Checking and Inferring Local Non-Aliasing

Alex Aiken
UC Berkeley

Jeffrey S. Foster
UMD College Park

John Kodumal
UC Berkeley

Tachio Terauchi
UC Berkeley
Introduction

• Aliasing: A long-standing problem

  - Pointers are hard to analyze
  
    ...\*p = 3 ... what is updated?

  - We need to know for
    - compilers (optimization)
    - software analysis tools (CQual, Cyclone, Vault, SLAM, ESP, ...)

Alias Analysis

• What expressions may refer to same location?
  - To what memory locations do expressions point?

• Alias analysis abstracts memory with bounded set of locations
  - Choices affect subsequent analysis

• Programmer has little input to alias analysis
Example from CQual: Modeling Arrays

lock *locks[n]; // define array of n locks

• CQual: All elements of locks[] may alias
  - Represented with a single abstract location
Example from CQual

```c
void foo(int i) {
    do_with_lock(locks[i]);
}
void do_with_lock(lock *l) {
    spin_lock(l);    // unlocked
    work();
    spin_unlock(l);
}
```
Example from CQual: Weak Update

- After acquiring lock, $r$ is locked or unlocked
  - This is a weak update
Example from CQual

```c
void foo(int i) {
    do_with_lock(locks[i]);
}

void do_with_lock(lock *l) {
    spin_lock(l);
    work();
    spin_unlock(l);
}

// unlocked

// locked or unlocked

// locked or unlocked

// locked or unlocked
```
Why This Design?

• Simple, efficient, scalable alias analysis

• More abstract locations =
  - Greater precision
  - Less efficiency
    • Need to model facts about more names at each state

• Observation: A little extra local information would make a big difference
Restrict

```c
void do_with_lock(lock *restrict l) { ... }
```

- Let `l` point to lock `a`
- Within `l`'s scope, all accesses to `a` are through `l`
  - (Approximately)

- In `do_with_lock`, no other aliases of `a` used
  - Can use `strong updates` on state of `l`
Contributions of This Work

• Type and effect system for checking that uses of `restrict` are safe
  - Provably sound
  - Contrast to ANSI C: restrict annotation trusted
    • unchecked, with informal semantics

• New construct `confine`
  - Short-hand for common uses of restrict
  - Easier to use
Contributions of This Work (cont'd)

• Automatic inference of `restrict` and `confine`

• Experiment using `confine` inference in checking locking in Linux kernel
  - Recovered strong updates in 95% of cases
  - Enabled new deadlocks to come to light
Example from CQual

```c
void foo(int i) {
    do_with_lock(locks[i]);
}
void do_with_lock(lock *restrict l) {
    spin_lock(l);
    work();
    spin_unlock(l);
}
```
Example from CQual

```c
void foo(int i) {
    do_with_lock(locks[i]);
}
void do_with_lock(lock *restrict l) {
    spin_lock(l);
    work();
    spin_unlock(l);
}
```
Example from CQual

```c
void foo(int i) {
    do_with_lock(locks[i]);
}

void do_with_lock(lock *restrict l) {
    spin_lock(l);
    work();
    spin_unlock(l);
}
```
Example from CQual

```c
void foo(int i) {
    do_with_lock(locks[i]);
}
void do_with_lock(lock *restrict l) {
    spin_lock(l);
    work();
    spin_unlock(l);
}
```
Example from CQual

```c
void foo(int i) {
    do_with_lock(locks[i]);
}

void do_with_lock(lock *restrict l) {
    spin_lock(l);
    work();
    spin_unlock(l);
}
```

- `h` represents only one location
  - Safe to perform strong update (replacement)
Example from CQual

```c
void foo(int i) {
    do_with_lock(locks[i]);
}
void do_with_lock(lock * restrict l) {
    spin_lock(l);
    work();
    spin_unlock(l);
}
```

• ⬤ represents only one location
  - Safe to perform strong update (replacement)
Example from CQual

```c
void foo(int i) {
    do_with_lock(locks[i]);
}
void do_with_lock(lock *restrict l) {
    spin_lock(l);
    work();
    spin_unlock(l);
}
```

- □ represents only one location
  - Safe to perform strong update (replacement)
Example from CQual

```c
void foo(int i) {
    do_with_lock(locks[i]);
}
void do_with_lock(lock *restrict l) {
    spin_lock(l);
    work();
    spin_unlock(l);
}
```

