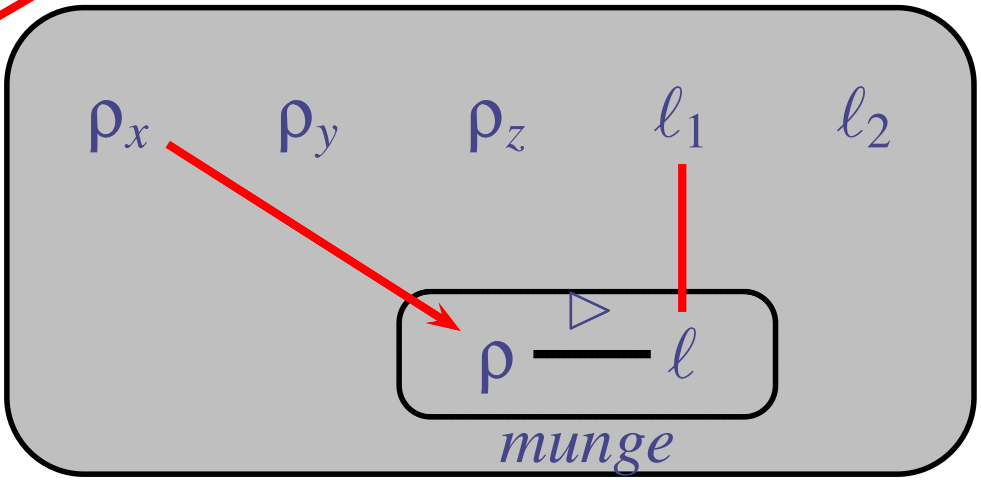
Context Sensitivity

```
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);
```



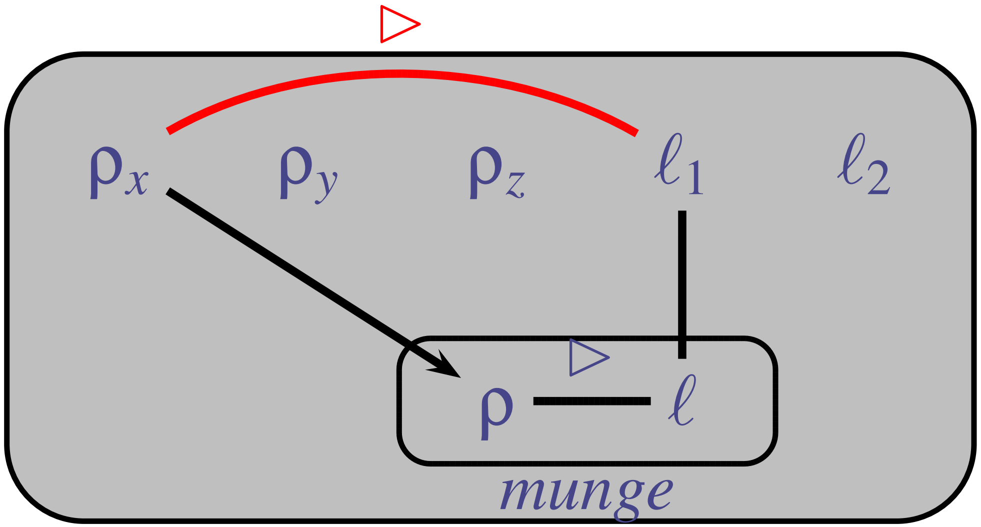
Context Sensitivity

```
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);
```



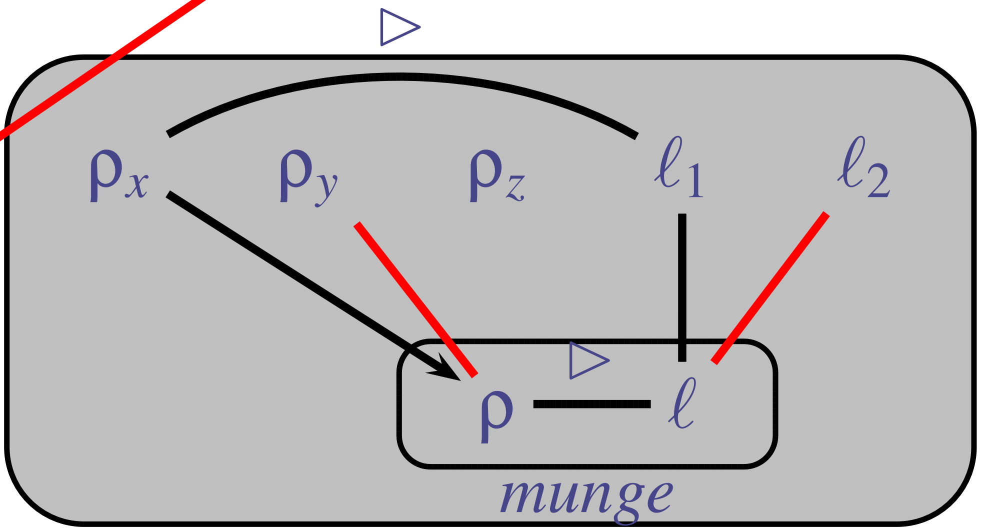
Context Sensitivity

```
pthread_mutex_t⟨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);
```



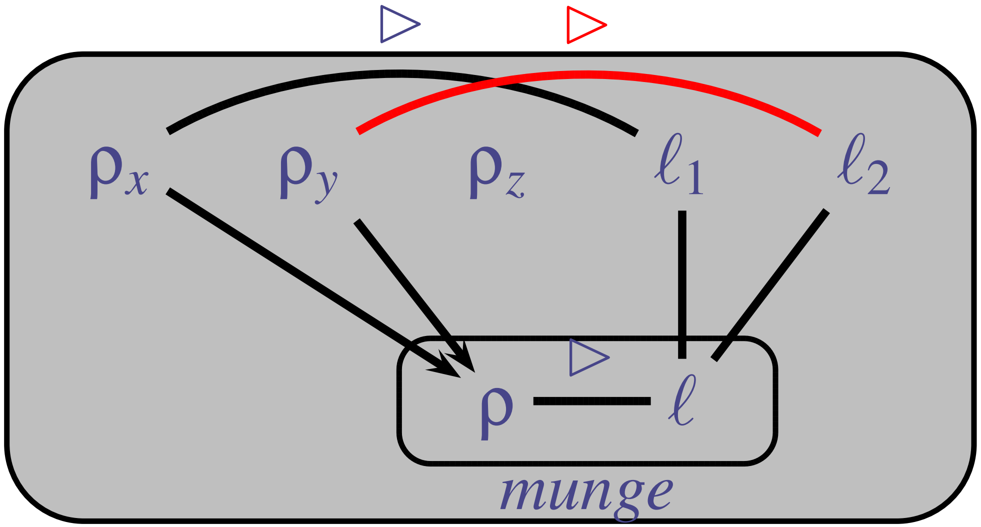
Context Sensitivity

```
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);
```



