Type Qualifiers: Lightweight Specifications to Improve Software Quality

by

Jeffrey Scott Foster

B.S. (Cornell University) 1995
M.Eng. (Cornell University) 1996

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Alexander S. Aiken, Chair
Professor Susan L. Graham
Professor Hendrik W. Lenstra

Fall 2002

Type Qualifiers: Lightweight Specifications to Improve Software Quality

Abstract

Type Qualifiers: Lightweight Specifications to Improve Software Quality

by

Jeffrey Scott Foster

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Alexander S. Aiken, Chair

Software plays a pivotal role in our daily lives, yet software glitches and security vulnerabilities continue to plague us. Existing techniques for ensuring the quality of software are limited in scope, suggesting that we need to supply programmers with new tools to make it easier to write programs with fewer bugs. In this dissertation, we propose using type qualifiers, a lightweight, type-based mechanism, to improve the quality of software. In our framework, programmers add a few qualifier annotations to their source code, and type qualifier inference determines the remaining qualifiers and checks consistency of the qualifier annotations. In this dissertation we develop scalable inference algorithms for flow-insensitive qualifiers, which are invariant during execution, and for flow-sensitive qualifiers, which may vary from one program point to the next. The latter inference algorithm incorporates flow-insensitive alias analysis, effect inference, ideas from linear type systems, and lazy constraint resolution to scale to large programs. We also describe a new language construct "restrict" that allows a programmer to specify certain aliasing properties, and we give a provably sound system for checking usage of restrict. In our system, restrict is used to improve the precision of flow-sensitive type qualifier inference. Finally, we describe a tool for adding type qualifiers to the C programming language, and we present several experiments using our tool, including finding security vulnerabilities in popular C programs and finding deadlocks in the Linux kernel.

To my wife Elise

Contents

List of Figures

Acknowledgements

First and foremost, I would like to thank my advisor Alex Aiken for his advice, support, wisdom, and unrelenting optimism, all of which have benefited me tremendously the past six and a half years. I would also like to thank the current and former members of my research group, Manuel Fähndrich, Zhendong Su, David Gay, Ben Liblit, Tachio Terauchi, and John Kodumal. Their questions, answers, and camaraderie have been a source of inspiration I have come to rely on. I am also indebted to Martin Elsman, Umesh Shankar, Kunal Talwar, and David Wagner, my additional collaborators on some of this work. Finally, I would like to thank my other committee members, Susan Graham and Hendrik Lenstra, for their helpful advice and feedback.

Chapter 1

Introduction

Software systems are an increasingly important part of our daily lives. Today, everything from routine office transactions to critical infrastructure services relies on the effectiveness of large, complicated software systems, yet our ability to produce such systems far outstrips our ability to ensure their quality. Well-publicized software glitches have led to failures such as the Mars Climate Orbiter crash [76], and security vulnerabilities in software have paved the way for attacks such as the Code Red Worm [15]. The potentially staggering cost of software quality problems has led to a renewed call to increase the safety, reliability, and maintainability of software [49, 89].

There are currently two widely used techniques for validating program properties: testing and code auditing. In testing, either programmers or testers check their program on a series of inputs designed to exercise the system's various features. In code auditing, groups of programmers manually review source code together, looking for potential problems. Both techniques are extremely useful in practice: testing catches many errors and shows that the code runs correctly on a variety of inputs, and code auditing can find both obvious and subtle bugs in software. Unfortunately, while effective, both techniques have serious limitations. Although well-designed test cases cover much of the behavior of a program, the only assurance testing provides is that the test cases work correctly. Code auditing is extremely difficult, and it is unreasonable to assume that manual inspection can show the safety of a large, complicated software system. Given these limitations, it seems clear that we need to complement both testing and code auditing with other techniques.

In this dissertation we propose using *type qualifiers* to improve software quality. Type qualifiers are lightweight annotations for specifying program properties (see below).

