A Theory of Type Qualifiers*

Jeffrey S. Foster
jfoster@cs.berkeley.edu

Manuel Fähndrich
maf@microsoft.com

Alexander Aiken
aiken@cs.berkeley.edu

EECS Department
University of California, Berkeley
Berkeley, CA 94720-1776

Abstract

We describe a framework for adding type qualifiers to a language. Type qualifiers encode a simple but highly useful form of subtyping. Our framework extends standard type rules to model the flow of qualifiers through a program, where each qualifier or set of qualifiers comes with additional rules that capture its semantics. Our framework allows types to be polymorphic in the type qualifiers. We present a `const`-inference system for C as an example application of the framework. We show that for a set of real C programs, many more `const`s can be used than are actually present in the original code.

1 Introduction

Programmers know strong invariants about their programs, and it is widely accepted by practitioners that such invariants should be automatically, statically checked to the extent possible [Mag93]. However, except for static type systems, modern programming languages provide little or no support for expressing such invariants. In our view, the problem is not a lack of proposals for expressing invariants; the research community, and especially the verification community, has proposed many mechanisms for specifying and proving properties of programs. Rather, the problem lies in gaining widespread acceptance in practice. A central issue is what sort of invariants programmers would be willing to write down.

In this paper we consciously seek a conservative framework that minimizes the unfamiliar machinery programmers must learn while still allowing interesting program invariants to be expressed and checked. One kind of programming annotation that is widely used is a *type qualifier.*

Type qualifiers are easy to understand, yet they can express strong invariants. The type system guarantees that in every program execution the semantic properties captured by the qualifier annotations are maintained. This is in contrast to dynamic invariant checking (e.g., `assert` macros or Purify [Pur]), which test for properties in a particular program execution.

A canonical example of a type qualifier from the C world is the ANSI C qualifier `const`. A variable with a type annotated with `const` can be initialized but not updated.[1] A primary use of `const` is annotating pointer-valued function parameters as not being updated by the function. Not only is this information useful to a caller of the function, but it is automatically verified by the compiler (up to casting).

Another example is Evans's `lclint` [Eva96], which introduces a large number of additional qualifier-like annotations to C as an aid to debugging memory usage errors. One such annotation is `nonnull`, which indicates that a pointer value must not be null. Evans found that adding such annotations greatly increased compile-time detection of null pointer dereferences [Eva96]. Although it is not a type-based system, we believe that annotations like `lclint`'s can be expressed naturally as type qualifiers in our framework.

Yet another example of type qualifiers comes from binding-time analysis, which is used in partial evaluation systems [Hen91, DHM95]. Binding-time analysis infers whether values are known at compile time (the qualifier `static`) or may not be known until run time (the qualifier `dynamic`) by specializing the program with respect to an initial input.

There are also many other examples of type qualifiers in the literature. Each of the cited examples adds particular type qualifiers for a specific application. This paper presents a framework for adding new, user-specified type qualifiers to a language in a general way. Our framework also extends the standard type system to perform *qualifier inference*, which propagates programmer-supplied annotations through the program and checks them. Such a system gives the programmer more complete information about qualifiers and makes qualifiers more convenient to use than a pure checking system.

The main contributions of the paper are

- We show that it is straightforward to parameterize a language by a set of type qualifiers and inference rules for checking conditions on those qualifiers. In particular, the changes to the lexing, parsing, and type checking (see below) phases of a compiler are minimal. We believe it would be realistic to incorporate our proposal into software engineering tools for any typed language.

*This research was supported in part by the National Science Foundation Young Investigator Award No. CCR-9457812, NASA Contract No. NAG2-1210, and an NDSEG fellowship.

To appear in the Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, May 1999.

[1] C allows type casts to remove `const`ness, but the result is implementation dependent [KR88].

- We show that the handling of type qualifiers can be separated from the standard type system of the language. That is, while the augmented type system includes rules for manipulating and checking qualifiers, in fact the computation of qualifiers can be isolated in a separate phase after standard typechecking has been performed. This factorization is similar to that of region inference [TT94].

- We introduce a natural notion of *qualifier polymorphism* that allows types to be polymorphic in their qualifiers. We present examples from existing C programs to show that qualifier polymorphism is useful and in fact necessary in some situations.

- We present experimental evidence from a prototype qualifier inference system. For this study, we examine the use of the qualifier `const` on a set of C benchmarks. We show that even in cases where programmers have apparently tried to systematically mark variables as `const`, monomorphic qualifier inference is able to infer many additional variables as `const`. Furthermore, polymorphic qualifier inference finds more `const` variables than monomorphic inference. This study shows both that qualifier inference is practical and useful, even for existing qualifiers and programs.

The technical observation behind our framework is that a type qualifier q introduces a simple form of subtyping: For all types τ, either $\tau \preceq q\ \tau$ or $q\ \tau \preceq \tau$. Here, as through the rest of the paper, we write qualifiers in prefix notation, so $q\ \tau$ represents standard type τ qualified by q. We illustrate the subtyping relationship using the examples given above. In C, non-`const` l-values can be promoted to `const` l-values, but not vice-versa. We capture this formally by saying that $\tau \preceq$ `const` τ for any type τ. In Evans's system, the set of non-null pointers is a subset of the set of all pointers, which is expressed as `nonnull` $\tau \preceq \tau$. In binding time analysis values may be promoted from `static` to `dynamic`. Since `static` and `dynamic` are dual notions, we can choose to write `static` $\tau \preceq \tau$ or $\tau \preceq$ `dynamic` τ, depending on which qualifier name we regard as the canonical one.

Our framework extends a language with a set of *standard types* and *standard type rules* to a *qualified type system* as follows.[2] The user defines a set of n type qualifiers $q_1, \ldots, q_n$ and indicates the subtyping relation for each (whether $q_i\ \tau \preceq \tau$ or $\tau \preceq q_i\ \tau$ for any standard type τ). Each level of a standard type may be annotated with a set of qualifiers, e.g., if $ref(int)$ is a standard type, then $q_1\ ref(q_2\ int)$ is a qualified type, where q_1 qualifies the *ref* and q_2 qualifies the *int*. We extend the standard type system to infer qualified types.

The polymorphic version of our system requires polymorphic *constrained types* to capture bounds on polymorphic qualifier variables. This form of polymorphic types involves only relatively simple constraints that can be solved with very efficient algorithms [HR97].

Each qualifier comes with rules that describe *well-formed types* and how qualifiers interact with the operations in the language. These rules are supplied by the user and may be nearly arbitrary (see Section 2.4). For example, a rule for `const` adds a qualifier test to require that the left-hand side of an assignment is non-`const`. An example of a well-formedness condition comes from binding time analysis: Nothing `dynamic` may appear within a value that is `static`. Thus, a type such as `static` (`dynamic` $\alpha \rightarrow$ `dynamic` β) is not well-formed.[3]

Because our framework is parameterized by the set of qualifiers, we must extend not only the types, but also the source language. We add both *qualifier annotations*, which introduce qualifiers into types, and *qualifier assertions*, which enforce checks on the qualifiers of a qualified type. These extensions allow the programmer to express the invariants that are to be checked by the qualifier inference rules.

We conclude this section with a brief illustration of the need for qualifier polymorphism. Qualifier polymorphism solves a problem with `const` familiar to C and C++ programmers. One of the more awkward consequences of the standard (monomorphic) C++ type system appears in the Standard Template Library (STL) [MSS96] for C++. STL must always explicitly provide two sets of operations, one for constant data structures and one for non-constant data structures. For illustration, consider the following pair of C functions:

```
typedef const int ci;
int *id1(int *x) { return x; }
ci  *id2(ci  *x) { return x; }
```

C programmers would like to have only one copy of this function, since both versions behave identically and in fact compile to the same code. Unfortunately we need both. A pointer to a constant cannot be passed to `id1` without a cast. A pointer to a non-constant can be passed to `id2`, but then the return value will be `const`. In the language of type theory, this difficulty occurs because the identity function has type $\kappa\,\alpha \rightarrow \kappa\,\alpha$, with qualifier set κ appearing both co- and contravariantly.