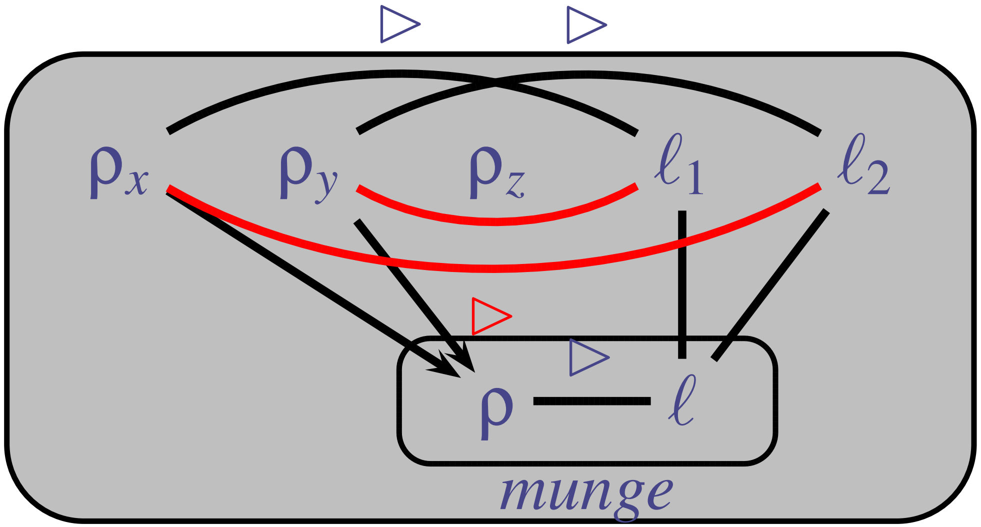
Context Sensitivity

```
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`$\langle\ell_1\rangle$ `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);`

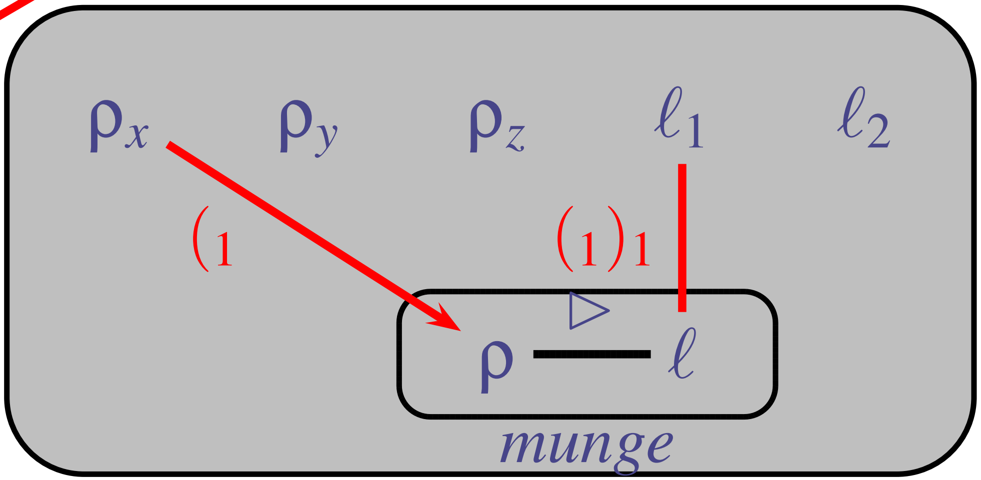
`}`

`...`

`munge`^{1}`(&L1, &x);`

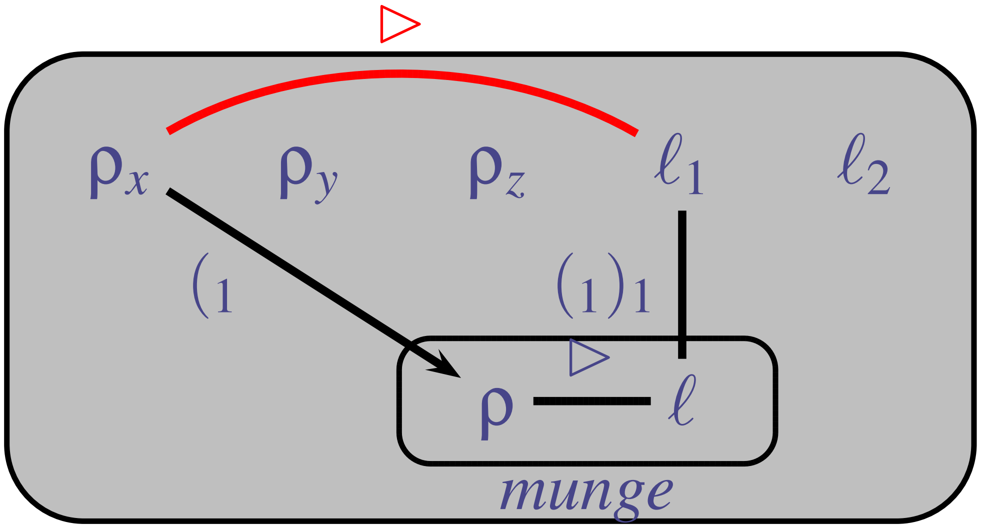
`munge`^{2}`(&L2, &y);`

`munge`^{3}`(&L2, &z);`



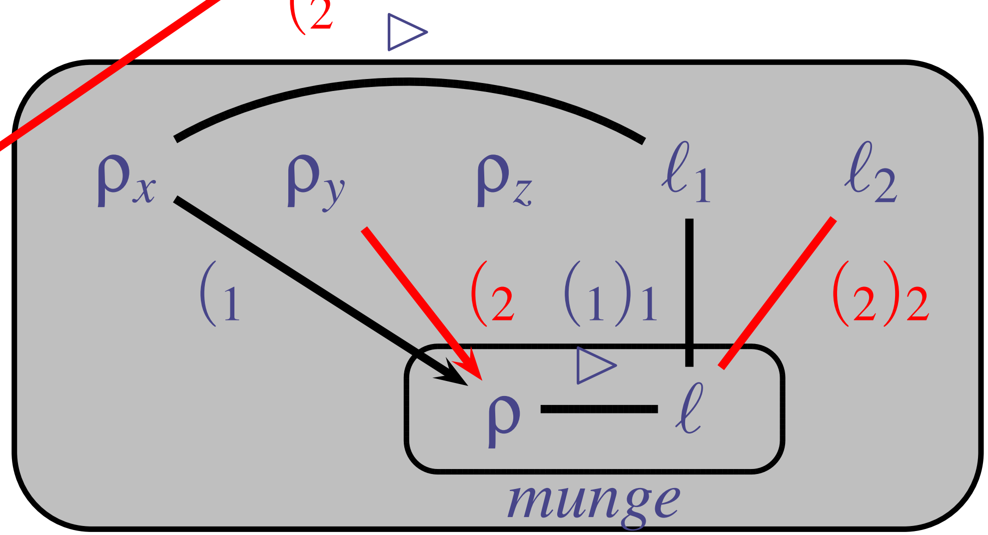
Context Sensitivity

