

# Flow-Sensitive Type Qualifiers

Jeffrey S. Foster, Tachio Terauchi, Alex Aiken

Presenter: Noam Zeilberger

Type Refinements Seminar

November 24, 2003

# Introduction

Standard type systems are flow-insensitive:

$$\dots \textcircled{1} x := e \textcircled{2} \dots$$

$x$ 's type is the same at  $\textcircled{1}$  and  $\textcircled{2}$ .

But many properties are flow-sensitive:

- File/network operations
- Locking mechanisms
- Error handling

Idea: keep types invariant while allowing type qualifiers to change.

# Introduction (cont.)

Three hurdles to constructing a flow-sensitive type system:

First, how do we represent flow-sensitive properties?

- Need a judgment  $\Gamma, C \vdash e : \tau, C'$ , with  $C$  and  $C'$  standing for pre- and post-stores.
- But cannot possibly represent an entire store.

Second, how do we deal with aliases?

- Soundness requires that aliased locations have the same type.
- Can only perform weak update if location is aliased.
  - strong:  $\llbracket loc(x) \rrbracket_{(2)} = \llbracket e \rrbracket$
  - weak:  $\llbracket loc(x) \rrbracket_{(2)} = \llbracket e \rrbracket \cup \llbracket loc(x) \rrbracket_{(1)}$

# Introduction (cont.)

Finally, how do we *deal* with aliases?

- Tradeoff between aliasing and strong updates is onerous to programmer.
- We need a way to recover strong updates.

*Flow-Sensitive Type Qualifiers* attempts to address all of these issues:

- Stores represented concisely by constraints.
- Alias analysis and linearity computations infer admissibility of strong/weak updates.
- New syntactic construct `restrict` (inspired by C99, similar to, but different than `focus`).

# C stream bugs

```
void log_msg(const char *log_file, const char *msg) {
    FILE *log = fopen(log_file, "a");
    fprintf(log, "%s\n", msg); /* bug: log may be NULL */
    fclose(log);
}

void increment(const char *file) {
    FILE *fp;
    if ((fp = fopen("r")) == NULL) { die("error opening %s", file); }
    else {
        int d;
        fscanf(fp, "%d", &d);
        d++;
        fprintf(fp, "%d", &d); /* bug: fp not open for write */
        fclose(fp);
    }
}
```

# Locking bugs

```
void f(struct obj *o) {
    acquire_lock(&o->lock);
    do_stuff();
    g(o);
    release_lock(&o->lock);
}
```

```
void g(struct obj *o) {
    if (test(o)) {
        acquire_lock(&o->lock); /* bug: deadlock */
        do_stuff();
        release_lock(&o->lock);
    }
}
```

# Linearity and restrict

```
void h(struct obj **os) {
    struct obj *o;

    for (o=&os[0]; *o!=NULL; o++) {
        acquire_lock(&o->lock); /* error: &o->lock non-linear */
        do_stuff();
        release_lock(&o->lock);
    }

    for (o=&os[0]; *o!=NULL; o++) {
        restrict lock = &o->lock in {
            acquire_lock(lock); /* okay */
            do_stuff();
            release_lock(lock);
        }
    }
}
```

# Source language

Basically same as before:

$$\begin{aligned} e & ::= x \mid n \mid \lambda x.e \mid e_1 e_2 \\ & \mid \text{ref } e \mid !e \mid e_1 := e_2 \\ & \mid \text{annot}(e, Q) \\ & \mid \text{check}(e, Q) \end{aligned}$$

Flow-insensitive alias analysis and effect inference are performed by decorating the source language with abstract locations, types, and effects.