- ℹ️ represents only one location
  - Safe to perform strong update (replacement)
Example from CQual

void foo(int i) {
    do_with_lock(locks[i]);
}
void do_with_lock(lock *restrict l) {
    spin_lock(l);
    work();
    spin_unlock(l);
}

- □ represents only one location
  - Safe to perform strong update (replacement)
Check Restrict with Type and Effect System

• Types extended with *abstract locations*
  - Flow-insensitive, unification-based may-alias analysis

\[ t ::= \ldots \mid \text{ref}\{\ldots\} \]
pointer to abstract loc \(
\)

• Effects are sets of locations

\[ L ::= \emptyset \mid \ldots \mid L_1 \| L_2 \mid L_1 \ll L_2 \]

Type Rules

- $A \mid e : \mathbb{V} \triangleright L$
  - In environment $A$, expression $e$ has type $\mathbb{V}$
  - evaluating $e$ has effect $L$

\[
A \mid e : \text{ref}(\mathbb{V}); L \\
A \mid ^* e : \mathbb{V} \triangleright L \triangleright \{\bullet\}
\]
Restrict

- restrict x = e1 in e2
  - x is a pointer initialized to e1
  - x is in scope only within e2
  - within e2, only x and copies derived from x can be used to access *x
  - outside of e2, values derived from x cannot be used
Restrict Rule

\[
A \mid e_1 : \text{ref}^{\top}(t_1); L_1 \quad A[\text{ref}^{\top}(t_1)\backslash x] \mid e_2 : t_2; L_2 \\
\downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \text{locs}(A, t_1, t_2)
\]

\[
A \mid \text{restrict } x = e_1 \text{ in } e_2 : t_2; L_1 \downarrow L_2 \downarrow \{\}
\]
Soundness

- Type system for checking \texttt{restrict} is sound
  - Well-typed program doesn’t go wrong according to a formal semantics
  - Uses of \texttt{restrict} are safe

- Provable using standard subject-reduction techniques
Improving Restrict

• Must bind a variable name in restrict

```c
spin_lock(locks[i]);
work();
spin_unlock(locks[i]);
```
Improving Restrict

• Must bind a variable name in `restrict`

```c
restrict mylock = locks[i] in {
    spin_lock(mylock)
    work();
    spin_unlock(mylock);
}
```

- Time consuming, need to manually check transformation is safe
Confine

- Short-hand for previous transformation

```c
confin e (locks[i]) in {
    spin_lock(locks[i])
    work();
    spin_unlock(locks[i]);
}
```

- Only need to pick expression, introduce scope
Confine Inference

• Assume we know expression to confine
  - In experiment, given spin_lock(e) try confining e

• Algorithm
  - Introduce confine everywhere
  - Eliminate incorrect confines
  - Greedily combine adjacent confines
    • (confine e in e1 ; confine e in e2) = confine e in e1;e2

• Use heuristics in practice to speed up
PLDI’02 Results

• Found a number of locking bugs in Linux kernel
  - Many weak updates with multi-file analysis
    • In CQual, weak updates to locks yield type errors
  - Makes it hard to find true locking bugs

• Can we eliminate weak updates with restrict?
  - Yes, but painful to put in by hand
New Experiment: Eliminating Weak Updates

• How many more strong updates with confine inference?

• Metric: # lock updates involved in error
  - Lower bound: Assume all updates are strong
    • But unsound!
Type Errors

Potential improvement

No type errors because of weak updates

352/60%

152/26%

85/14%

589 total modules

No type errors
Actual Improvement in Strong Updates

- 138/152: Same as assuming updates strong
  - Optimal result

- 14/152: Confine misses some strong updates
  - See paper for numbers
Experimental Summary

• Overall, confine inference gets 95% of cases
  - Could eliminate 3277 type errors
  - Does eliminate 3116 type errors
  - (Includes duplicates from duplicated modules)

• Remaining type errors
  - Deadlocks (found 4 new ones)
  - Aliasing conservatism and lack of path sensitivity
Conclusion

- **restrict** and **confine** successfully recover local strong updates
  - Can be used as programmer annotations to document aliasing properties

- **Automatic confine** inference works 95% of the time
  - Large improvement in quality of type errors