```
pthread_mutex_t⟨ℓ₁⟩ L1 = ..., ⟨ℓ₂⟩ L2 = ...;
int x, y, z; // ⟨ρₓ⟩, ⟨ρᵧ⟩, ⟨ρ_z⟩
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);
```



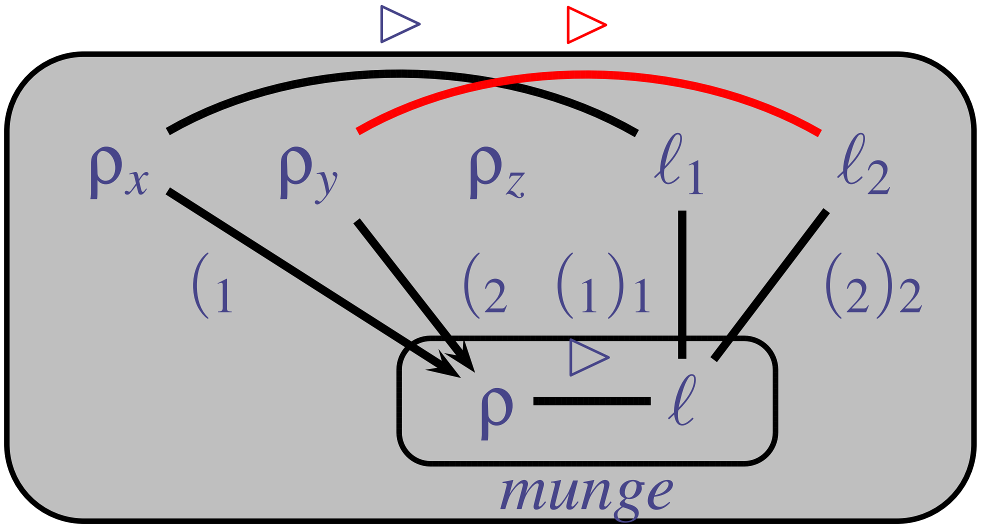
Context Sensitivity

```
pthread_mutex_t$\langle\ell_1\rangle$ 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$^1$(&L1, &x);
munge$^2$(&L2, &y);
munge$^3$(&L2, &z);
```



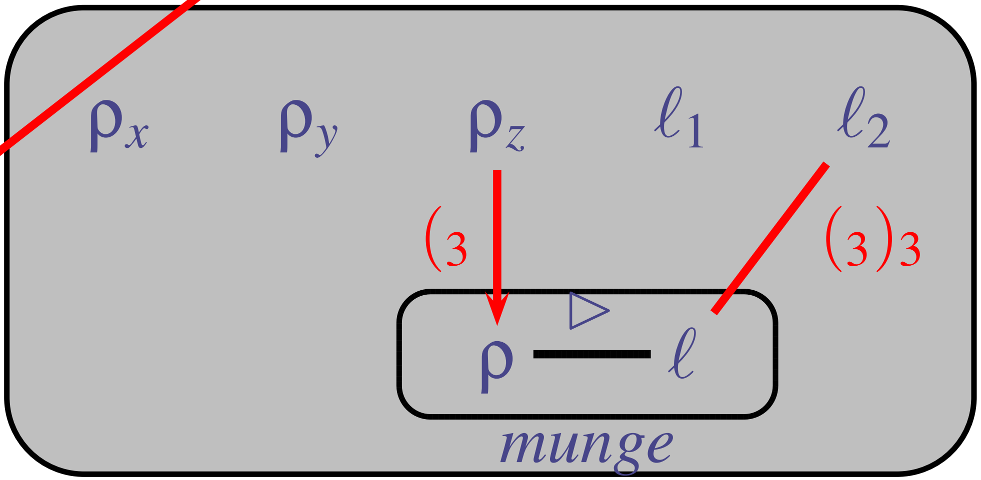
Context Sensitivity

```
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);
}
...
munge$^1$(&L1, &x);
munge$^2$(&L2, &y);
munge$^3$(&L2, &z);
```



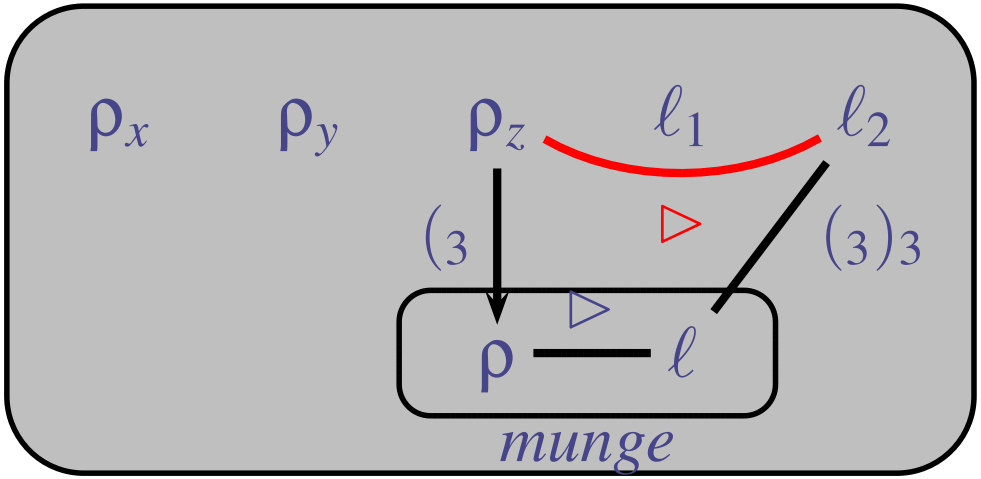
Context Sensitivity

```
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^1(&L1, &x);
munge^2(&L2, &y);
munge^3(&L2, &z);
```



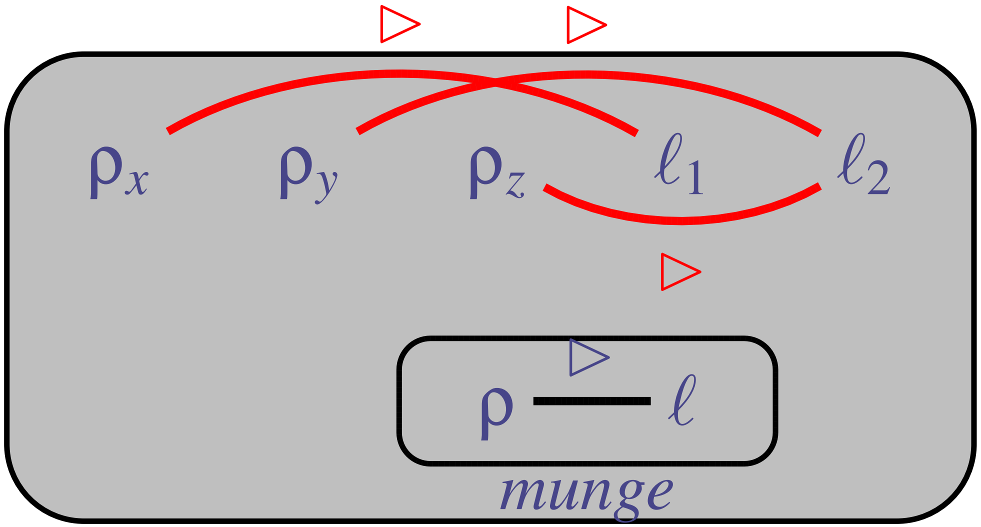
Context Sensitivity

```
pthread_mutex_t⟨ℓ_1⟩ L1 = ..., ⟨ℓ_2⟩ L2 = ...;
int x, y, z; // ⟨ρ_x⟩, ⟨ρ_y⟩, ⟨ρ_z⟩
void munge(pthread_mutex_t⟨ℓ⟩ *l, int *⟨ρ⟩ p) {
   pthread_mutex_lock(l);
   *p = 3;
   pthread_mutex_unlock(l);
}
...
munge^1(&L1, &x);
munge^2(&L2, &y);
munge^3(&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);
}
...
munge$^1$(&L1, &x);
munge$^2$(&L2, &y);
munge$^3$(&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

