

# A Theory of Type Qualifiers

Jeffrey S. Foster, Manuel Fähndrich, Alex Aiken

Presenter: Noam Zeilberger

Type Refinements Seminar

November 24, 2003

# A brief history of qualifiers

- `const`: read-only references
  - C++ → C89, Java (`final`)
- `volatile`: asynchronously modified values
  - C89 → C++, Java
- `noalias`: “alias-free objects”
  - proposed for C ... abandoned  
(Dennis Ritchie, “Why I do not like X3J11 type qualifiers”)
- `restrict`: non-aliased pointers
  - Fortran → C99

# Type qualifier literature

## Flow-insensitive

- Foster, Fähndrich, Aiken: *A Theory of Type Qualifiers*, 1999
- Shankar, Talwar, Foster, Wagner: *Detecting Format-String Vulnerabilities with Type Qualifiers*, 2001

## Flow-sensitive

- Foster, Terauchi, Aiken: *Flow-Sensitive Type Qualifiers*, 2002
- Aiken, Foster, Kodumal, Terauchi: *Checking and Inferring Local Non-Aliasing*, 2003

Also Foster's Ph.D. thesis (2002)

# Flow-insensitive qualifiers

Positive/negative qualifier defines a two-point lattice  $L_q$ :

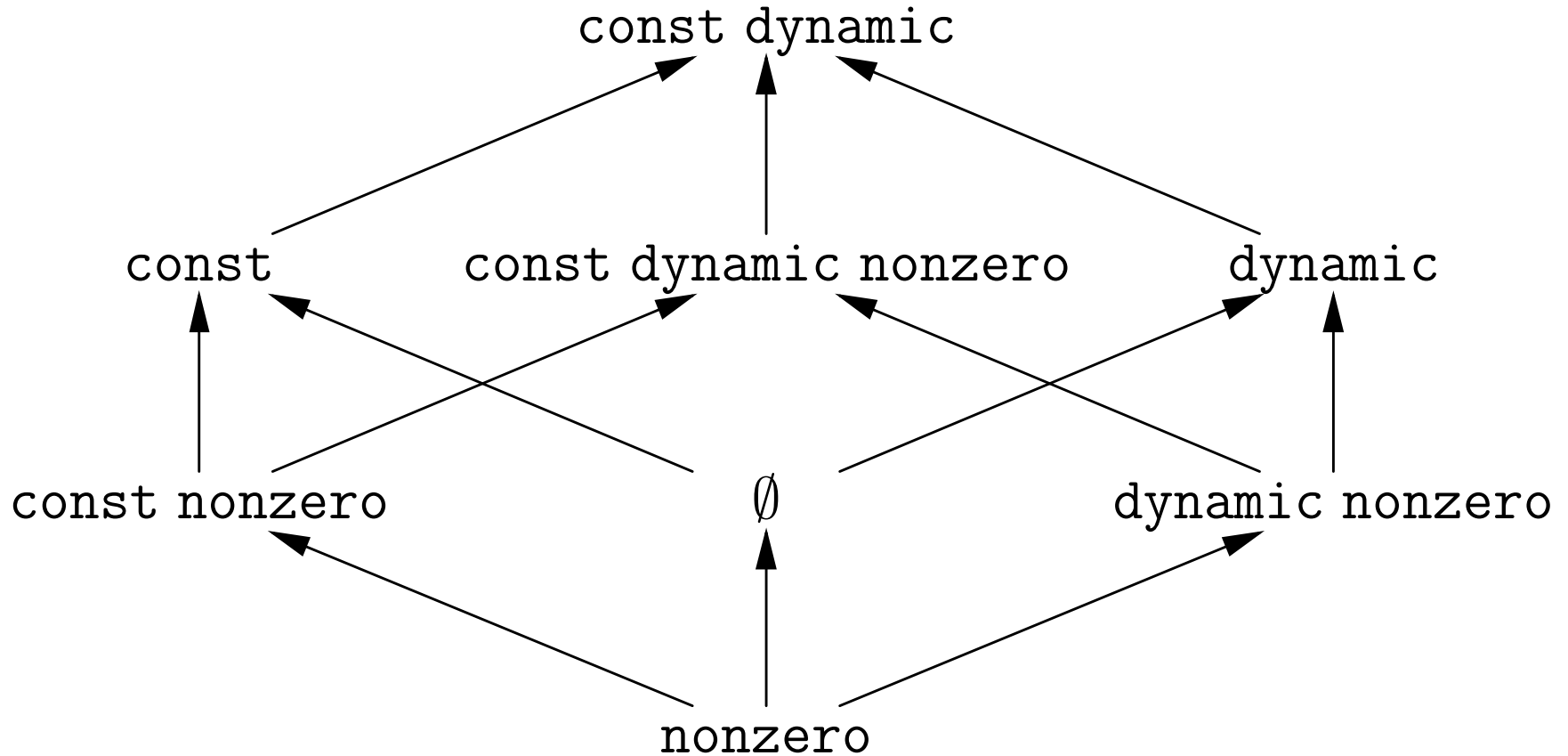
- $+$  :  $L_q = \perp_q \sqsubseteq q$
- $-$  :  $L_q = q \sqsubseteq \top_q$