In part because of the lack of `const` polymorphism in C and C++, `const` is often either not used, or function results are deliberately cast to non-`const`. For example, the standard library function `strchr` takes a `const char *s` as a parameter but returns a `char *` pointing somewhere in `s`. By adding polymorphism, we allow `const` to be used more easily without resorting to casting.

The rest of this paper is organized as follows. Section 2 describes our framework in detail, including the rules for `const`. Section 3 discusses type inference, qualifier polymorphism, and soundness. Section 4 describes our `const`-inference system. Section 5 discusses related work, Section 6 suggests future directions, and Section 7 concludes.

2 Qualified Type Systems

For our purposes, *types* are terms over a set of type constructors Σ and type variables $TVar$. Program variables are denoted by $PVar$. Each type constructor c has an arity $a(c)$. We denote the set of types by Typ:

$$Typ ::= \alpha \mid c(Typ_1, \ldots, Typ_{a(c)})$$

where $\alpha \in TVar$ and $c \in \Sigma$. A *type environment* A is a map $A : PVar \rightarrow Typ$. We abbreviate the vector $(x_1, \ldots, x_n)$ by

[2] Apologies to Mark P. Jones for overloading the term *qualified types* [Jon92].

[3] Many descriptions of binding-time analysis omit the standard types. In such a system, this type would be written `static` (`dynamic` $\rightarrow$ `dynamic`).

$$
\begin{array}{rcll}
e & ::= & v & \\
& | & e_1\ e_2 & \\
& | & \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ \texttt{fi} & \\
& | & \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{ni} & \\
v & ::= & x & x \in PVar \\
& | & n & n \in \mathbb{Z} \\
& | & \lambda x.e & \\
\tau & ::= & \alpha & \\
& | & int & \\
& | & \tau \to \tau &
\end{array}
$$

Figure 1: Source language

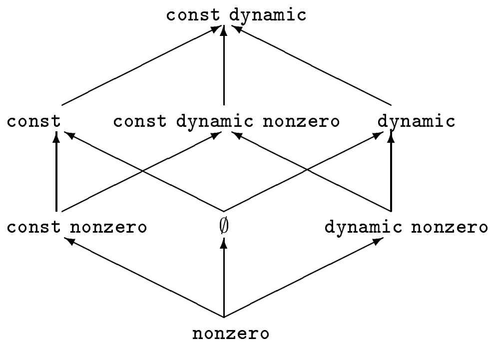

Figure 2: Example qualifier lattice

$\vec{x}$. We define

$$A[\vec{x} \mapsto \vec{\tau}](y) = \begin{cases} A(y) & y \notin \{x_1, \dots, x_n\} \\ \tau_i & y = x_i \end{cases}$$

where the x_i are distinct.

We demonstrate our framework by adding type qualifiers to the functional language shown in Figure 1. Using the C convention, we interpret 0 in the guard of an `if` statement as false and any non-zero value as true. Here we distinguish syntactic values v (which can be evaluated without computation) from general expressions e. We use a call-by-value language, though the addition of qualifiers works equally well for call-by-name languages.

For this language, $\Sigma = \{int, \to\}$ with arities 0 and 2, respectively, and the type system is that of the simply-typed lambda calculus. Although this language is convenient for demonstrating the type checking system, some qualifiers (e.g., `const`) are not meaningful in it. In Section 2.4, we add updateable references (in C terminology, l-values) to our language and give the additional inference rules for `const`.

The user supplies a set of qualifiers $q_1, \dots, q_n$, annotated to indicate the subtyping relation.

Definition 1 A type qualifier q is *positive* (*negative*) if $\tau \preceq q\ \tau$ ($q\ \tau \preceq \tau$) for any type τ.

For convenience we denote the absence of qualifier q by $\bot_q$ if q is positive or $\top_q$ if q is negative.

We extend subtyping to sets of qualifiers by defining a *qualifier lattice*.

Definition 2 (Qualifier lattice) Each positive qualifier q defines a two-point lattice $L_q = \bot_q \sqsubseteq q$. Each negative qualifier q defines a two-point lattice $L_q = q \sqsubseteq \top_q$. The *qualifier lattice* L is defined by $L = L_{q_1} \times \cdots \times L_{q_n}$. We write $\bot$ and $\top$ for the bottom and top elements of L.

$$
\begin{array}{rcl}
\rho & ::= & Q\ \tau \\
\tau & ::= & \alpha \mid int \mid (\rho_1 \to \rho_2) \\
Q & ::= & \kappa \mid l
\end{array}
$$

Figure 3: Type language with qualifiers

Clearly it is unnecessary to model both positive and negative qualifiers, since they are dual notions. Instead of using a negative qualifier q, we can give $\top_q$ a name and use it as a positive qualifier, rearranging the type inference rules appropriately. However, as it is often more intuitive to think of certain qualifiers as being positive or negative, we allow both.

Figure 2 shows the qualifier lattice for the positive qualifiers `const` and `dynamic` and the negative qualifier `nonzero`. A `nonzero` qualifier on an integer indicates that the integer cannot be zero. Instead of writing $\bot_q$ or $\top_q$ in the picture we have simply omitted the name q. (`static` is just another name for $\bot_{\texttt{dynamic}}$, and we have omitted it.) Notice that moving up the lattice adds positive qualifiers or removes negative qualifiers.

We use $\sqsubseteq$ for the ordering on L, and $\sqcap$ and $\sqcup$ for meet and join. For a positive (negative) qualifier q_i, we denote by $\neg q_i$ the lattice element $(\top_{q_1}, \dots, \top_{q_{i-1}}, \bot_{q_i}, \top_{q_{i+1}}, \dots, \top_{q_n})$ (for negative qualifiers $(\bot_{q_1}, \dots, \bot_{q_{i-1}}, \top_{q_i}, \bot_{q_{i+1}}, \dots, \bot_{q_n})$), where $\bot_{q_j}$ and $\top_{q_j}$ are the minimal and maximal elements of L_{q_j}.

This general formulation allows any combination of qualifiers to appear on any type. In practice, however, qualifiers need not be orthogonal. The analysis designer may specify inference rules that depend on multiple qualifiers and well-formedness conditions that prohibit certain combinations of qualifiers.

2.1 Qualified Types

The next step is to add qualifiers to the standard type system. We define a new set of types $QTyp$, the *qualified types*, by

$$
\begin{array}{rcll}
QTyp & ::= & Q\ \tau & \\
\tau & ::= & \alpha \mid c(QTyp_1, \dots, QTyp_{a(c)}) & c \in \Sigma \\
Q & ::= & \kappa \mid l &
\end{array}
$$

where $\kappa \in QVars$, the set of variables that range over type qualifiers, and the l are elements of the lattice L. The qualified types are just the standard types annotated with sets of qualifiers, i.e., lattice elements or qualifier variables. Notice that we do not need variables that range over qualified types, since the combination of a qualifier variable and a type variable $\kappa\ \alpha$ serves the same purpose.

Figure 3 shows the qualified types for our example language. To avoid ambiguity, we parenthesize function types. Example qualified types are `dynamic nonzero` int and `dynamic` (`const` $\alpha \to \kappa\ \beta$). Notice that we allow qualifiers to appear on all levels of a type, even though a particular qualifier may only be associated with certain standard types (e.g., `const` only applies to updateable references).

We now extend the $\sqsubseteq$ relation to a subtyping relation $\preceq$ on *QTyp*. We create a set of subtyping rules that give judgments $\vdash \rho \preceq \rho'$, meaning that ρ is a subtype of ρ'. We abbreviate $\vdash \{\rho \preceq \rho',\ \rho' \preceq \rho\}$ by $\vdash \rho = \rho'$. The system also uses judgments of the form $\vdash Q \sqsubseteq Q'$, which is valid if $Q \sqsubseteq Q'$ holds in the lattice. Figure 4a contains the subtyping rules for our example language.