In later chapters we present techniques that verify, at compile-time, the correctness of type qualifier annotations in source code. This kind of static, specification-and-checking approach has a number of advantages:

- Unlike dynamic techniques like testing, static analysis conservatively models all runs of a program. This is especially valuable for finding bugs that are hard to replicate and for finding security vulnerabilities, and both are exactly the problems that are most difficult to identify with testing.

- Static checking can provide strong guarantees by proving that certain program properties always hold. In particular, type qualifier systems, when applied to type-safe languages, are sound, meaning that programs with valid qualifier annotations do not violate the semantics of the qualifiers. This assurance enables the programmer to use type qualifiers to eliminate whole classes of bugs from their program.

- Programmers work hard to convince themselves that their programs behave as intended. By providing programmers with a specification language for writing down some of their intentions, and by providing automatic checking of their specifications, we help programmers write and design correct programs from the start.

- Specifications that are incorporated into the source code are a very precise form of documentation. Such documentation is invaluable when modifying and upgrading code, and automatic checking can identify attempted changes that violate existing interfaces.

The type qualifiers we propose as specifications are atomic properties that "qualify" the standard types. Many programming languages have a few special purpose type qualifiers. In contrast, in this dissertation we propose a general framework for adding new, user-specified qualifiers to languages such as C, C++, Java, and ML. In our framework, programmers add a few key type qualifier annotations to their programs and then apply *type qualifier inference* to the source code, automatically inferring the remaining qualifiers and checking the consistency of the qualifier annotations.

As one example, we can use type qualifiers to detect potential security vulnerabilities (Section 6.2). Security-conscious programs need to distinguish untrusted values read from the network from trusted values the program itself creates. We can model this property by using qualifiers *tainted* and *untainted* to mark the types of untrusted and trusted

data, respectively. Type qualifier errors occur when a value of type *tainted* T is used where a value of type *untainted* T is expected, where T is a standard unqualified type. Any such type qualifier error indicates a potential security vulnerability.

As another example, we can use type qualifiers to statically check correct usage of I/O operations on files (Section 6.4). In most systems, file operations can only be used in certain ways: a file must be opened for reading before it is read, it must be opened for writing before it is written to, and once closed a file cannot be accessed. We can express these rules with type qualifiers. We introduce qualifiers *open*, *read*, *write*, and *readwrite* to mark files that have been opened in an undetermined mode, for reading, for writing, and for both reading a writing, respectively. We also introduce a qualifier *closed* to mark files that are not open. File operations are given types for tracking file states. For example, the *close* function takes an *open file* and changes it to a *closed file*. Type qualifier errors occur when we misuse the file interface—for example, if we attempt to read a *closed file* or write a *read file*. Any such error indicates a potential bug in the program.

Type qualifiers have a number of advantages as a mechanism for specifying and checking properties of programs:

- Of the multitude of proposals for statically-checked program annotations, types are arguably the most successful. In many languages, programmers must already include type annotations in their source code. Thus the machinery of types is familiar to the programmer, and we believe it is natural for a programmer to specify additional properties with a type qualifier. This bodes well for the adoption of type qualifiers in practice, since a key concern about any specification language is whether programmers are willing to use it.

- Type qualifiers are additional annotations layered on top of the standard types. As such, they can be safely ignored by conventional tools (such as standard compilers) that do not understand them. This natural backward compatibility lowers the barrier to adopting type qualifiers.

- Type qualifiers support efficient inference, which reduces the burden on the programmer by requiring fewer annotations. Efficient inference also allows us to apply type qualifiers to large bodies of legacy code; we can sprinkle in a few type qualifier annotations, and inference determines the remaining qualifiers automatically.

- While lightweight, type qualifiers can express a number of interesting properties, some of which we discuss in Chapter 6.

Outline of This Work and Contributions

In our framework, type qualifiers are added to every level of the standard types. The key technical property of type qualifiers is that they do not affect the underlying standard type structure (Section 2.1 describes standard types). That is, a program with type qualifier annotations should type check only if the same program type checks with the annotations removed. Aside from this restriction, type qualifiers could potentially affect program semantics arbitrarily. In this dissertation, however, we focus on a very useful subclass of type qualifiers, those that introduce *subtyping*.