# Target language

$$\begin{aligned} e & ::= x \mid n \mid \lambda^L x : t.e \mid e_1 e_2 \\ & \mid \text{ref}^\rho e \mid !e \mid e_1 := e_2 \\ & \mid \text{annot}(e, Q) \\ & \mid \text{check}(e, Q) \end{aligned}$$
$$\begin{aligned} t & ::= \alpha \mid \text{int} \\ & \mid \text{ref}(\rho) \\ & \mid t \longrightarrow^L t' \end{aligned}$$
$$\begin{aligned} L & ::= \psi \\ & \mid \{\rho\} \\ & \mid L_1 \cup L_2 \mid L_1 \cap L_2 \end{aligned}$$

$\rho$       **abstract location**

**effect variable**  
**effect constant**

# Tranlation

We maintain  $C_I$ , a *global* mapping from abstract locations to types.

Rules for  $\boxed{\Gamma \vdash e \Rightarrow e' : t; L}$ :

$$\frac{x \in \text{dom}(\Gamma)}{\Gamma \vdash x \Rightarrow x : \Gamma(x); \emptyset} \quad (\text{Var})$$

$$\frac{}{\Gamma \vdash n \Rightarrow n : \text{int}; \emptyset} \quad (\text{Int})$$

# Translation (cont.)

$$\frac{\Gamma \vdash e \Rightarrow e' : t; L \quad C_I(\rho) = t \quad \rho \text{ fresh}}{\Gamma \vdash \mathbf{ref} \ e \Rightarrow \mathbf{ref}^\rho \ e' : \mathit{ref}(\rho); L \cup \{\rho\}} \quad \text{(Ref)}$$

$$\frac{\Gamma \vdash e \Rightarrow e' : t; L \quad t = \mathit{ref}(\rho) \quad \rho \text{ fresh}}{\Gamma \vdash !e \Rightarrow !e' : C_I(\rho); L \cup \{\rho\}} \quad \text{(Deref)}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow e'_1 : t_1; L_1 \quad \Gamma \vdash e_2 \Rightarrow e'_2 : t_2; L_2 \quad t_1 = \mathit{ref}(\rho) \quad C_I(\rho) = t_2 \quad \rho \text{ fresh}}{\Gamma \vdash e_1 := e_2 \Rightarrow e'_1 := e'_2 : t_2; L_1 \cup L_2 \cup \{\rho\}} \quad \text{(Assign)}$$

# Translation (cont.)

$$\frac{\Gamma[x \mapsto \alpha] \vdash e \Rightarrow e' : t; L \quad L \subseteq \psi \quad \alpha, \psi \text{ fresh}}{\Gamma \vdash \lambda x. e \Rightarrow \lambda^\psi x : \alpha. e' : \alpha \longrightarrow^\psi t; \emptyset} \quad (\text{Lam})$$

$$\frac{\Gamma \vdash e_1 \Rightarrow e'_1 : t_1; L_1 \quad \Gamma \vdash e_2 \Rightarrow e'_2 : t_2; L_2 \quad t_1 = t_2 \longrightarrow^\psi \beta \quad \psi, \beta \text{ fresh}}{\Gamma \vdash e_1 e_2 \Rightarrow e'_1 e'_2 : \beta; L_1 \cup L_2 \cup \psi} \quad (\text{App})$$

$$\frac{\Gamma \vdash e \Rightarrow e' : t; L}{\Gamma \vdash \text{annot}(e, Q) \Rightarrow \text{check}(e', Q) : t; L} \quad (\text{Annot})$$

$$\frac{\Gamma \vdash e \Rightarrow e' : t; L}{\Gamma \vdash \text{check}(e, Q) \Rightarrow \text{check}(e', Q) : t; L} \quad (\text{Check})$$

# Translation (cont.)

Effects on purely local state need not be exposed outside a function's scope. Hence this rule:

$$\frac{\Gamma \vdash e \Rightarrow e' : t; L}{\Gamma \vdash e \Rightarrow e' : t; L \cap (\text{locs}(\Gamma) \cup \text{locs}(t))} \quad (\text{Down})$$

where

$$\begin{aligned} \text{locs}(\text{int}) &= \emptyset \\ \text{locs}(\text{ref}(\rho)) &= \{\rho\} \cup \text{locs}(C_I(\rho)) \\ \text{locs}(t_1 \xrightarrow{L} t_2) &= L \cup \text{locs}(t_1) \cup \text{locs}(t_2) \end{aligned}$$

and  $\text{locs}(\Gamma) = \bigcup_{x \mapsto t \in \Gamma} \text{locs}(t)$

(Down) is non-syntactic, but need only be applied once per function.