The choice of subtyping rules depends on the meanings of the type constructors Σ. In general, for any $c \in \Sigma$ the rule

$$\frac{\begin{array}{c}\vdash Q \sqsubseteq Q' \\ \vdash \rho_i = \rho_i' \quad i \in [1..n]\end{array}}{\vdash Q\ c(\rho_1, \ldots, \rho_n) \preceq Q'\ c(\rho_1', \ldots, \rho_n')}$$

is sound. Indeed, this is the standard choice if c constructs updateable references (see Section 2.4).

2.2 Qualifier Annotations and Qualifier Assertions

Now we wish to extend the standard type system to infer qualified types. Our construction should apply to any set of type qualifiers. Thus we immediately encounter a problem, because when constructing a qualified type we do not know how to choose the top-level qualifiers, i.e., the qualifiers on the outermost constructor.

We divorce this issue from the type system by adding *qualifier annotations* to the source language. Initially we assume that any new top-level qualifier is $\perp$. We then allow user annotations that change the top-level qualifier monotonically. Dually, we also add *qualifier assertions* to the source language that allow the user to check the top-level qualifier on a type. While we also allow extra constraints on the qualifiers to be added to the type rules, qualifier assertions are a simple way to test invariants, and their use does not require extensive knowledge of type systems.

For our example language, we add productions for annotations and assertions:

$$\begin{array}{rcl} e & ::= & \cdots \\ & | & e|_l \\ & | & l\ e \end{array}$$

Here qualifier annotation $l\ e$ tells the type checker that $l\ e$'s top-level qualifier should be at least l. Note that the qualifier on an abstraction qualifies the function type itself and not the type of the parameter. The qualifier assertion $e|_l$ requires that if Q_e is e's top-level qualifier, then $Q_e \sqsubseteq l$.

2.3 Qualified Type Systems

The final step is to extend the original type checking system to handle qualified types. Intuitively this extension should be natural, in the sense that adding type qualifiers should not modify the structure of inferred types but only their qualifiers. We must also extend the type system with a subsumption rule, to allow subtyping, and rules for qualifier assertions and annotations. The resulting *qualified type system* for our example language is shown in Figure 4b. Judgments are of the form $A \vdash e : \rho$, meaning that in the type environment A expression e has qualified type ρ. The system in Figure 4 is the standard subtyping system (see [Mit91]) specialized to our application. Section 3.1 contains a formal description of the construction of a qualified type system from a standard type system.

In general each qualifier comes with a set of rules describing how the qualifier interacts with the operations in the language. Notice in Figure 4b that the antecedents of certain rules, e.g., (App), match the types of subexpressions against arbitrary qualifiers Q. We allow the qualifier designer to restrict these Q to enforce the semantics of particular qualifiers. In Section 2.4 we show how a type rule for assignment is modified for the `const` qualifier.

We define two pairs of transformation functions between standard and qualified types and expressions. For a qualified type $\rho \in QTyp$, we define $\mathsf{strip}(\rho) \in Typ$ to be ρ with all the qualifiers removed. Analogously, for an expression e in the annotated language, $\mathsf{strip}(e)$ is e without any qualifier annotations or assertions.

In the other direction, for a standard type $\tau \in Typ$ we define $\perp(\tau)$ to be the qualified type ρ with the same type structure as τ and all qualifiers set to $\perp$. Analogously, for an expression e in the original language, $\perp(e)$ is the corresponding expression in the annotated language with only $\perp$ qualifier annotations and no qualifier assertions.

Observation 1 Let $\vdash_S$ be the judgment relation of the type system of the simply-typed lambda calculus, and let $\vdash$ be the judgment relation of the type system given in Figure 4. Then

- If $\emptyset \vdash_S e : \tau$, then $\emptyset \vdash \perp(e) : \perp(\tau)$.
- If $\emptyset \vdash e' : \rho$, then $\emptyset \vdash_S \mathsf{strip}(e') : \mathsf{strip}(\rho)$.

This captures our intuitive requirement that the type qualifiers do not modify the underlying type structure.

Even without any additional rules on qualifiers, the qualified type system can be quite useful. Perhaps the most obvious kind of type qualifier to add is one that captures a property of a data structure. For example, we may want to distinguish between sorted lists and possibly unsorted lists. We add a negative type qualifier `sorted` and annotate all of our sorting functions so they return sorted lists. (We do not attempt to verify that `sorted` is placed correctly—we simply assume it is.) We can then add qualifier assertions, e.g., to check that a merge function is only called with sorted lists.

2.4 Example: `const`

Many qualifiers include restrictions on their usage. In our system, these restrictions can be expressed as qualifier assertions or as extra constraints between qualifiers. We illustrate the general pattern by adding updateable references (in C terminology, l-values) to our example language and giving the rules for `const`.

Qualifier annotations and assertions can always be used safely (see Section 3.3), whereas modifications to the type rules must be made with care. It is up to the qualifier designer to ensure that after any modifications the type inference rules remain not only sound, but also intuitive to the programmer, who sees only the presence or absence of qualifiers and not the underlying type system. This is especially important when designing multiple, interacting qualifiers, which can potentially complicate the type system.

We add ML-style references to the language in Figure 1; for a discussion of `const` in the C type system, see Section 4. As mentioned in the introduction `const` is positive (for any τ, $\tau \preceq$ `const` τ). We extend the source language and the

$$\frac{\vdash Q_1 \sqsubseteq Q_2}{\vdash Q_1\ int \preceq Q_2\ int} \qquad \text{(SubInt)}$$

$$\frac{\vdash Q_1 \sqsubseteq Q_2 \quad \vdash \rho_2 \preceq \rho_1 \quad \vdash \rho_1' \preceq \rho_2'}{\vdash Q_1\ (\rho_1 \to \rho_1') \preceq Q_2\ (\rho_2 \to \rho_2')} \qquad \text{(SubFun)}$$

(a) Subtyping rules

$$\frac{A \vdash e : \rho \quad \vdash \rho \preceq \rho'}{A \vdash e : \rho'} \qquad \text{(Sub)}$$

$$\frac{A \vdash e : Q\ \tau \quad \vdash Q \sqsubseteq l}{A \vdash e|_l : Q\ \tau} \qquad \text{(Assert)}$$

$$\frac{A \vdash e : Q\ \tau \quad \vdash Q \sqsubseteq l}{A \vdash l\ e : l\ \tau} \qquad \text{(Annot)}$$

$$\frac{}{A \vdash n : \bot\ int} \qquad \text{(Int)}$$

$$\frac{}{A \vdash x : A(x)} \qquad \text{(Var)}$$

$$\frac{A[x \mapsto \rho_x] \vdash e : \rho}{A \vdash \lambda x.e : \bot\ (\rho_x \to \rho)} \qquad \text{(Lam)}$$

$$\frac{A \vdash e_1 : Q\ (\rho_2 \to \rho) \quad A \vdash e_2 : \rho_2}{A \vdash e_1\ e_2 : \rho} \qquad \text{(App)}$$

$$\frac{A \vdash e_1 : Q\ int \quad A \vdash e_2 : \rho \quad A \vdash e_2 : \rho}{A \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ \texttt{fi} : \rho} \qquad \text{(If)}$$

$$\frac{A \vdash e_1 : \rho_1 \quad A[x \mapsto \rho_1] \vdash e_2 : \rho_2}{A \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{ni} : \rho_2} \qquad \text{(Let)}$$

(b) Syntax-directed rules

Figure 4: Basic type checking rules

qualified type language:

$$\begin{array}{lcl} e & ::= & \cdots \mid \texttt{ref}\ e \mid !e \mid e_1 := e_2 \\ v & ::= & \cdots \mid () \\ \tau & ::= & \cdots \mid ref(\rho) \mid unit \end{array}$$

In this language, $\texttt{ref}\ e$ creates an updateable reference, $!e$ returns the contents of a reference, and $e_1 := e_2$ stores the value of e_2 in reference e_1. The type *unit* has only one value, ().