In our system, each set of related type qualifiers forms a partial order (Section 2.3), which is extended to a subtype relation among *qualified types*, which are simply types with qualifiers. For example, consider the qualifiers *tainted* and *untainted*. While it is an error for *tainted* data to be used in *untainted* positions, the reverse is perfectly fine—presumably positions that accept *tainted* data can accept any kind of data. Thus we choose $\mathit{untainted} < \mathit{tainted}$ as the partial order. For the qualifiers *open*, *read*, *write*, *readwrite*, and *closed*, we choose the partial order

$$\mathit{readwrite} < \mathit{read} < \mathit{open}$$
$$\mathit{readwrite} < \mathit{write} < \mathit{open}$$

In other words, a file open for reading and writing can be treated as a file open reading or as a file open for writing, and any of those is an *open* file. A closed file can never be considered a open file, nor vice-versa, hence *closed* is incomparable to the other four qualifiers.

Chapter 3 presents a generic system for extending a standard type system with type qualifiers and related annotations. Chapter 3 also describes an algorithm for performing *flow-insensitive* (see below) *type qualifier inference*. Our inference algorithm is designed using *constraint-based analysis*. To infer qualifiers in a source program, we scan the program text and generate a series of constraints $q_1 \leq q_2$ among qualifiers and qualifier variables, which stand for as-yet-unknown qualifiers. We solve the constraints for the qualifier variables and warn the programmer if the constraints have no solution, which indicates a type qualifier error.

In program analysis, there is an important distinction between *flow-insensitive* analysis, which tends to be very efficient, and *flow-sensitive* analysis, which is more precise but usually does not scale well to whole programs. Flow-insensitive analysis proves facts about a program that are true throughout the whole execution. For example, it is reasonable to model *tainted* and *untainted* qualifiers flow-insensitively, i.e., variables are either always tainted or always untainted. Flow-sensitive analysis, in contrast, proves facts that may change from one program point to another. For example, it makes the most sense to model *open*, *closed*, etc. flow-sensitively, since the state of a file can vary from one program point to the next.

The centerpiece of this dissertation is Chapter 4, which presents a flow-sensitive type qualifier system, including a novel, lazy constraint-based algorithm for flow-sensitive type qualifier inference. In contrast to classical data flow analysis, the system described in Chapter 4 explicitly models pointers, heap-allocated data, aliasing, and function calls. Since we would like to apply our system to large programs, our inference algorithm is carefully crafted to scale to whole program analysis. We use an inexpensive flow-insensitive alias analysis and effect inference to produce an approximate model of the store. That model of the store forms the basis for a second inference step that computes flow-sensitive information, using ideas from linear type systems to model updates. To achieve scalability, rather than explicitly modeling the entire state at each program point, we lazily solve only a portion of the constraints generated by the second step, namely the portion of the constraints needed to check qualifier annotations.

In the context of our flow-sensitive qualifier system, we introduce a new language construct that may be of independent interest, `restrict` $x = e_1$ `in` e_2 (Section 4.2). The `restrict` construct, related to the ANSI C type qualifier of the same name [6], allows programmers to specify *aliasing* behavior in their programs. We say that two expressions referring to a memory location *alias* if they evaluate to the same location. The presence of aliasing, which is an essential feature of most modern programming languages, makes program analysis much more difficult. A programmer can use occurrences of `restrict` $x = e_1$ `in` e_2 to help improve the precision of a program analysis. In this construct, the name x is initialized to e_1, which must be a pointer, and x is in scope during evaluation of e_2. Suppose x and e_1 point to object o, i.e., $*x$ and $*e_1$ alias, where $*e$ reads indirectly through pointer e. At a high level, the meaning of `restrict` is that, within the scope of e_2, only x and values derived from x may be used to access o. This fact often allows a program analysis—

in particular, our flow-sensitive type qualifier inference—to track the state of o precisely within e_2, intuitively because the `restrict` construct guarantees that the programmer does not modify o "behind the back" of the program analysis; the programmer always modifies o through x or values derived from x. Our inference system checks the correctness of `restrict` annotations using effects, and we give proof that this checking is sound.