# Example

```
fun f w =  
  let x = ref 0  
      y = ref (annot(1, qa))  
      z = ref (annot(2, qb))  
  in  
    x := 3;  
    w := 4;  
    y := annot(5, qc);  
    f z;  
    check(!y, qc)
```

# Example translation

```
fun{ $\rho_z$ } f w : ref( $\rho_z$ ) =  
  let x = ref $\rho_x$  0  
      y = ref $\rho_y$  (annot(1,  $q_a$ ))  
      z = ref $\rho_z$  (annot(2,  $q_b$ ))  
  in  
    x := 3;  
    w := 4;  
    y := annot(5,  $q_c$ );  
    f z;  
    check(!y,  $q_c$ )
```

$$C_I(\rho_x) = C_I(\rho_y) = C_I(\rho_z) = \text{int}$$

# Flow-sensitive qualifiers

Building upon flow-insensitive type-, alias-, and effect-inference, flow-sensitive analysis associates a store  $C$  with each program point and computes qualified types.

$$\tau ::= Q \sigma$$

$$Q ::= \kappa \mid B$$

$$\sigma ::= \alpha \mid int$$

$$\mid ref(\rho)$$

$$\mid (C, \tau) \longrightarrow^L (C', \tau')$$

# Representing stores

A *ground store*  $G$  is a vector  $\{\rho_1^{\eta_1} : \tau_1, \dots, \rho_n^{\eta_n} : \tau_n\}$ , associating (qualified) types  $\tau_i$  and linearities  $\eta_i$  with each abstract location  $\rho_i$ .

$$G(\rho_i) = \tau_i, G(\rho_i)_{lin} = \eta_i$$

where the linearities are taken from a lattice  $0 < 1 < \omega$ , with  $0 + x = x$ ,  $1 + 1 = \omega$ ,  $\omega + x = \omega$

Stores are represented through a constraint formalism

- $\varepsilon$  represents an unknown store.
- Store constructors and constraints relate stores at consecutive points.
- A *solution*  $S$  maps store variables to ground stores.

# Representing stores (cont.)

Store constraint  $C_1 \leq C_2$ :

$$\frac{\tau_i \leq \tau'_i \quad \eta_i \leq \eta'_i \quad i = 1..n}{\{\rho_1^{\eta_1} : \tau_1, \dots, \rho_n^{\eta_n} : \tau_n\} \leq \{\rho_1^{\eta'_1} : \tau'_1, \dots, \rho_n^{\eta'_n} : \tau'_n\}} \quad (\text{Store}_{\leq})$$

Store constructors:

$$\begin{array}{ll} \text{Alloc}(C, \rho) & \text{Assign}(C, \rho : \tau) \\ \text{Merge}(C, C', L) & \text{Filter}(C, L) \end{array}$$

A solution  $S$  satisfies a system of store constraints if  $S(C_1) \leq S(C_2)$  for each  $C_1 \leq C_2$ , and  $S$  respects the store constructor rules...