Since we have introduced a new type constructor *ref*, we also need to describe how it interacts with subtyping. There are well-known problems with mixing subtyping and updateable references [AC96]. The obvious rule,

$$\frac{\vdash Q_1 \sqsubseteq Q_2 \qquad \vdash \tau_1 \preceq \tau_2}{\vdash Q_1\ ref(\tau_1) \preceq Q_2\ ref(\tau_2)} \qquad \text{(Unsound)}$$

is unsound. For example, suppose we allowed subtyping under a *ref*. Then we could typecheck the following code (any missing qualifiers are $\bot$):

```
1  let x = ref(nonzero 37) in
2  let y = x in
3     y := 0;
4     (!x)|nonzero
5  ni ni
```

Line 3 typechecks because we can promote the type of y to $\neg\texttt{nonzero}\ int$, since by subtyping $\texttt{nonzero}\ int \preceq \neg\texttt{nonzero}\ int$. But notice that this does not affect the type of x, hence line 4 also typechecks even though the contents of x is now 0.

The solution to this problem is to ensure that any aliases of the same *ref* cell contain the same qualifiers, which can be achieved by using equality on the type of the *ref*'s contents in the subtyping rule.

$$\frac{\vdash Q_1 \sqsubseteq Q_2 \qquad \vdash \rho_1 = \rho_2}{\vdash Q_1\ ref(\rho_1) \preceq Q_2\ ref(\rho_2)} \qquad \text{(SubRef)}$$

The subtyping rule for *unit* is the expected rule:

$$\frac{\vdash Q_1 \sqsubseteq Q_2}{\vdash Q_1\ unit \preceq Q_2\ unit} \qquad \text{(SubUnit)}$$

We give type rules for our new constructs; here we jump directly to the qualified type rules.

$$\frac{}{A \vdash () : \bot\ unit} \qquad \text{(Unit)}$$

$$\frac{A \vdash e : \rho}{A \vdash \texttt{ref}\ e : \bot\ ref(\rho)} \qquad \text{(Ref)}$$

$$\frac{A \vdash e : Q\ ref(\rho)}{A \vdash !e : \rho} \qquad \text{(Deref)}$$

$$\frac{A \vdash e_1 : Q\ ref(\rho_2) \quad A \vdash e_2 : \rho_2}{A \vdash e_1 := e_2 : \bot\ unit} \qquad \text{(Assign)}$$

The semantics of `const` requires that the left-hand side of an assignment be non-`const`. In our framework, this requirement can be expressed with an assertion $e_1|_{\neg\texttt{const}} := e_2$ on every assignment. Notice that such assertions can be added automatically.

Another way to add this restriction is to change (Assign). Recall that in our construction of the qualified type rules, whenever we needed to insert a qualifier but had no way of choosing one, we simply allowed all qualifiers. This is where Q came from in (Assign).

Rather than using annotations, we allow the qualifier designer to place restrictions at these choice points. Thus (Assign) becomes

$$\frac{A \vdash e_1 : \neg\texttt{const}\ \mathit{ref}(\rho_2) \quad A \vdash e_2 : \rho_2}{A \vdash e_1 := e_2 : \bot\ \mathit{unit}} \qquad (\text{Assign}')$$

2.5 Practical Considerations

Although adding qualifier annotations and assertions changes the syntax of the source language, in practice the changes to the lexer and parser can be minimal. We can require that all qualifiers begin with a reserved symbol, so that the lexer can unambiguously tokenize qualifiers. The grammar for types is extended so that qualifiers can appear on all levels of a type, using well-understood techniques to avoid ambiguity [ASU88]. We add a special syntactic form for assertions.

We have prototyped such a set of extensions to an ANSI C front end. The extended language accepts standard ANSI C as a subset. The extensions required only trivial modifications.

We can transform a qualified program to an unqualified program simply by removing the qualifiers and the assertions. One way to do this is to follow the approach of Evans [Eva96] and use special comment syntax for our language extensions. This has the advantage that a compiler for the standard language will automatically ignore all qualifiers, though it makes the parser for the qualified type system much more complicated, especially when arbitrary levels of qualification are permitted.

3 Type Inference, Polymorphism, and Soundness

3.1 Type Inference

The rules in Figure 4 describe a type checking system. We can also extend a type inference system in a similar way. As before we assume that the original type system is monomorphic; polymorphism can be dealt with as described in Section 3.2. We view the standard type inference system as a collection of type inference rules $R_1, \ldots, R_k$ giving judgments of the form $A \vdash e : \tau; C$, meaning in type environment A expression e has type τ under equality constraints C. Formally, the constraints generated by typing judgments are given by

$$C ::= \{\tau_1 = \tau_2\} \mid C_1 \cup C_2$$

A *solution* to a set of equality constraints $\{l_i = r_i\}$ is a substitution $S : \mathit{TVar} \to \mathit{Typ}$ that maps type variables to ground types (types without variables) such that $\vdash S(l_i) = S(r_i)$ for all i. If $A \vdash e : \tau; C$ and a solution S of C exists, then S defines a valid typing of e. If no solution exists, e is untypable.

For expository purposes we assume that the type rules R_i can be written in the form

$$\frac{\begin{array}{c} A[\vec{x}^1 \mapsto \vec{\tau}^1] \vdash e_1 : \tau_1; C_1 \quad \cdots \quad A[\vec{x}^n \mapsto \vec{\tau}^n] \vdash e_n : \tau_n; C_n \\ C = (\bigcup_{i=1}^n C_i) \cup \{l_i = r_i\} \end{array}}{A \vdash e : \tau_e; C}$$

where the e_i are the immediate subexpressions of e (i.e., the inference rules are compositional), and the $\{l_i = r_i\}$ are a set of equality constraints between types, usually the τ_i and τ_e.

In order to construct a new rule for qualified types, we define a spread operation (similar to [TT94])

$$\mathsf{sp} : (\mathit{TVar} \to \mathit{QTyp}) \times \mathit{Typ} \to \mathit{QTyp}$$

that consistently rewrites standard types as qualified types. The first parameter of $\mathsf{sp}(\cdot,\cdot)$ is a mapping V that is used to consistently rewrite type variables and metavariables, and the second parameter is the type to be rewritten.

$$\begin{array}{rcl} \mathsf{sp}(V, \alpha) & = & V(\alpha) \\ \mathsf{sp}(V, c(\tau_1, \ldots, \tau_{a(c)})) & = & \\ \multicolumn{3}{r}{\kappa\ c(\mathsf{sp}(V, \tau_1), \ldots, \mathsf{sp}(V, \tau_{a(c)}))} \end{array}$$

where the α are standard type variables and the κ are fresh variables ranging over lattice elements. Intuitively, whenever $\mathsf{sp}(\cdot,\cdot)$ encounters a type constructor, it does not know which qualifier to add, and so the translation allows any qualifier to appear on the constructor.

From the original type inference rules R_i we construct the qualified type inference rules R_i' as

$$\frac{\begin{array}{c} A[\vec{x}^1 \mapsto \overrightarrow{\mathsf{sp}(V,\tau)}^1] \vdash e_1 : \mathsf{sp}(V, \tau_1); C_1 \quad \cdots \\ A[\vec{x}^n \mapsto \overrightarrow{\mathsf{sp}(V,\tau)}^n] \vdash e_n : \mathsf{sp}(V, \tau_n); C_n \\ C = (\bigcup_{i=1}^n C_i) \cup \{\mathsf{sp}(V, l_i) = \mathsf{sp}(V, r_i)\} \end{array}}{A \vdash e : \mathsf{sp}(V, \tau_e); C}$$

where V maps each distinct metavariable τ in R_i to a distinct qualified type metavariable ρ, and each variable α in R_i to a distinct qualified type $\kappa\ \alpha$.