To test our ideas in practice, we built a tool called CQUAL for adding user-defined flow-insensitive and flow-sensitive type qualifiers to C (Chapter 5). CQUAL has been used both in our own research and by others [127]. A key feature of CQUAL is that it includes a user interface that shows programmers not only what type qualifiers were inferred but why they were inferred. After inference, the program source code is presented to the user with each identifier colored according to its inferred qualifiers. For each error message, the user can browse qualifier constraints that exhibit the error. For example, if an error occurs because *tainted* data is used in an *untainted* position, the user is shown a set of constraints and a program path that shows step-by-step how *tainted* was propagated to *untainted*. From our own personal experience, such an interface, while often neglected in the research literature, is one of the most important and visible features of any program analysis tool, and we found the interface invaluable in our research.

We have performed a number of experiments with CQUAL (Chapter 6). We have used CQUAL to infer *const* qualifiers [6] in ANSI C programs. We found that qualifier inference is able to infer many additional *const*s, even in programs that already make a significant effort to use *const* (Section 6.1). We have used CQUAL to check for format-string bugs, a particular kind of vulnerability, in several popular C programs. Using CQUAL, we were able to find security vulnerabilities that were not known to us (Section 6.2). We have used CQUAL to find several new deadlocks in the Linux kernel (Section 6.3). Finally, we have used CQUAL to check for proper file operation usage in two C programs (Section 6.4).

In summary, this dissertation makes a number of new contributions:

- We present a framework for adding type qualifiers to almost any language with standard types, and we show that flow-insensitive type qualifier inference can be carried out efficiently.

- We show how to extend our system to flow-sensitive type qualifiers, and we give a novel, lazy, scalable, constraint-based algorithm for inferring flow-sensitive type qualifiers.

- We introduce a new language construct `restrict` that allows programmers to specify aliasing behavior in their programs. We give a system for checking `restrict` and show that it is sound.
- We describe a practical tool CQUAL that adds type qualifiers to the C programming language. We believe many of the lessons learned in developing CQUAL are applicable to other languages, as well.
- We present empirical evidence that type qualifiers are useful in practice by describing a number of experiments with type qualifier systems, both flow-insensitive and flow-sensitive. In the process, we show that our algorithms scale to large programs.

Chapter 2

Background

In this and the next two chapters we present the theoretical underpinnings of type qualifier systems. Type qualifiers can be added to any language with a standard static type system. In order to abstract away from many of the tedious details of real languages, in this and the next two chapters we present type qualifiers in the context of a particularly simple, abstract language: the call-by-value lambda calculus [55] extended with updatable references [122]. Chapter 5 discusses an implementation of type qualifiers for the C programming language.

Figure 2.1 gives our source language. Note that there are no qualifiers in this language; we introduce qualifiers in Chapter 3. We sometimes use the non-terminal v to denote *values*, which are expressions that cannot be further evaluated.[1] Our language contains three kinds of values: *variables*, written with lowercase letters x, y, z, etc., integers, and *functions* $\lambda x.e$, which denotes a function with parameter x that evaluates to e. The constructs in our language that can be evaluated are *function application* $e_1\ e_2$, which applies function e_1 to argument e_2, *name binding* $\texttt{let}\, x = e_1\, \texttt{in}\, e_2$ which evaluates e_1 and then binds the variable x to e_1 within the scope of e_2, *allocation* $\texttt{ref}\, e$, which allocates a new cell in memory and initializes it to e, *dereference* $\texttt{*}e$, which returns the contents of cell e, and *assignment* $e_1 \texttt{ := } e_2$, which replaces the contents of cell e_1 with the value of e_2. Rather than add explicit recursion to the language, we assume without further comment that we have a primitive function `Y` such that $\texttt{Y}\ f$ reduces to $f\ (\texttt{Y}\ f)$ [120, 122].