# Store constructors

$$S(\text{Alloc}(C, \rho'))(\rho) = S(C)(\rho)$$

$$S(\text{Alloc}(C, \rho'))_{lin}(\rho) = \begin{cases} 1 + S(C)_{lin}(\rho) & \rho = \rho' \\ S(C)_{lin}(\rho) & \text{otherwise} \end{cases}$$

$$S(\text{Merge}(C, C', L))(\rho) = \begin{cases} S(C)(\rho) & \rho \in L \\ S(C')(\rho) & \text{otherwise} \end{cases}$$

$$S(\text{Merge}(C, C', L))_{lin}(\rho) = \begin{cases} S(C)_{lin}(\rho) & \rho \in L \\ S(C')_{lin}(\rho) & \text{otherwise} \end{cases}$$

# Store constructors (cont.)

$$S(\text{Filter}(C, L))(\rho) = S(C)(\rho) \quad \rho \in L$$

$$S(\text{Filter}(C, L))_{lin}(\rho) = \begin{cases} S(C)_{lin}(\rho) & \rho \in L \\ 0 & \text{otherwise} \end{cases}$$

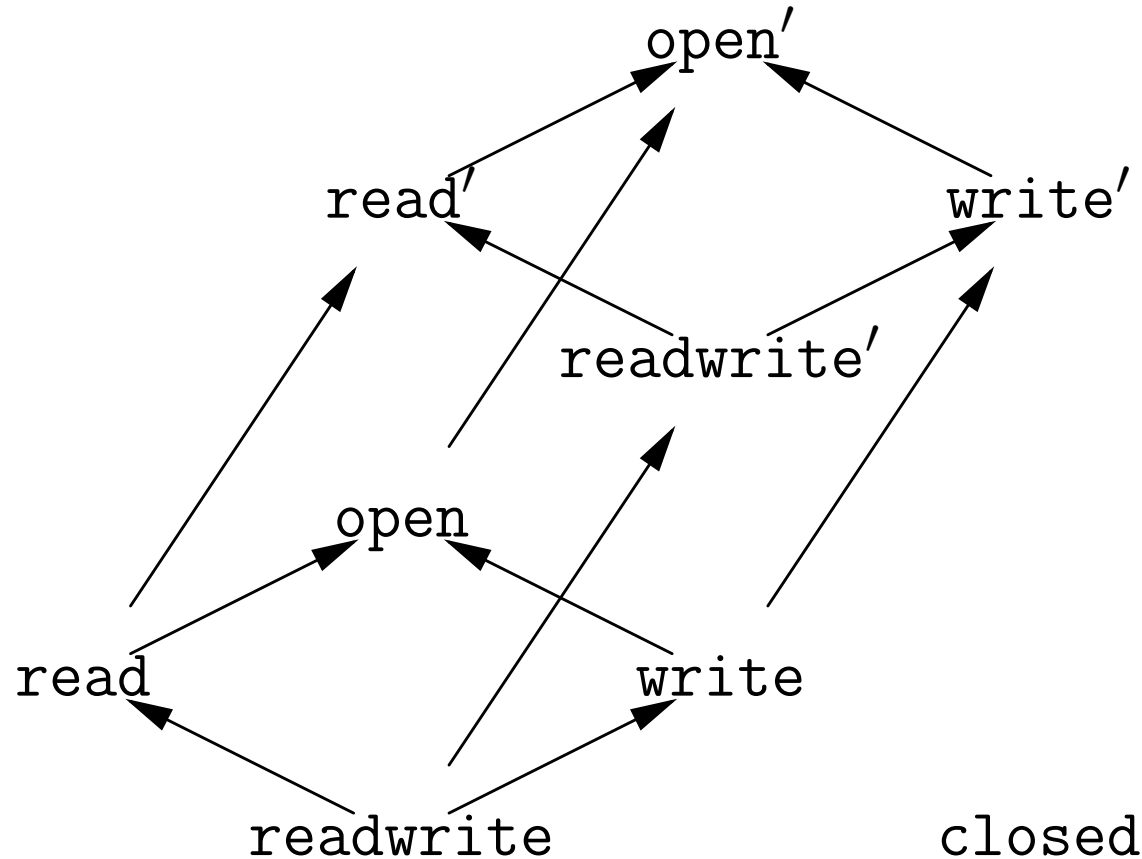
$$S(\text{Assign}(C, \rho' : \tau))(\rho) = \begin{cases} \tau' \text{ where } \tau \leq \tau' & \rho = \rho' \\ S(C)(\rho) & \text{otherwise} \end{cases}$$