For example, in the standard type inference system for our language, the application rule is

$$\frac{\begin{array}{c} A \vdash e_1 : \tau_1; C_1 \quad A \vdash e_2 : \tau_2; C_2 \\ C = C_1 \cup C_2 \cup \{\tau_1 = \tau_2 \to \alpha\} \end{array}}{A \vdash e_1\ e_2 : \alpha; C}$$

The constructed rule in the qualified type system is

$$\frac{\begin{array}{c} A \vdash e_1 : \rho_1; C_1 \qquad A \vdash e_2 : \rho_2; C_2 \\ C = C_1 \cup C_2 \cup \{\rho_1 = \kappa\ (\rho_2 \to \kappa'\ \alpha)\} \end{array}}{A \vdash e_1\ e_2 : \kappa'\ \alpha; C}$$

As in Figure 4 we add a subsumption rule and rules for qualifier annotations and assertions to the constructed type inference system. The resulting qualified type inference system proves judgments of the form $A \vdash e : \rho; C$, where now C contains subtyping constraints and lattice inequalities:

$$C ::= \{\rho_1 \preceq \rho_2\} \mid \{Q_1 \sqsubseteq Q_2\} \mid C_1 \cup C_2$$

These constraints arise from the subsumption rule and from equality constraints in the original rules (recall that $\rho = \rho'$ is an abbreviation for $\{\rho \preceq \rho',\ \rho' \preceq \rho\}$, where ρ and ρ' are qualified types).

To solve the subtyping constraints, we first apply the subtyping rules (in Figure 4a for our example language) to the constraints so that we are left with only lattice constraints. These constraints are of the form $\kappa \sqsubseteq L$, $L \sqsubseteq \kappa$, or $L_1 \sqsubseteq L_2$. This is an atomic subtyping system, which can be solved in linear time for a fixed set of qualifiers [HR97].

3.2 Polymorphism

As mentioned in the introduction, we can add a notion of polymorphic type qualifiers. We begin by adding polymorphic constrained types σ to our type language:

$$
\begin{array}{rcl}
\sigma & ::= & \forall\vec{\kappa}.\rho\backslash C \\
\rho & ::= & Q\ \tau \\
\tau & ::= & \alpha \mid int \mid \rho_1 \to \rho_2 \\
Q & ::= & \kappa \mid l
\end{array}
$$

The type $\forall\vec{\kappa}.\rho\backslash C$ represents any type of the form $\rho[\vec{\kappa} \mapsto \vec{Q}]$ under the constraints $C[\vec{\kappa} \mapsto \vec{Q}]$, for any choice of qualifiers $\vec{Q}$. Note that polymorphism only applies to the qualifiers and not to the underlying types.

Following [OSW97], we introduce existential quantification on constraint systems:

$$C ::= \{\rho_1 \preceq \rho_2\} \mid \{Q_1 \sqsubseteq Q_2\} \mid C_1 \cup C_2 \mid \exists\vec{\kappa}.C$$

Note that we can lift existential quantification to the top-level by renaming variables. If $S[\vec{\kappa} \mapsto \vec{Q}]$ is a solution of C, then S is a solution of $\exists\vec{\kappa}.C$. Intuitively, existential quantification binds purely local qualifier variables (see below).

As is standard in `let`-style polymorphism [Mil78], we restrict the introduction of polymorphic types to `let`-expressions. Due to well-known problems with mixing updateable references and polymorphism, we only allow syntactic values (i.e., functions in C) to be polymorphic [Wri95]. We extend the qualified type inference system to introduce and eliminate polymorphic types:

$$\frac{A \vdash v : \rho_1; C_1 \quad A[x \mapsto \forall\vec{\kappa}.\rho_1\backslash C_1] \vdash e_2 : \rho_2; C_2 \quad \vec{\kappa} \text{ not free in } A}{A \vdash \texttt{let}\ x = v\ \texttt{in}\ e_2\ \texttt{ni} : \rho; (\exists\vec{\kappa}.C_1) \cup C_2} \quad (\mathrm{Let}_v)$$

$$\frac{A(x) = \forall\vec{\kappa}.\rho\backslash C}{A \vdash x : \rho[\vec{\kappa} \mapsto \vec{Q}]; C[\vec{\kappa} \mapsto \vec{Q}]} \quad (\mathrm{Var}')$$

In (Let_v), we bind $\vec{\kappa}$ before adding C_1 to the constraints generated by e_2 so the purely local $\vec{\kappa}$ can be renamed freely without changing the conclusion of the rule. This matches the intuition that the $\vec{\kappa}$ are local to the body of the `let`, and also allows for a relatively simple proof of soundness. See [EST95] for an alternate approach.

Polymorphism solves the problem with C's monomorphic type system that was outlined in the introduction. Consider the following code fragment:

```
1  let id = λx.x in
2  let y = id(ref 1) in
3  let z = id(const ref 1) in
   ...
   ni ni ni
```

We first derive that $\lambda x.x$ has type $\bot\ (\kappa_x\ \alpha_x \to \kappa_x\ \alpha_x)$. Then we apply the rule (Let_v) to give *id* the polymorphic type $\forall\kappa_x.\bot\ (\kappa_x\ \alpha_x \to \kappa_x\ \alpha_x)\backslash\emptyset$. Now when we apply *id* in lines 2 and 3, we can use rule (Var') to instantiate *id* with two separate sets of qualifiers, and so y can have type $\bot\ ref(\bot\ int)$ even though z must have type $\texttt{const}\ ref(\bot\ int)$.

3.3 Soundness

By using standard techniques found in [WF94, EST95, OSW97] we show that the type system we have presented, with qualifiers, references, and polymorphism, satisfies a subject reduction property. Our proof closely follows [EST95]. We give only a proof sketch, due to space limitations.

We begin by defining a store s as a finite mapping from locations (i.e., variables) to values. We denote locations by a as a reminder that they must be bound in the store. The semantics assumes that all values are qualified, so that a semantic value is a qualifier annotation and a syntactic value $(l\ v)$. A program can always be rewritten in this form by inserting $\bot$ annotations. We define a reduction context to fix the left-to-right ordering of evaluation:

$$
\begin{array}{rcl}
R & ::= & [\,] \mid R\ e \mid (l\ v)\ R \mid \texttt{if}\ R\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ \texttt{fi} \\
& & \mid \texttt{let}\ x = R\ \texttt{in}\ e_2\ \texttt{ni} \mid Q\ \texttt{ref}\ R \\
& & \mid !R \mid R := e \mid Q\ a := R \mid l\ R \mid R|_l
\end{array}
$$

We give single-step operational semantics for the execution of a program in Figure 5. A configuration $\langle s, e\rangle$ is a pair where s represents the store and e represents the current redex. We assume that all values are qualified. We extend typings to configurations:

Definition 3 (Store Typing) We write $A \vdash \langle s, e\rangle : \rho; C$ if both of the following hold:

1. $A \vdash e : \rho; C$
2. For all $a \in dom(s)$, $A(a) = Q_a\ ref(\rho_a)$ and $A \vdash s(a) : \rho_a; C$.

The first condition guarantees that e has the right type, and the second condition guarantees that the typing of the store is consistent with the values in the store.

Lemma 1 If $A \vdash e : \rho; C$ and S is a substitution such that SC is satisfiable, then $SA \vdash e : S\rho; SC$.

Proof: By induction on the derivation of $A \vdash e : \rho; C$. Since SC is satisfiable all subsets of the constraints SC are satisfiable. The only interesting case is in (Let_v). In this case, we first rewrite the proof of $A \vdash e : \rho; C$ so that none of the variables $\vec{\kappa}$ are changed by S; we can do so because the $\vec{\kappa}$ are bound by an existential quantifier in the conclusion of (Let_v). □

Theorem 1 (Subject Reduction) If $A \vdash \langle s, e\rangle : \rho; C$ and $\langle s, e\rangle \to \langle s', e'\rangle$, then there exists an A' such that $A'|_{dom(A)} = A$ and $A' \vdash \langle s', e'\rangle : \rho; C'$ where $C' \subseteq C$.