In addition to the grammar for our source language, in order to reason about how a program is supposed to behave we need to have some formal statement of what a program

[1] We do not allow evaluation within a function body

e	::=	v	values
	$\mid$	$e_1\ e_2$	application
	$\mid$	$\texttt{let}\, x = e_1\, \texttt{in}\, e_2$	name binding
	$\mid$	$\texttt{ref}\, e$	allocation
	$\mid$	$\texttt{*}e$	dereference
	$\mid$	$e_1\, \texttt{:=}\, e_2$	assignment
v	::=	x	variable
	$\mid$	n	integer
	$\mid$	$\lambda x.e$	function

Figure 2.1: Source Language

means, i.e., we need a semantics for our language. Figure 2.2 gives a big-step operational semantics [90] for our language. In these semantics a store S is a mapping from locations l to values v. We use $\emptyset$ for the empty store.

In the rules in Figure 2.2, as well as throughout this dissertation, we present semantics and type systems in the *natural deduction* style. Rules are of the form

$$\frac{H_1 \quad \ldots \quad H_n}{C}$$

meaning that if we know the hypotheses H_1 through H_n are true, then we can prove that conclusion C is true. For example, here is modus ponens written in this style:

$$\frac{A \qquad A \Rightarrow B}{B}$$

If we know that A is true and A implies that B is true, then we can conclude that B is true.

The semantics in Figure 2.2 is a set of reduction rules of the form $S \vdash e \rightarrow v; S'$, meaning that in initial store S, expression e evaluates to value v and yields a new store S'. Here a value is either a location, an integer, or a function. Notice that our semantics contains no environments for variables—instead, we use substitution to bind variables to values. We discuss each of the rules:

- In [Var], [Int], and [Lam], values remain the same—they do not reduce any further.
- In [App], we evaluate $e_1\ e_2$ by first evaluating e_1, which must yield a function of the form $\lambda x.e$. Next we evaluate e_2, which yields some value v. Finally, we evaluate the

$$\frac{l \in dom(S)}{S \vdash l \to l; S} \text{ [Var]}$$

$$\frac{}{S \vdash n \to n; S} \text{ [Int]}$$

$$\frac{}{S \vdash \lambda x.e \to \lambda x.e; S} \text{ [Lam]}$$

$$\frac{S \vdash e_1 \to \lambda x.e; S' \quad S' \vdash e_2 \to v; S'' \quad S'' \vdash e[x \mapsto v] \to v'; S'''}{S \vdash e_1\ e_2 \to v'; S'''} \text{ [App]}$$

$$\frac{S \vdash e_1 \to v_1; S' \quad S' \vdash e_2[x \mapsto v_1] \to v_2; S''}{S \vdash \texttt{let}\, x = e_1\, \texttt{in}\, e_2 \to v_2; S''} \text{ [Let]}$$

$$\frac{S \vdash e \to v; S' \quad l \notin dom(S')}{S \vdash \texttt{ref}\, e \to l; S'[l \mapsto v]} \text{ [Ref]}$$

$$\frac{S \vdash e \to l; S' \quad l \in dom(S')}{S \vdash *e \to S'(l); S'} \text{ [Deref]}$$

$$\frac{S \vdash e_1 \to l; S' \quad S' \vdash e_2 \to v; S'' \quad l \in dom(S'')}{S \vdash e_1 \texttt{:=} e_2 \to v; S''[l \mapsto v]} \text{ [Assign]}$$

Figure 2.2: Big-Step Operational Semantics

function body e with formal argument x replaced by actual argument v. Notice the sequencing specified in this rule: we evaluate function application left-to-right, and we evaluate the argument of a function before performing the function call. The latter corresponds to our choice of a call-by-value semantics.

- In [Let], we evaluate e_1 first to yield a value v_1, and then we evaluate e_2 with x in e_2 replaced by v_1. Notice that in fact we could treat $\texttt{let}\, x = e_1\, \texttt{in}\, e_2$ as syntactic sugar for $(\lambda x.e_2)\ e_1$.

- In [Ref], we first evaluate e to yield value v. Then we find an unused (fresh) location l, and we return a new store in which l has been bound to value v. The whole expression evaluates to l.