Soundness

- Formal system for a functional language: $\lambda_{\triangleright}$
- Proof: type safety in $\lambda_{\triangleright}$ 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

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

Ψ_{in}

Ψ_1

Ψ_2

Ψ_3

Ψ_{out}

Ψ_{call}

Ψ_{ret}

`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);`

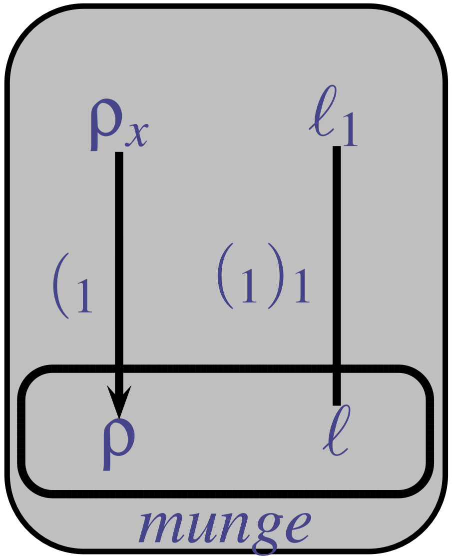
`    *p = 3;`

`    pthread_mutex_unlock(l);`

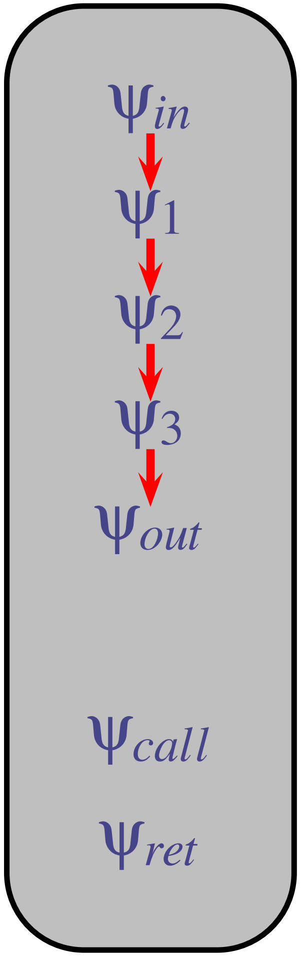
`}`

`...`

`munge`^{1}`(&L1, &x);`



Example: generating constraints



ψ_{in}

ψ_1

ψ_2

ψ_3

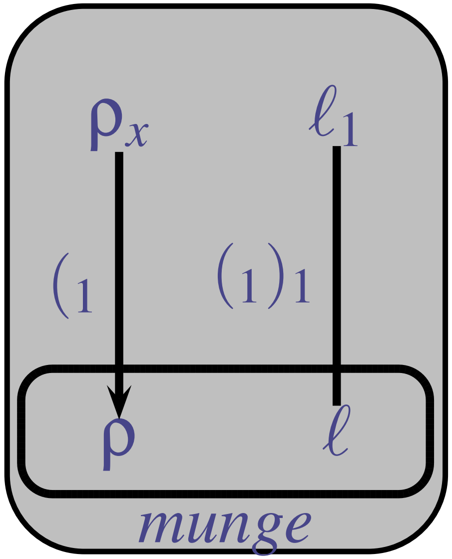
ψ_{out}

ψ_{call}

ψ_{ret}

```
pthread_mutex_t⟨ℓ_1⟩ L1 = ...;
int x; // &x:  int*⟨ρ_x⟩

void munge(pthread_mutex_t⟨ℓ⟩ *l, int *⟨ρ⟩ p) {
    pthread_mutex_lock(l);
    *p = 3;
    pthread_mutex_unlock(l);
}
...
munge^1(&L1, &x);
```