Proof: By induction on the structure of e. In the case of (Let_v), we need to show that we can give $e_2[x \mapsto v]$ the same type as $\texttt{let}\ x = v\ \texttt{in}\ e_2\ \texttt{ni}$. We have $A \vdash v : \rho_1; C_1$. In the typing proof $A[x \mapsto \forall\vec{\kappa}.\rho_1\backslash C_1] \vdash e_2 : \rho_2; C_2$, at each occurrence of x in e_2 we applied (Var') with some substitution S on $\vec{\kappa}$. By Lemma 1 we have $A \vdash v : S\rho_1; SC_1$, so we can replace x by v and prove the same judgment. □

Next we observe that *stuck* expressions (expressions that are not values but for which no reduction applies [WF94]) do not typecheck, which is trivial to prove. Then we can show

Corollary 1 (Soundness) If $\emptyset \vdash e : \rho; C$, then either e is a value or e diverges.

$$
\begin{array}{rcll}
\langle s, R[(l_2\ v)|_{l_1}]\rangle & \rightarrow & \langle s, R[l_2\ v]\rangle & l_2 \sqsubseteq l_1 \\
\langle s, R[l_1\ (l_2\ v)]\rangle & \rightarrow & \langle s, R[l_1\ v]\rangle & l_2 \sqsubseteq l_1 \\
\langle s, R[\texttt{if}\ (l\ n)\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ \texttt{fi}]\rangle & \rightarrow & \langle s, R[e_2]\rangle \quad n \neq 0 & \\
\langle s, R[\texttt{if}\ (l\ 0)\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ \texttt{fi}]\rangle & \rightarrow & \langle s, R[e_3]\rangle & \\
\langle s, R[(l\ \lambda x.e_1)\ v]\rangle & \rightarrow & \langle s, R[e_1[x \mapsto v]]\rangle & \\
\langle s, R[\texttt{let}\ x = v\ \texttt{in}\ e_2\ \texttt{ni}]\rangle & \rightarrow & \langle s, R[e_2[x \mapsto v]]\rangle & \\
\langle s, R[l\ \texttt{ref}\ v]\rangle & \rightarrow & \langle s[a \mapsto v], R[l\ a]\rangle & a \text{ fresh} \\
\langle s, R[!(l\ a)]\rangle & \rightarrow & \langle s, R[s(a)]\rangle & a \in \mathit{dom}(s) \\
\langle s, R[(l\ a) := v]\rangle & \rightarrow & \langle s[a \mapsto v], R[\bot\ ()]\rangle & a \in \mathit{dom}(s)
\end{array}
$$

Figure 5: Operational Semantics

4 Const Inference

In this section we describe a `const`-inference system for C that takes an entire C program and infers the maximum number of `const`s that can be syntactically present in the program. Such a system relieves the programmer of the burden of annotating all possible `const` locations. Instead the programmer can annotate the most important `const`s and use the inference to determine the `const`ness of the remaining variables and parameters. Furthermore, our experiments show that the polymorphic qualifier system allows more `const` annotations than the C type system, which is monomorphic.

4.1 C Types

C types already contain qualifiers, hence our implementation does not use the `sp` operator defined in Section 3.1. However, our system does need to translate the C types into the form described in Section 2.4. All variables in C refer to updateable memory locations. In the terminology of this paper, they are all *ref* types. When C variables appear in r-positions, they are automatically dereferenced. For example, consider the following code:

```
int x;
const int y;
x = y;
```

In our example language, this program is written `x := !y`. Omitting the qualifiers on *int*, let $A = \emptyset[\texttt{x} \mapsto \bot\ \mathit{ref}(\mathit{int}), \texttt{y} \mapsto \texttt{const}\ \mathit{ref}(\mathit{int})]$ as can be derived from the definitions of `x` and `y`. Then we can type this program in our system as follows:

$$
\dfrac{A \vdash \texttt{x} : \bot\ \mathit{ref}(\mathit{int}) \qquad \dfrac{A \vdash \texttt{y} : \texttt{const}\ \mathit{ref}(\mathit{int})}{A \vdash \texttt{!y} : \mathit{int}} \qquad \vdash \bot \sqsubseteq \neg\texttt{const}}{A \vdash \texttt{x := !y} : \mathit{unit}}
$$

Even though in the C type it appears that the `const` is associated with the `int`, in fact `const` qualifies the *ref* constructor of `y`. Hence `y`'s `const`ness does not affect `x`.

We can explain this systematically by giving a translation θ from the C types to *ref* types. For the sake of simplicity we only discuss pointer and integer types. Let the C types be given by the grammar

$$
\mathit{CTyp} ::= Q\ \texttt{int} \mid Q\ \texttt{ptr}(\mathit{CTyp})
$$

We define the mapping $\theta : \mathit{CTyp} \rightarrow \mathit{QTyp}$ as follows:

$$
\begin{array}{rcl}
\theta(\mathit{CTyp}) & = & Q'\ \mathit{ref}(\rho) \\
 & & \text{where } (Q', \rho) = \theta'(\mathit{CTyp}) \\[1ex]
\theta'(Q\ \texttt{int}) & = & (Q, \bot\ \mathit{int}) \\
\theta'(Q\ \texttt{ptr}(\mathit{CTyp})) & = & (Q, (Q'\ \mathit{ref}(\rho))) \\
 & & \text{where } (Q', \rho) = \theta'(\mathit{CTyp})
\end{array}
$$

Intuitively, the qualified type corresponding to a C type has one extra *ref* on the outside, and the `const` qualifiers have shifted up one level in the type. Note that these are the types of l-values, and the outermost *ref* should be removed to get the type of an r-value.

The advantage of this transformation is that we can use the standard subtyping rules for *ref*. Consider the following example:

```
int *x;
const int *y;
y = x;
```

In the C type system, we are assigning `x`, which has type `ptr(int)`, to `y`, which has type `ptr(const int)`, thus it appears that we are using a non-standard subtyping rule, because pointers are updateable. However, when we translate this into our system, we see that the r-value `x` has type $\bot\ \mathit{ref}(\mathit{int})$, and the l-value `y` has type $\bot\ \mathit{ref}(\texttt{const}\ \mathit{ref}(\mathit{int}))$. In order to assign `x` to `y`, we must show $\vdash \bot\ \mathit{ref}(\mathit{int}) \preceq \texttt{const}\ \mathit{ref}(\mathit{int})$ which is true in the standard subtyping relation we use.

4.2 Other Considerations

Ultimately we would like the analysis result to be the text of the original C program with some extra `const` qualifiers inserted. Thus we place some restrictions on the types we infer. In C different variables with the same `struct` type share the declaration of their fields. Thus in our system, if a and b are declared with the same `struct` type, we only allow a and b to differ on the outermost (top-level) qualifier; the qualifiers on their fields must be identical. For example, consider the following code:

```
struct st { int x; };
struct st a, b;
a = b;
```

The assignment `a=b` is equivalent to `a.x = b.x`. To satisfy the type rules, it is sufficient for the r-type of `b.x` to be a subtype of the r-type of `a.x`. However, because `a.x` and `b.x` share the field annotation in `struct st`, we require them to be equal. Note that the top-level qualifier attached to the *ref* constructors of the l-types of `a` and `b` can be distinct from each other. For example, although `a` must be a non-`const` *ref*, we do not require that `b` be non-`const`.

On the other hand, we treat typedefs as macro-expansions, e.g., in

```
typedef int *ip;
ip c, d;
```

`c` and `d` do not share any qualifiers.

One of the complications of analyzing real programs is that real programs use libraries, the code for which is often either unavailable or written in another language. For any undefined functions, we make the most conservative assumption possible: We treat any parameters not declared `const` as non-`const`. In general library functions are annotated with as many `const`s as possible, and so lack of `const` does mean can't-be-`const`.

C contains many different ways to defeat the type system, of which the most obvious is casting. For explicit casts we choose to lose any association between the value being cast and the resulting type. For implicit casts we retain as much information as possible.