$$S(\text{Assign}(C, \rho' : \tau))_{lin}(\rho) = S(C)_{lin}(\rho)$$

Weak updates:

$$S(C)_{lin}(\rho) = \omega \implies S(C)(\rho) \leq S(\text{Assign}(C, \rho : \tau))(\rho)$$

# Example: C streams



$\text{fclose} : (C, \text{ref}(\rho)) \longrightarrow^{\rho} (\text{Assign}(C, \rho : \text{closed } \tau), \text{int})$   
where  $C(\rho) \leq \text{open } \tau$

$\text{fopen} : (C, \text{string} \times \text{mode}) \longrightarrow^{\rho} (\text{Assign}(\text{Alloc}(C, \rho), \rho : \text{mode } \tau), \text{ref}(\rho))$   
where  $C(\rho) \leq \text{closed } \tau$

# Constraint generation

Again use spread operation  $sp(t)$ :

$$sp(\alpha) = \kappa \alpha$$

$$sp(int) = \kappa int$$

$$sp(ref(\rho)) = \kappa ref(\rho)$$

$$sp(t \longrightarrow^L t') = \kappa (\varepsilon, sp(t)) \longrightarrow^L (\varepsilon', sp(t'))$$

Judgment  $\boxed{\Gamma, C \vdash e : \tau, C'}$ :

$$\frac{x \in dom(\Gamma)}{\Gamma, C \vdash x : \Gamma(x) : C} \quad (\text{Var})$$

$$\frac{\kappa \text{ fresh}}{\Gamma, C \vdash n : \kappa int : C} \quad (\text{Int})$$

# Constraint generation (cont.)

$$\frac{\Gamma, C \vdash e : \tau, C' \quad \tau \leq C'(\rho) \quad \kappa \text{ fresh}}{\Gamma, C \vdash \mathbf{ref}^\rho e : \kappa \mathit{ref}(\rho), \mathit{Alloc}(C', \rho)} \quad (\mathbf{Ref})$$

$$\frac{\Gamma, C \vdash e : Q \mathit{ref}(\rho), C'}{\Gamma, C \vdash !e : C'(\rho), C'} \quad (\mathbf{Deref})$$

$$\frac{\Gamma, C \vdash e_1 : Q \mathit{ref}(\rho), C' \quad \Gamma, C' \vdash e_2 : \tau, C''}{\Gamma, C \vdash e_1 := e_2 : \tau, \mathit{Assign}(C'', \rho : \tau)} \quad (\mathbf{Assign})$$

# Constraint generation (cont.)

$$\frac{\begin{array}{l} \tau = sp(t) \quad \varepsilon, \varepsilon', \kappa \text{ fresh} \\ \Gamma[x \mapsto \tau], \varepsilon \vdash e : \tau', C' \leq \varepsilon' \end{array}}{\Gamma, C \vdash \lambda^L x : t.e : \kappa (\varepsilon, \tau) \longrightarrow^L (\varepsilon', \tau'), C} \quad (\text{Lam})$$

$$\frac{\begin{array}{l} \Gamma, C \vdash e_1 : Q (\varepsilon, \tau) \longrightarrow^L (\varepsilon', \tau'), C' \quad \Gamma, C' \vdash e_2 : \tau_2, C'' \\ \tau_2 \leq \tau \quad Filter(C'', L) \leq \varepsilon \end{array}}{\Gamma, C \vdash e_1 e_2 : \tau', Merge(\varepsilon', C'', L)} \quad (\text{App})$$

$$\frac{\Gamma, C \vdash e : Q' \sigma, C' \quad Q' \sqsubseteq Q}{\Gamma, C \vdash \text{annot}(e, Q) : Q \sigma, C'} \quad (\text{Annot})$$

$$\frac{\Gamma, C \vdash e : Q' \sigma, C' \quad Q' \sqsubseteq Q}{\Gamma, C \vdash \text{check}(e, Q) : Q' \sigma, C'} \quad (\text{Check})$$

# Constraint resolution

Linearities  $S(C)_{lin}(\rho)$  can be determined with fixpoint computation. (Actually found by computing cycles.)