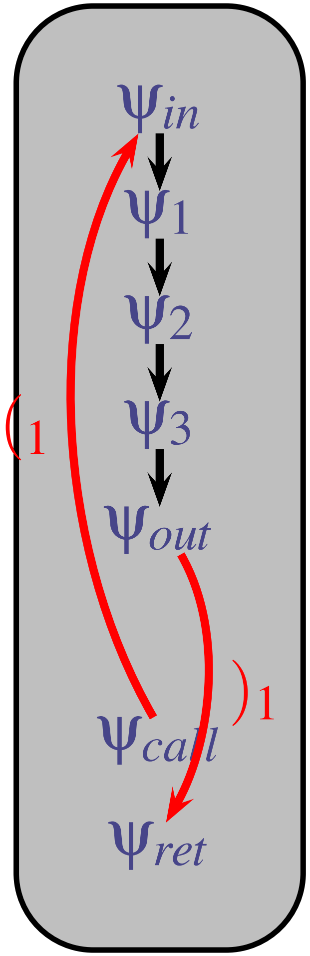
ρ_x ℓ_1

$(_1$ $(_1)_1$

ρ ℓ

munge

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);`

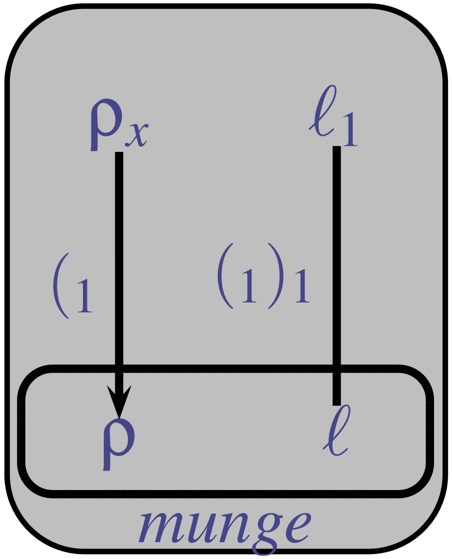
    `*p = 3;`

    `pthread_mutex_unlock(l);`

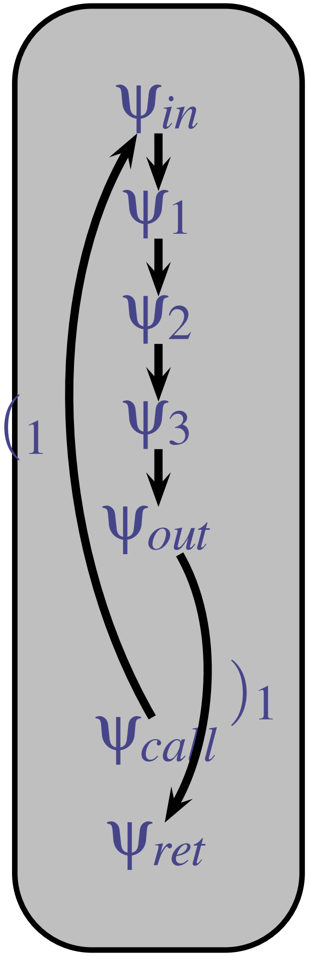
`}`

`...`

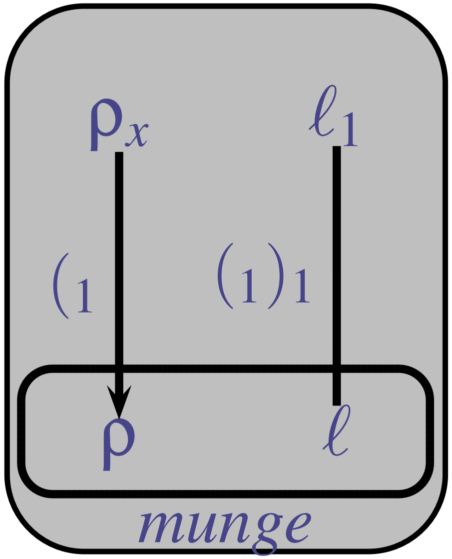
`munge`^{1}`(&L1, &x);`



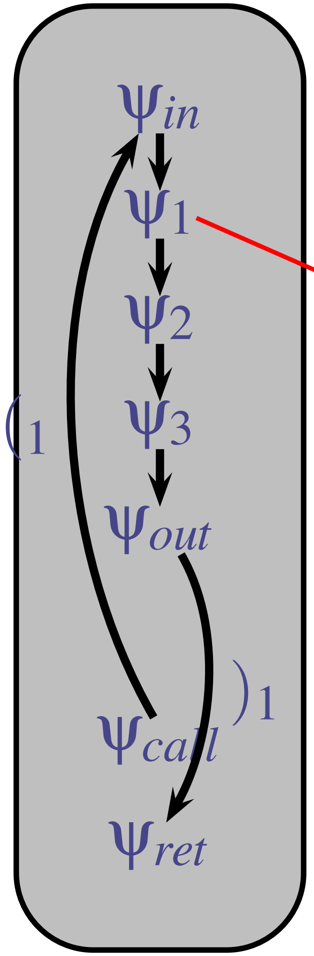
Example: generating constraints



```
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);
}
...