Another way to defeat the type system is to use variable-length argument lists, or call a function with the wrong number of arguments. Both cases happen in practice; we simply ignore extra arguments.

4.3 Polymorphic Inference

Recall that we allow standard `let`-style polymorphism, in which polymorphic expressions are explicitly marked. Since a C program is made up of a set of possibly mutually-recursive functions, we need to syntactically analyze the program to find the `let` blocks.

Definition 4 The *function dependence graph (FDG)* of a program is a graph $G = (V, E)$ with vertices V and edges E. V is the set of all functions in the program, and there is an edge in E from `f` to `g` iff function `f` contains an occurrence of the name `g`.

The FDG exactly captures the implicit structure of function definitions. There is an edge from `f` to `g` if `g` must be type checked before `f`, and the strongly-connected components of the FDG are the sets of mutually-recursive functions.

To apply the polymorphic inference to a C program, we first construct the FDG. Then we traverse the strongly-connected components of the FDG in reverse depth-first order (the traversal can be computed in time linear in the size of the graph [CLR90]). We analyze each set of mutually recursive functions monomorphically and then we apply the rule for quantification. After we reach the root node of the FDG, we analyze any global variable definitions.

More work is required after type inference to measure the results. We want to know how many formal parameters can be polymorphic, i.e., either `const` or non-`const`. However, in general a C function may refer to global variables, so a C function's polymorphic type is not closed.

The types of global variables are closed once we have analyzed the whole program. A straightforward post-analysis pass combines this information with the types inferred during the FDG traversal to compute the results.

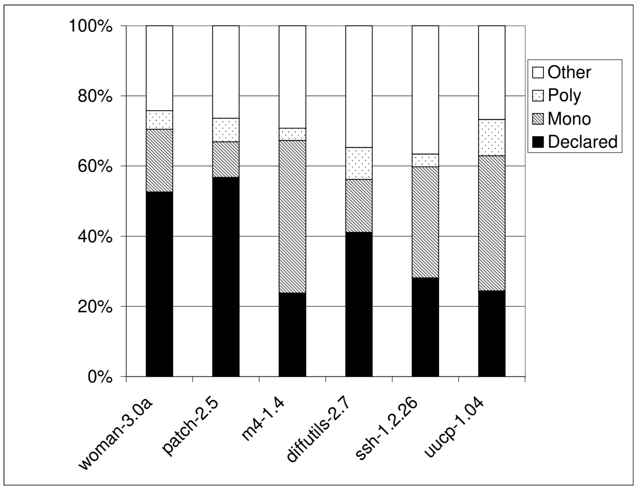

Figure 6: Number of inferred `consts` for benchmarks

We would prefer to use polymorphic recursion rather than let-style polymorphism to avoid working with the FDG, but BANE [AFFS98], the toolkit used to conduct our experiments, did not support polymorphic recursion when this work began. Because the qualifier lattice is finite and qualifiers do not change the type structure, the computation of polymorphic recursive types is decidable and in fact should be very efficient. We have recently learned that Jakob Rehof has written a polymorphic recursive type inference system for C++ [Reh99].

4.4 Experiments

We perform `const` inference using the rules for `const` outlined in Section 2.4. Table 1 lists the set of benchmarks we used to test our analysis. We purposely selected programs that show a significant effort to use `const`, rather than those that use it in only a few places.

Several of these "programs" are actually collections of programs that share a common code base. We analyzed each set of programs at once. This occasionally required renaming certain functions that were defined in several files to be distinct.

For each benchmark, we measured the number of interesting `consts` (see below) inferred by the monomorphic and the polymorphic version of our analysis. For any given *ref* type, there are three possible results that our analysis can infer: It can decide that the *ref*

1. must be `const`,
2. must not be `const`, or
3. could be either.

If the analysis inferred that something not marked as `const` must in fact be `const`, this would indicate a type error. Since all of our benchmarks are correct C programs, all of the possible additional `consts` detected must be from (3). The total number of possible `consts` is the sum of (1) and (3). Note that the number of possible `consts` does not depend on the source-level `const` annotations, since removing a `const` merely shifts the annotation on a *ref* type from (1) to (3).

Name	Lines	Description
woman-3.0a	1496	Replacement for man package
patch-2.5	5303	Apply a diff file to an original
m4-1.4	7741	Unix macro preprocessor
diffutils-2.7	8741	Collection of utilities for diffing files
ssh-1.2.26[4]	18620	Secure shell
uucp-1.04	36913	Unix to unix copy package

Table 1: Benchmarks for `const` inference

Name	Compile time (s)	Mono time (s)	Poly time (s)	Declared	Mono	Poly	Total possible
woman-3.0a	4.84	3.91	8.91	50	67	72	95
patch-2.5	16.98	18.70	33.43	84	99	107	148
m4-1.4	19.48	36.81	64.43	88	249	262	370
diffutils-2.7	24.46	35.70	57.34	153	209	243	372
ssh-1.2.26	84.55	101.90	174.28	147	316	347	547
uucp-1.04	113.75	177.71	457.16	433	1116	1299	1773

Table 2: Number of inferred possibly `const` positions for benchmarks

We only counted the number of "interesting" `consts` placed on arguments and results of defined functions. Recall that `consts` can only be placed on pointers and that arguments are passed by value, so the function `int foo(int x, int *y)` has only one interesting location where `const` can go, namely on the contents of `y`, which is itself a *ref*.

Figure 6 shows our results, which are tabulated in Table 2. Our current implementation uses BANE [AFFS98], a framework for constructing constraint-based analyses, for the qualifier inference. BANE handles constraint representation and solution, and our analysis tool generates constraints and interprets the results.

The first column of measurements gives the compile time. The next two columns give the running time (average of five) for the monomorphic and polymorphic `const`-inference. We do not include the parsing time. Note that the inference scales roughly linearly with the program size, and that the polymorphic inference takes at most 3 times longer than the monomorphic inference. Our implementation uses a generic set constraint engine to solve qualifier constraints, and we expect substantial speedups would be achieved with a framework specialized to the qualifier lattice.

The next column lists the number of interesting `consts` that were declared in the program. The right-most column indicates the total number of places that are syntactically allowed to have a `const` qualifier (according to our definition of interesting).

The Mono and Poly columns list the results of the monomorphic and polymorphic inference algorithms, respectively. As mentioned previously, any additional qualifiers inferred can be either `const` or non-`const` (these correspond to unconstrained qualifier variables). For the monomorphic type system we can make all of these positions `const` and still have a type correct program. For the polymorphic type system we need to leave these as unconstrained variables, since they may be required to be `const` or non-`const` in different contexts.

The measurements show that many more `consts` can be inferred than are typically present in a program. For some programs the results are quite dramatic, notably for uucp-1.04, which can have more than 2.5 times more `consts` than are actually present. Recall these are already programs in which some effort was made to use `const`.

For this set of benchmarks polymorphic analysis allows 5-16% more `consts` than monomorphic analysis. These results show that qualifier polymorphism is both useful and already latent in C programs, although we believe that most of the benefit for polymorphism comes from allowing fewer type casts rather than more `consts`.

Our experiments show that an automated inference tool makes it much easier for a programmer to fully use `const` annotations to express information about the side-effects of functions. They also show that polymorphism allows more `const` annotations than the monomorphic C type system without casts.

5 Related Work

There are three threads of related work: examples of systems that use type qualifiers, frameworks related to type qualifiers, and other techniques for checking programmer-specified invariants.

We have already mentioned the example qualifier systems of `const` from ANSI C [KR88], Evans's `lclint` [Eva96], and `static` and `dynamic` annotations from binding-time analysis [DHM95]. Two other examples are the secure information flow system of [VS97], which annotates types with high- and low-security qualifiers, and the λ-calculus with trust annotations of [ØP97]. [ØP97] suggests an extension of their system to multiple levels of trust, which is similar to our idea of a lattice of type qualifiers.