e.g.,  $+$ : const, dynamic ( $\sqsubseteq$  static),  $-$ : nonzero

Qualifier lattice is a product  $L = \prod_q L_q$

- a.k.a. bit vector/powerset/hypercube lattice.
- Inessential restriction? (It's gone in future work.)
- The qualifier lattice induces a subtyping relationship on qualified types.

# Example lattice



$\perp_{\text{const}} \sqsubseteq \text{const}$        $\perp_{\text{dynamic}} \sqsubseteq \text{dynamic}$        $\text{nonzero} \sqsubseteq \top_{\text{nonzero}}$

e.g.,

`const nonzero ref(dynamic int) ≤ const ref(dynamic int)`

# Source language

$$e ::= x \mid n \mid \lambda x.e \mid e_1 e_2$$
$$\mid \text{let } x = e_1 \text{ in } e_2$$
$$\mid \text{ref } e \mid !e \mid e_1 := e_2 \mid ()$$
$$\mid \text{annot}(e, Q) \quad \text{qualifier annotation}$$
$$\mid \text{check}(e, Q) \quad \text{qualifier check}$$

(Beware qualifier “assertion”!)

# Types with qualifiers

$\tau ::= Q \sigma$       qualified type

$\sigma ::= \alpha$   
|  $int$   
|  $\tau_1 \rightarrow \tau_2$   
|  $ref(\tau)$   
|  $unit$

$Q ::= \kappa$       qualifier variable  
|  $B$       constant qualifier (lattice element)

# Subtyping

General case (arbitrary type constructor  $c$ ):

$$\frac{Q \sqsubseteq Q' \quad \tau_i = \tau'_i \quad i \in [1..n]}{Q \ c(\tau_1, \dots, \tau_n) \leq Q' \ c(\tau'_1, \dots, \tau'_n)}$$

Subtyping rules:

$$\frac{Q \sqsubseteq Q'}{Q \ \text{int} \leq Q' \ \text{int}} \quad (\text{SubInt}) \qquad \frac{Q \sqsubseteq Q'}{Q \ \text{unit} \leq Q' \ \text{unit}} \quad (\text{SubUnit})$$

$$\frac{Q \sqsubseteq Q' \quad \tau'_1 \leq \tau_1 \quad \tau_2 \leq \tau'_2}{Q \ (\tau_1 \rightarrow \tau_2) \leq Q' \ (\tau'_1 \rightarrow \tau'_2)} \quad (\text{SubFun})$$

$$\frac{Q \sqsubseteq Q' \quad \tau = \tau'}{Q \ \text{ref}(\tau) \leq Q' \ \text{ref}(\tau')} \quad (\text{SubRef})$$

# Type checking

$$\frac{\Gamma \vdash e : \tau \quad \tau \leq \tau'}{\Gamma \vdash e : \tau'} \quad (\text{Sub})$$