munge^1(&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;`

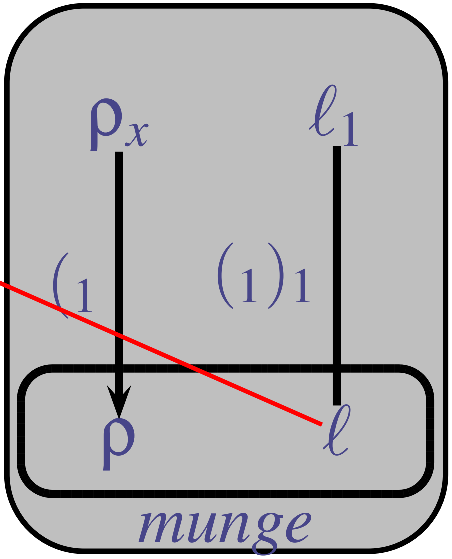
`    pthread_mutex_unlock(l);`

`}`

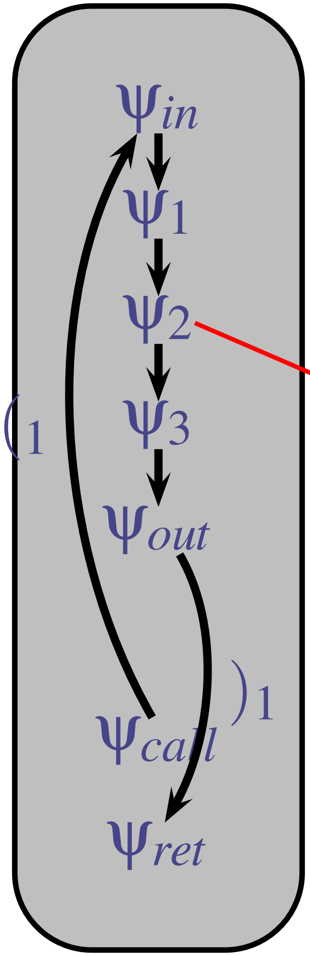
`...`

`munge`^{1}`(&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;`

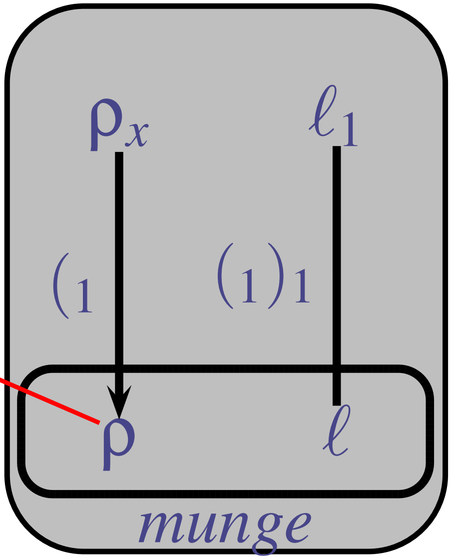
`    pthread_mutex_unlock(l);`

`}`

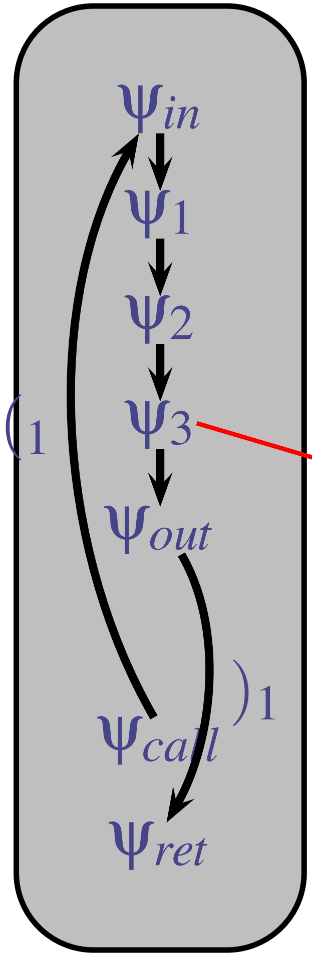
`...`

`munge`^{1}`(&L1, &x);`

Dereferenced

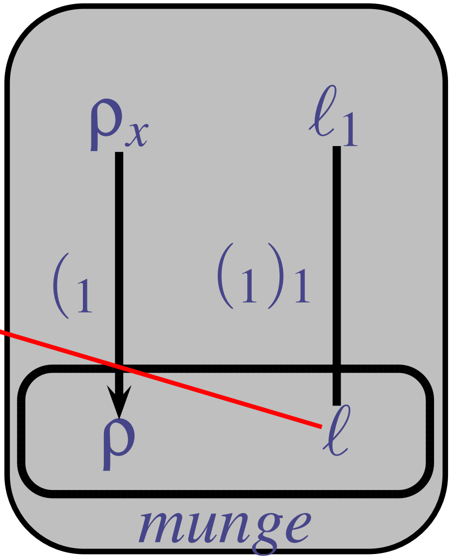


Example: generating constraints

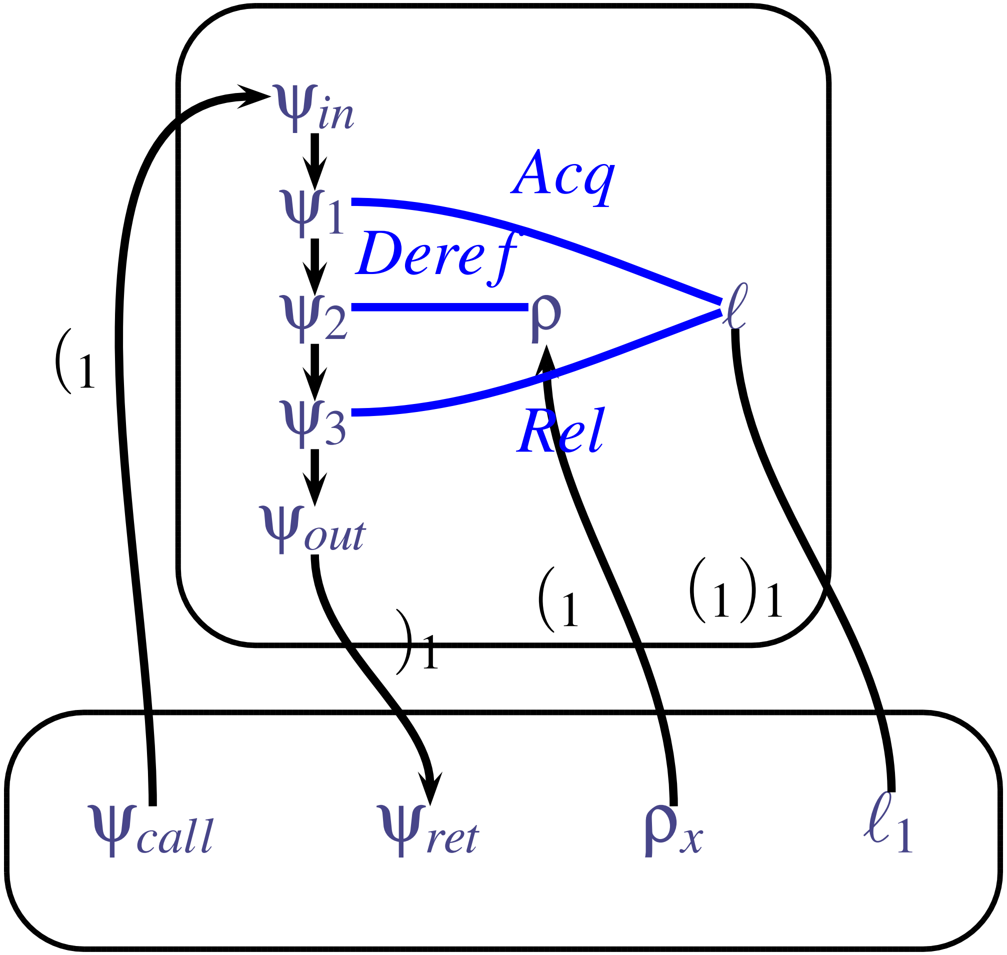