- In [Deref], we first evaluate e, which yields a location l. We then return the value bound to l in S'.

- Finally, in [Assign] we evaluate e_1 followed by e_2 (notice the left-to-right order of evaluation). The evaluation of e_1 must yield a location l, and we rebind l to the value of e_2.

The operational semantics of Figure 2.2 shows us how to execute a program in our source language. Unfortunately, not all programs in our source language make sense—some programs cannot be executed according to the rules in Figure 2.2. For example, rule [App] requires that the expression in application position evaluate to a function. If we try to evaluate the expression 3 4 (apply function 3 to argument 4), then rule [App] does not apply. In fact, there is no rule that allows us to evaluate 3 4, since 3 is not a function.

We model such erroneous programs by reducing them to a special symbol **err**. The symbol **err** is not a value. Intuitively, if while evaluating an expression e we encounter an expression to which the rules of Figure 2.2 do not apply, then $S \vdash e \rightarrow \mathbf{err}; S'$. As shorthand, we often write this as $S \vdash e \rightarrow \mathbf{err}$, since S' is meaningless once we have produced an **err** result. For the sake of completeness, Figure 2.3 gives the error reduction rules for our semantics. In these rules we use the symbol r to stand for either a value v or the symbol **err**. Notice that Figure 2.3 contains two kinds of rules: rules that propagate **err** from a sub-step of the reduction (our semantics is *strict* in **err**), and rules that introduce **err** when an error is detected locally. Thus there are three possible results for evaluating an expression: either it reduces to a value, it reduces to **err**, or it does not terminate.

2.1 Standard Type Systems

As we have just seen, in our operational semantics, among all the programs we can write down in our source language there are some bad ones that reduce to **err**. The main goal of adding a type system to a language is to disallow such programs. Since determining whether a program reduces to **err** is undecidable [60], we choose to make our type system *sound* but not *complete*: none of the programs our type system accepts reduce to **err**, but there may be some programs our type system rejects that also do not reduce to **err**.

We observe that one major case when reduction fails is when we apply an operation to the wrong kind of object (for example, we try to assign to a function, or we try to use an integer as a function). The idea behind standard static type systems is to try to assign a static (compile-time) *type* to each expression e, indicating whether e is an integer, a

$$\frac{S \vdash e_1 \to r \qquad r \text{ is not of the form } \lambda x.e}{S \vdash e_1\ e_2 \to \mathbf{err}}\ [\text{App}']$$

$$\frac{S \vdash e_1 \to \lambda x.e; S' \qquad S' \vdash e_2 \to \mathbf{err}}{S \vdash e_1\ e_2 \to \mathbf{err}}\ [\text{App}'']$$

$$\frac{\begin{array}{c} S \vdash e_1 \to \lambda x.e; S' \qquad S' \vdash e_2 \to v; S'' \\ S'' \vdash e[x \mapsto v] \to \mathbf{err} \end{array}}{S \vdash e_1\ e_2 \to \mathbf{err}}\ [\text{App}''']$$

$$\frac{S \vdash e_1 \to \mathbf{err}}{S \vdash \texttt{let}\, x = e_1\, \texttt{in}\ e_2 \to \mathbf{err}}\ [\text{Let}']$$

$$\frac{S \vdash e_1 \to v_1; S' \qquad S' \vdash e_2[x \mapsto v_1] \to \mathbf{err}}{S \vdash \texttt{let}\, x = e_1\, \texttt{in}\ e_2 \to \mathbf{err}}\ [\text{Let}'']$$

$$\frac{S \vdash e \to \mathbf{err}}{S \vdash \texttt{ref}\, e \to \mathbf{err}}\ [\text{Ref}']$$

$$\frac{S \vdash e \to r \qquad r \text{ is not of the form } l}{S \vdash *e \to \mathbf{err}}\ [\text{Deref}']$$

$$\frac{S \vdash e \to l; S' \qquad l \notin dom(S')}{S \vdash *e \to \mathbf{err}}\ [\text{Deref}'']$$

$$\frac{S \vdash e_1 \to r \qquad r \text{ is not of the form } l}{S \vdash e_1 := e_2 \to \mathbf{err}}\ [\text{Assign}']$$

$$\frac{S \vdash e_1 \to l; S' \qquad S \vdash e_2 \to \mathbf{err}}{S \vdash e_1 := e_2 \to \mathbf{err}}\ [\text{Assign}'']$$

$$\frac{S \vdash e_1 \to l; S' \qquad S \vdash e_2 \to v; S'' \qquad l \notin dom(S'')}{S \vdash e_1 := e_2 \to \mathbf{err}}\ [\text{Assign}''']$$