Another example comes from Titanium [YSP+98], a Java-based SPMD programming language. Titanium uses the qualifier `local` to distinguish pointers to local memory, which can be accessed with a simple load instruction, from pointers to non-local memory, which must be accessed with a network operation. A pointer annotated with `local` must

[4] The ssh distribution also includes a compression library `zlib` and the GNU MP library (arbitrary precision arithmetic). We treated both of these as unanalyzable libraries; `zlib` contains certain structures that are inconsistently defined across files, and the GNU MP library contains inlined assembly code.

be local; a pointer not annotated with `local` may either be local or non-local. In Titanium, `local` annotations are critical because they allow the compiler to remove expensive run-time tests.

Several other researchers have noted that type qualifiers are an important tool for program analysis. [Sol95] gives a framework for understanding a particular family of related analyses as type annotation (qualifier) systems. [ABHR99] describes the Dependency Core Calculus (DCC) and provides translations into DCC from several dependency-based type qualifier systems such as [VS97]. DCC is one example of a calculus based on monads. Recent work [Kie98, Wad98] has explored the connection between monads and effect systems [LG88]. Some effect systems can also be expressed as type qualifier systems. However, the exact connection between monads, effect systems, and type qualifiers is unclear.

Other frameworks choose a different design point by providing more powerful annotation languages. For example, Klarlund and Schwartzbach's *graph types* [KS93] allow programmers to specify detailed shape invariants on data structures. Another approach is the Extended Static Checking system [Det96, LN98], which uses sophisticated theorem-proving techniques that allow the programmer to check invariants. The advantage of such systems is that the invariants are much more precise than in a type qualifier system. However, specifying such invariants requires more effort and sophistication on the programmer's part.

6 Future Work

In the framework presented in this paper, types remain static throughout the source program, even though the values stored in some locations may change through updates. Indeed, as stated our framework cannot express the analysis of `lclint`, in which annotations on a given location may vary at each program point.

One solution we are investigating is to assign each location a distinct type at every program point and to add subtyping constraints between the different types. For example, suppose that `x` has type τ_1 before a non-branching statement s and `x` has type τ_2 after s. Then if s does not perform a strong update of `x` we add the constraint $\tau_1 \preceq \tau_2$; if s does strongly update `x` then we do not add this constraint. This technique allows a measure of flow sensitivity, which may make type qualifiers more useful in certain applications.

Finally, an issue we have not addressed is the presentation and specification of polymorphic function types. In our system each polymorphic type also carries a set of constraints, and we currently do not have a notation for specifying constraints in the source language. Additionally, in practice these constraint systems can be large and difficult to interpret. Simplifying these constrained types for presentation is an open research problem.

7 Conclusion

We believe that type qualifiers are a simple yet useful addition to standard type systems. We have presented a framework for adding type qualifiers, qualifier annotations, and qualifier assertions to an standard language, and we allow types to be polymorphic in the type qualifiers. Our experimental results show that for a set of benchmarks, many more `const` qualifiers can be added than are present, even though our benchmarks make significant use of `const`.

Acknowledgments

We would like to thank Daniel Weise, Henning Niss, Martin Elsman, Zhendong Su, and the anonymous referees for their helpful comments and suggestions.

References

[ABHR99] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A Core Calculus of Dependency. In *Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 147–160, San Antonio, Texas, January 1999.

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

[AFFS98] Alexander Aiken, Manuel Fähndrich, Jeffrey S. Foster, and Zhendong Su. A Toolkit for Constructing Type- and Constraint-Based Program Analyses. In *Proceedings of the second International Workshop on Types in Compilation*, Kyoto, Japan, March 1998.

[ASU88] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1988.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw Hill, 1990.

[Det96] David L. Detlefs. An overview of the Extended Static Checking system. In *Proceedings of the First Workshop on Formal Methods in Software Practice*, pages 1–9, January 1996.

[DHM95] Dirk Dussart, Fritz Henglein, and Christian Mossin. Polymorphic Recursion and Subtype Qualifications: Polymorphic Binding-Time Analysis in Polynomial Time. In *Static Analysis, Second International Symposium*, number 983 in Lecture Notes in Computer Science, pages 118–135, Glasgow, Scotland, September 1995. Springer-Verlag.

[EST95] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type Inference for Recursively Constrained Types and its Application to OOP. In *Mathematical Foundations of Programming Semantics, Eleventh Annual Conference*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995.

[Eva96] David Evans. Static Detection of Dynamic Memory Errors. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 44–53, Philadelphia, Pennsylvania, May 1996.

[Hen91] Fritz Henglein. Efficient Type Inference for Higher-Order Binding-Time Analysis. In J. Hughes, editor, *FPCA '91 Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 448–472, Cambridge, MA, August 1991. Springer-Verlag.

[HR97] Fritz Henglein and Jakob Rehof. The Complexity of Subtype Entailment for Simple Types. In *Proceedings, Twelfth Annual IEEE Symposium on Logic in Computer Science*, pages 352–361, Warsaw, Poland, July 1997.

[ICF98] *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming*, Baltimore, Maryland, September 1998.

[Jon92] Mark P. Jones. A theory of qualified types. In Bernd Krieg-Brücker, editor, *4th European Symposium on Programming*, number 582 in Lecture Notes in Computer Science, pages 287–306, Rennes, France, February 1992. Springer-Verlag.

[Kie98] Richard Kieburtz. Taming Effects with Monadic Typing. In ICFP'98 [ICF98], pages 51–62.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 2nd edition, 1988.

[KS93] Nils Klarlund and Michael I. Schwartzback. Graph Types. In *Proceedings of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 196–205, Charleston, South Carolina, January 1993.

[LG88] John M. Lucassen and David K. Gifford. Polymorphic Effect Systems. In *Proceedings of the 15th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 47–57, San Diego, California, January 1988.

[LN98] K. Rustan M. Leino and Greg Nelson. An Extended Static Checker for Modula-3. In *Compiler Construction: 7th International Conference*, volume 1383 of *Lecture Notes in Computer Science*, pages 302–305, April 1998.

[Mag93] Steve Maguire. *Writing Solid Code*. Microsoft Press, 1993.

[Mil78] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Mit91] John C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–285, July 1991.

[MSS96] David R. Musser, Atul Saini, and Alexander Stepanov. *STL Tutorial and Reference Guide*. Addison-Wesley Publishing Company, 1996.

[ØP97] Peter Ørbæk and Jens Palsberg. Trust in the λ-calculus. *Journal of Functional Programming*, 3(2):75–85, 1997.

[OSW97] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type Inference with Constrained Types. In Benjamin Pierce, editor, *Proceedings of the 4th International Workshop on Foundations of Object-Oriented Languages*, January 1997.

[Pur] Pure Atria. Purify: Fast detection of memory leaks and access errors.

[Reh99] Jakob Rehof. Personal communication, January 1999.

[Sol95] Kirsten Lackner Solberg. *Annotated Type Systems for Program Analysis*. PhD thesis, Aarhus University, Denmark, Computer Science Department, November 1995.

[TT94] Mads Tofte and Jean-Pierre Talpin. Implementation of the Typed Call-by-Value λ-Calculus using a Stack of Regions. In *Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 188–201, Portland, Oregon, January 1994.

[VS97] Dennis Volpano and Geoffrey Smith. A Type-Based Approach to Program Security. In Michel Bidoit and Max Dauchet, editors, *Theory and Practice of Software Development, 7th International Joint Conference*, volume 1214 of *Lecture Notes in Computer Science*, pages 607–621, Lille, France, April 1997. Springer-Verlag.

[Wad98] Philip Wadler. The Marriage of Effects and Monads. In ICFP'98 [ICF98], pages 63–74.

[WF94] Andrew K. Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115(1):38–94, 1994.

[Wri95] Andrew K. Wright. Simple Imperative Polymorphism. In *Lisp and Symbolic Computation 8*, volume 4, pages 343–356, 1995.

[YSP+98] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A High-Performance Java Dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, February 1998.