```
pthread_mutex_t⟨ℓ1⟩ L1 = ...;
int x; // &x:  int*⟨ρx⟩
void munge(pthread_mutex_t⟨ℓ⟩ *l, int *⟨ρ⟩ p) {
   pthread_mutex_lock(l);
   *p = 3;
   pthread_mutex_unlock(l);
}
...
munge^1(&L1, &x);
```

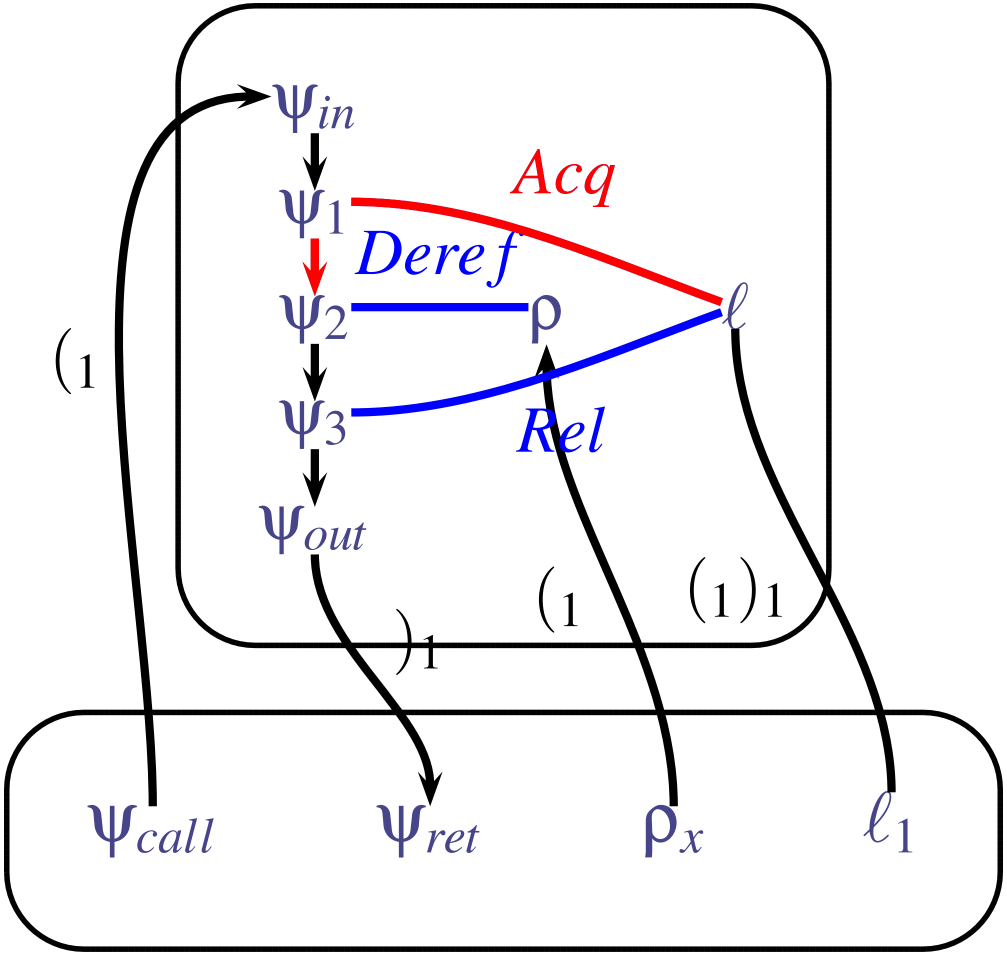
Released



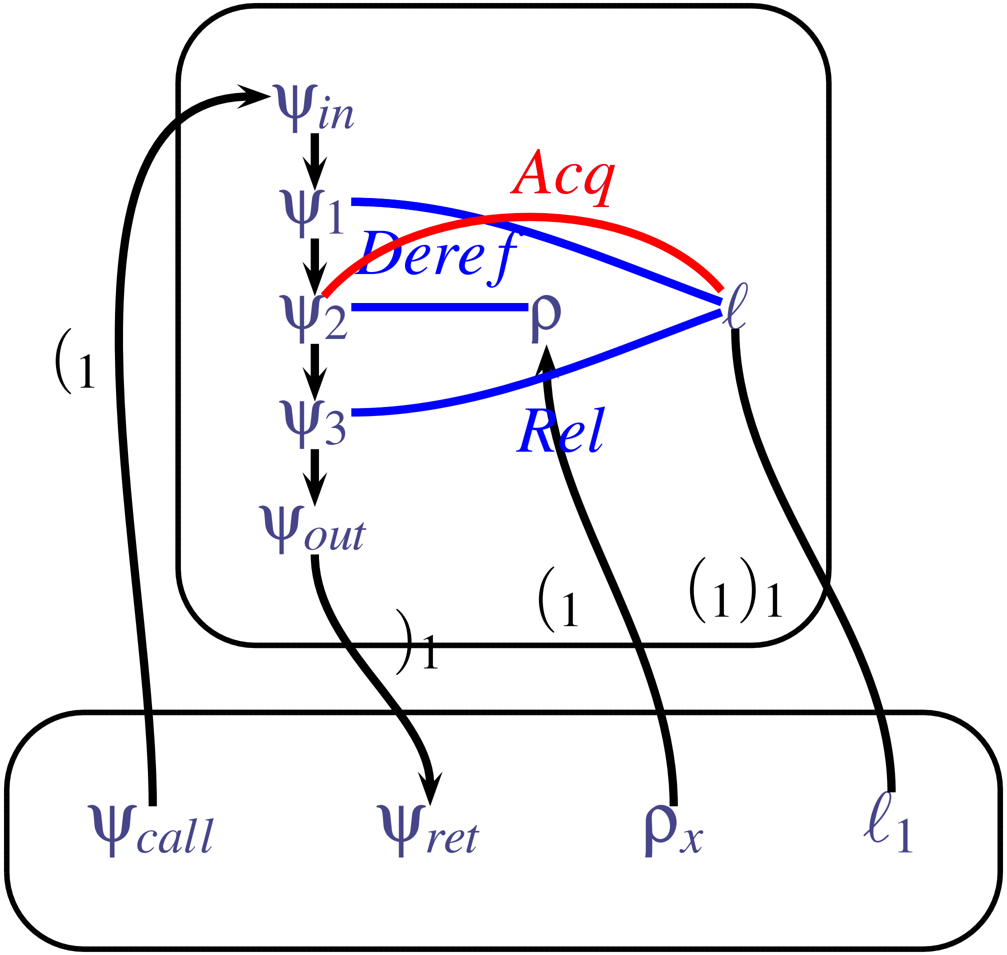
Example: solving constraints



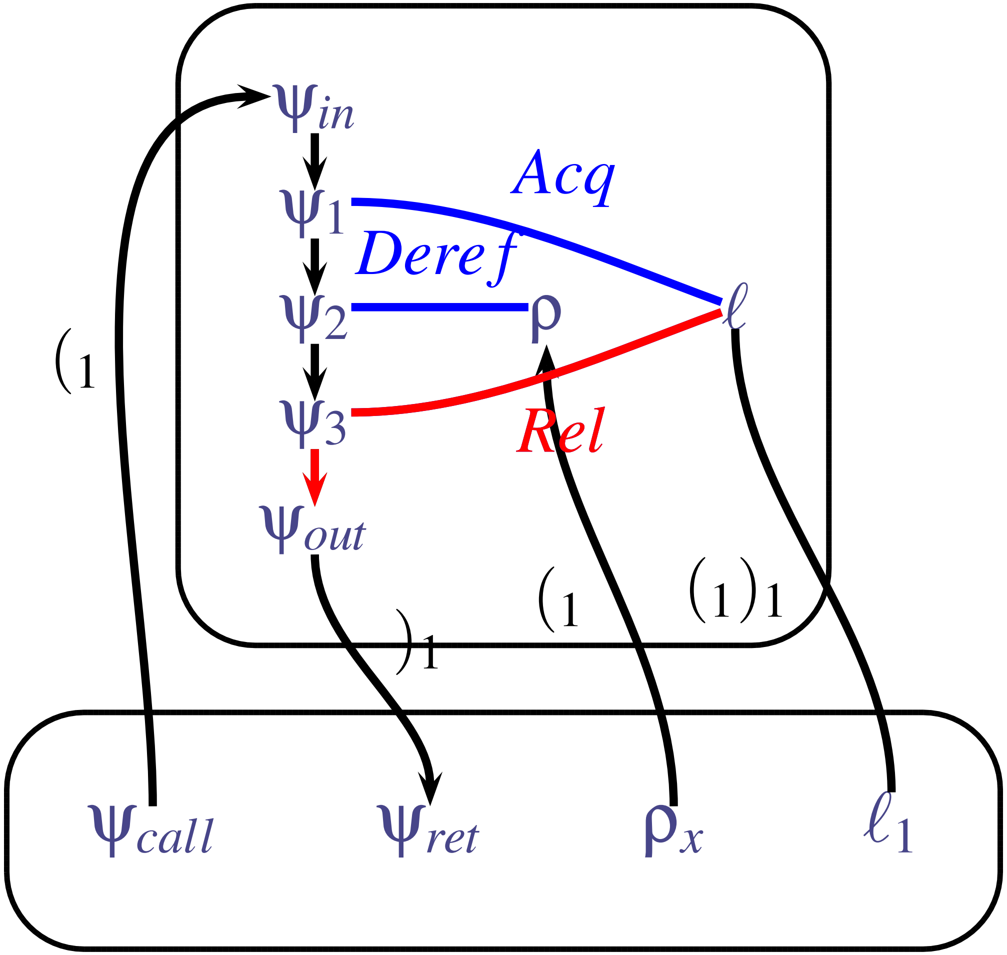
Example: solving constraints



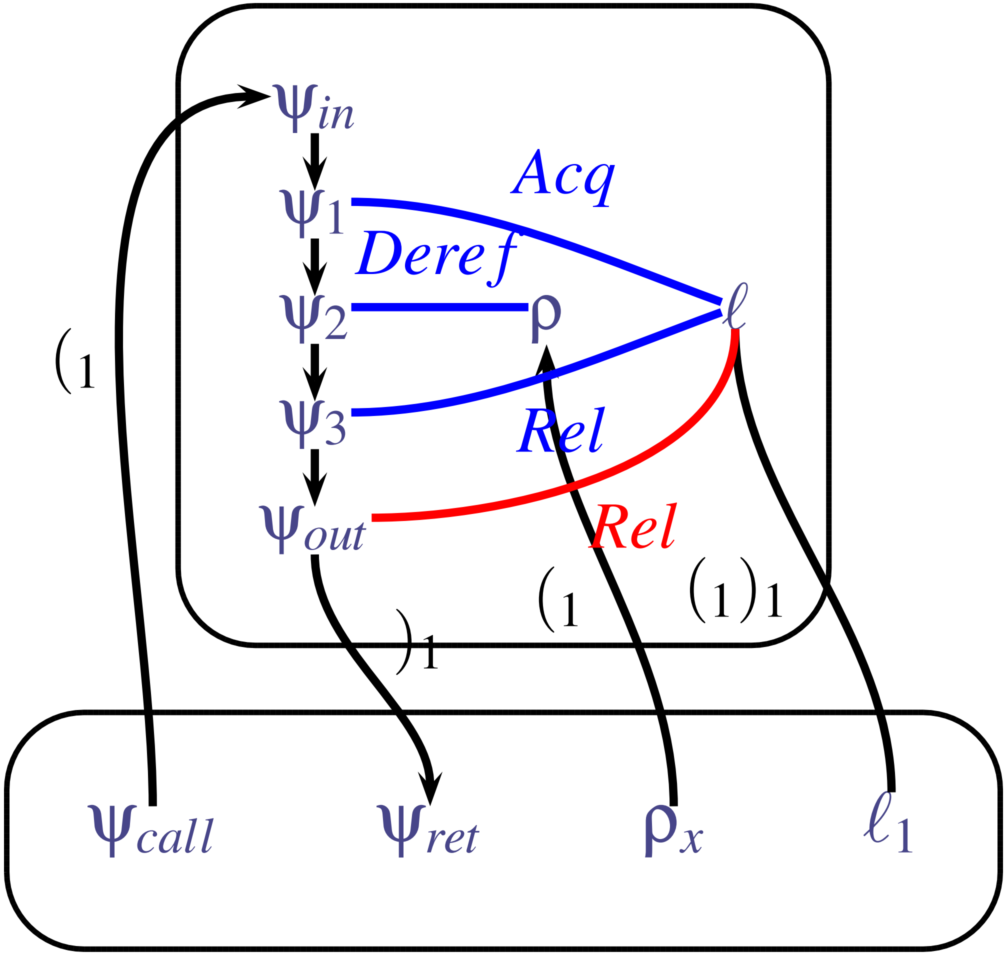