Qualified types  $S(C)(\rho)$  can then be determined by iterating store constructor and constraint generation rules, beginning with  $S$  mapping every  $\varepsilon$  to

$$\{\rho_1 : sp(C_I(\rho_1)), \dots, \rho_n : sp(C_I(\rho_n))\}$$

This is  $O(n^2)$  space/time. It can be reduced by noting:

- Many locations can be flow-insensitive—keep them only in global store.
- Not every store needs every location—add a location  $\rho$  to  $S(\varepsilon)$  only after  $\varepsilon(\rho)$  is requested, and there is a  $C \leq \varepsilon$  or  $\varepsilon \leq C$  such that  $\rho \in S(C)$ .

# restrict

Recall the linearity problem:

```
acquire_lock(&o->lock); /* error: &o->lock non-linear */
do_stuff();
release_lock(&o->lock);
```

We cannot update `o->lock`'s qualifier from `unlocked` to `locked` because `&o->lock` is non-linear.

```
restrict lock = &o->lock in {
    acquire_lock(lock); /* okay */
    do_stuff();
    release_lock(lock);
}
```

Inside the `restrict` block, we can strongly update `lock`. After it ends, we perform a weak update from `lock` to `&o->lock`.

# restrict: formal definition

$e ::= \dots \mid \text{restrict } x = e_1 \text{ in } e_2$

Alias and effect translation:

$$\frac{\begin{array}{l} \Gamma \vdash e_1 \Rightarrow e'_1 : t_1; L_1 \quad t_1 = \text{ref}(\rho) \quad \rho, \rho' \text{ fresh} \\ C_I(\rho') = C_I(\rho) \quad \Gamma[x \mapsto \text{ref}(\rho')] \vdash e_2 \Rightarrow e'_2 : t_2 : L_2 \\ \rho \notin L_2 \quad \rho' \notin \text{locs}(\Gamma) \cup \text{locs}(C_I(\rho)) \cup \text{locs}(t_2) \end{array}}{\Gamma \vdash \text{restrict } x = e_1 \text{ in } e_2 \Rightarrow} \\ \text{restrict}^{\rho'} x = e'_1 \text{ in } e'_2 : t_2; L_1 \cup L_2 \cup \{\rho\}$$

Constraint generation:

$$\frac{\begin{array}{l} \Gamma, C \vdash e_1 : Q \text{ ref}(\rho), C' \\ C'' = \text{Alloc}(C', \rho') \quad C'(\rho) \leq C''(\rho') \\ \Gamma[x \mapsto \text{ref}(\rho')], C'' \vdash e_2 : \tau_2, C''' \end{array}}{\Gamma, C \vdash \text{restrict}^{\rho'} x = e_1 \text{ in } e_2 : \tau_2, \text{Assign}(C''', \rho : C'''(\rho'))}$$

# Conclusions

Flow-sensitive type qualifiers are a useful tool for specifying and verifying protocol abstractions.

- Enable analysis of existing C programs with little annotation (CQUAL).
- Formalize and simplify previously ad hoc semantics.
- Strong updates can often be recovered using `restrict`.

But they as yet suffer from serious limitations:

- Highly dependent on (limited) flow analysis.
- No (good) syntax for specifying function signatures.
- Constraint-based type inference makes sources of type-errors difficult to locate (though this is ameliorated by the user interface).

# CQUAL

Insert demo here.