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

$$\frac{\Gamma \vdash e : Q t \quad Q \sqsubseteq Q'}{\Gamma \vdash \text{annot}(e, Q') : Q' t} \quad (\text{Annot})$$

# Type checking (cont.)

$$\frac{}{\Gamma \vdash n : \perp \text{ int}} \quad (\text{Int})$$

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

$$\frac{\Gamma[x \mapsto \tau_x] \vdash e : \tau}{\Gamma \vdash \lambda x. e : \perp (\tau_x \rightarrow \tau)} \quad (\text{Lam})$$

$$\frac{\Gamma \vdash e_1 : Q (\tau_2 \rightarrow \tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau} \quad (\text{App})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2} \quad (\text{Let})$$

# Type checking (cont.)

$$\frac{}{\Gamma \vdash () : \perp \textit{unit}} \quad \text{(Unit)}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \textit{ref } e : \perp \textit{ref}(\tau)} \quad \text{(Ref)}$$

$$\frac{\Gamma \vdash e : Q \textit{ref}(\tau)}{\Gamma \vdash !e : \tau} \quad \text{(Deref)}$$

$$\frac{\Gamma \vdash e_1 : Q \textit{ref}(\tau_2) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 := e_2 : \perp \textit{unit}} \quad \text{(Assign)}$$

# const

The rule for assignment ignores  $e_1$ 's qualifier—how can we incorporate `const` into the framework?

Two possibilities:

- **programmer** replaces  $e_1 := e_2$  by  $\text{check}(e_1, \neg\text{const}) := e_2$
- **language designer** creates new type checking rule

$$\frac{\Gamma \vdash e_1 : \neg\text{const } \text{ref}(\tau_2) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 := e_2 : \perp \text{ unit}} \quad (\text{Assign}')$$

The latter approach is simpler in this case, but the former has many useful applications...

# Diversion: C idiosyncracies

In C, a variable can denote a l-value or an r-value. We interpret a declaration “ $T x;$ ” as “ $x : ref(T)$ ” and an appearance of “ $x$ ” on the right as “ $!x$ ”.

Question: should we interpret “ $Q T x;$ ” as “ $x : ref(Q T)$ ” or “ $x : Q ref(T)$ ”? Ad-hoc answer:

$const T x;$	$\Rightarrow$	$x : const ref(T)$
$nonzero T x;$	$\Rightarrow$	$x : ref(nonzero T)$
		$\vdots$

Qualifier designer must specify whether qualifier constrains “ $ref$  level” or “value level”.

**Note:**  $T * nonzero x; \Rightarrow x : ref(nonzero ref(T))$

# Annotate and check

Qualifier annotations have no formal semantic meaning, but enforce data abstractions.

- **format-string vulnerabilities**—`untainted`  $\sqsubseteq$  `tainted`:

```
int printf(untainted const char *fmt, ...);
tainted char *getenv(const char *name);
```

- **Y2K bugs**—positive qualifiers `YY` and `YYYY`:

```
void pr_year(char *YY year) {
    printf("The year is 19%s", year);
}
void f() {
    pr_year((char *YY)"99");
    pr_year((char *YYYY)"2000");
}
```

# Qualifiers and effects

sorted qualifier? Perhaps:

*isort* : unsorted *iarray*  $\rightarrow$  sorted *iarray*

*merge* : sorted *iarray*  $\times$  sorted *iarray*  $\rightarrow$  sorted *iarray*

But we want to sort in place! From `stdlib.h`:

```
void qsort(void *base, size_t nmem, size_t size,  
           int (*compar)(const void *, const void *));
```

We need “flow-sensitive” qualifiers to annotate `qsort...`

# Type inference with constraints

Typing judgment  $\Gamma \vdash e : \tau; C$

Constraints contain subtyping and lattice inequalities:

$$C ::= \{\tau_1 \leq \tau_2\} \mid \{Q_1 \sqsubseteq Q_2\} \mid C_1 \cup C_2$$

Spread operation  $\text{sp} : Typ \rightarrow QTyp$  constructs most general qualified types:

$$\begin{aligned} \text{sp}(\alpha) &= \kappa \alpha \\ \text{sp}(int) &= \kappa int \\ \text{sp}(t_1 \rightarrow t_2) &= \kappa (\text{sp}(t_1) \rightarrow \text{sp}(t_2)) \\ &\vdots \end{aligned}$$