Example: solving constraints



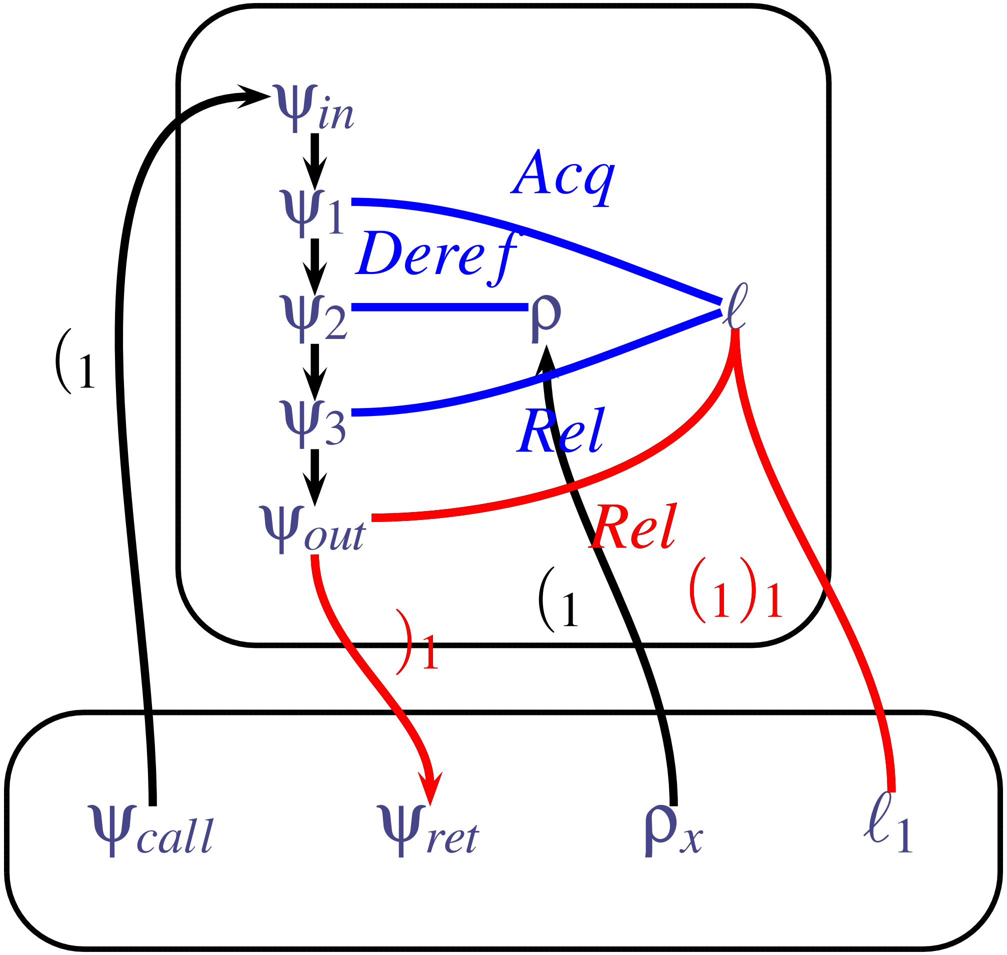
Example: solving constraints



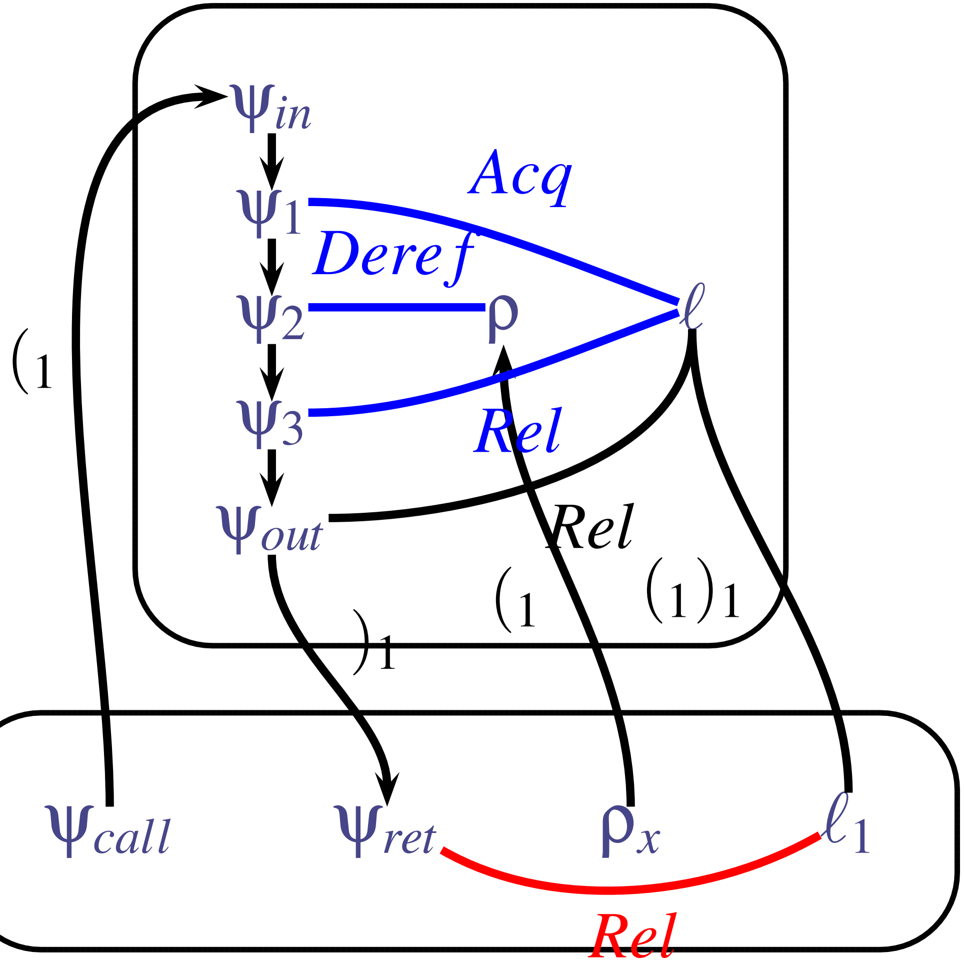
Example: solving constraints



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 { ∃ρ,ℓ.ρ ▷ ℓ
   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
- More details in the paper
  - Full description in SAS'06

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