# Type inference with constraints (cont.)

Checking rules become constrained inference rules via a standard HM(X) transformation, e.g.:

$$\frac{\Gamma \vdash e_1 : \tau_1; C_1 \quad \Gamma \vdash e_2 : \tau_2; C_2 \quad C = C_1 \cup C_2 \cup \{\tau_1 = \kappa (\tau_2 \rightarrow \kappa' \alpha)\}}{\Gamma \vdash e_1 e_2 : \kappa' \alpha; C} \quad (\text{App})$$

Applying subtyping rules transforms subtyping constraints into sets of lattice constraints... which can be solved efficiently.

# Two uses of strchr

```
/* checks if s ends in a single '\n' */
int ends_in_eol(const char *s) {
    const char *p = strchr(s, '\n');
    return (p && *(p+1) == '\0');
}
```

```
/* remove '\n' and trailing characters of s */
void chop(char *s) {
    char *p = strchr(s, '\n');
    *p = '\0';
}
```

(`strchr(s, c)` returns a pointer to the first occurrence of `c` in `s`.)

# What is `strchr`'s type?

`const ref(char) × int → const ref(char)` or  
`ref(char) × int → ref(char)`? These are incomparable.

C library compromise:

```
char *strchr(const char *s, int c);
```

(How do you implement this?)

This is an old problem—Ritchie describes it in 1988 and cites it as already known for several years.

Pernicious effect on C++ Standard Template library...

# C++ eyestrain

from stl\_tree.h:

```
template <class _Key, class _Value, class _KeyOfValue,
          class _Compare, class _Alloc>
typename _Rb_tree<_Key, _Value, _KeyOfValue, _Compare, _Alloc>::rb_tree_iterator
_Rb_tree<_Key, _Value, _KeyOfValue, _Compare, _Alloc>::find(const _Key& __k) const
{
    _Link_type __y = _M_header; /* Last node which is not less than */k.
    _Link_type __x = _M_root(); /* Current node.
    while (__x != 0) {
        if (!_M_key_compare(_S_key(__x), __k))
            __y = __x, __x = _S_left(__x);
        else
            __x = _S_right(__x);
    }
    return iterator const iterator(__y);
    return (__j == end() || !_M_key_compare(__k, _S_key(__j._M_node))) ?
        end() : __j;
}
```

# Qualifier polymorphism

Conclusion: we need polymorphism over type qualifiers.

Qualifier schemes  $\forall \vec{k}. \tau \setminus C$

New typing rules:

$$\frac{\Gamma \vdash v : \tau_1; C_1 \quad \Gamma, x : \forall \vec{k}. \tau_1 \setminus C \vdash e_2 : \tau_2; C_2 \quad \vec{k} \text{ not free in } \Gamma}{\Gamma \vdash \text{let } x = v \text{ in } e_2 : \tau_2; (\exists \vec{k}. C_1) \cup C_2} \quad (\text{Let})$$

$$\frac{\Gamma(x) = \forall \vec{k}. \tau \setminus C}{\Gamma \vdash x : \tau[\vec{k} \mapsto \vec{Q}]; C[\vec{k} \mapsto \vec{Q}]} \quad (\forall\text{-elim})$$

# Qualifiers vs. refinements

The lattice-based subtyping framework for qualifiers bears some superficial similarities to datasort refinements, but there are crucial differences:

- Qualifiers inject values (of arbitrary type) to qualified values, whereas datasorts define (regular tree) subsets of values of a given type.
- Hence qualifiers catch data-abstraction violations, while refinements catch data-representation violations.
- Not all qualified types are semantically significant. (Is overloading desirable?)
- Lack of intersection makes inferred types more tractable... which enables analysis of C programs with little annotation.