Figure 2.3: Big-Step Operational Semantics, Error Rules

function, or a memory cell. Then when we see an operation on e, we can determine at compile-time whether it is valid. For example, if we see $e_1\ e_2$, we accept this application as valid only if e_1 is a function, and if the type of the domain (parameter type) of e_1 matches the type of e_2. Note that reduction also may fail if we encounter a variable that is not bound in the current environment; our type system also prevents this from happening.

The types s (for $s$tandard type) we assign to expressions are given by the following

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

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

$$\frac{\Gamma[x \mapsto s] \vdash e : s'}{\Gamma \vdash \lambda x.e : s \longrightarrow s'} \text{ (Lam)}$$

$$\frac{\Gamma \vdash e_1 : s \longrightarrow s' \quad \Gamma \vdash e_2 : s}{\Gamma \vdash e_1\ e_2 : s'} \text{ (App)}$$

$$\frac{\Gamma \vdash e_1 : s_1 \quad \Gamma[x \mapsto s_1] \vdash e_2 : s_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : s_2} \text{ (Let)}$$

$$\frac{\Gamma \vdash e : s}{\Gamma \vdash \texttt{ref}\ e : ref\ (s)} \text{ (Ref)}$$

$$\frac{\Gamma \vdash e : ref\ (s)}{\Gamma \vdash \texttt{*}e : s} \text{ (Deref)}$$

$$\frac{\Gamma \vdash e_1 : ref\ (s) \quad \Gamma \vdash e_2 : s}{\Gamma \vdash e_1\ \texttt{:=}\ e_2 : s} \text{ (Assign)}$$

Figure 2.4: Standard Type Checking System

grammar:

$$s ::= int \mid ref\ (s) \mid s \longrightarrow s'$$

The type *int* is the type of integers. The type *ref* (s) is the type of a pointer to something of type s. Finally, the type $s \longrightarrow s'$ is the type of a function that given a parameter of type s produces a result of type s'. We can use the standard technique of currying [10] to model functions with multiple parameters.

Our type system is presented as a set of judgments $\Gamma \vdash e : s$, meaning that under type assumption Γ, expression e has type s. The type assumption Γ is a mapping from variables to types; intuitively Γ assigns types to the free variables of e. We write $\emptyset$ for the empty mapping.

Figure 2.4 presents our type checking rules. We discuss each of the rules:

- (Var) assigns a variable x the type it has in environment Γ. If x is not assigned a type in Γ, then this rule cannot apply—hence we reject as ill-typed any programs that

contain free variables.

- (Int) assigns all integers the type *int*.
- (Lam) assign a function $\lambda x.e$ type $s \longrightarrow s'$ if, under the assumption that x has type s, we can show that e has type s'.
- (App) checks that e_1 is a function and that the type of e_2 matches the type of the domain of e_1. Then (App) assigns $e_1\ e_2$ the type of the range of e_1.
- (Let) computes the type s_1 of e_1, and then type checks e_2 under the assumption that x has type s_1. The type of the `let` expression is the type of e_2.
- (Ref) computes the type s of e, and then assigns `ref` e the type *ref* (s), i.e., pointer to type s.
- (Deref) computes the type of e and checks that it is a *ref* type. (Deref) assigns $*e$ the type e points to.
- (Assign) checks that e_1 is an updatable reference (a pointer), and that e_2 is of the type e_1 points to. Then the expression e_1 `:=` e_2 is given the type of e_2.

Definition 2.1 *If e is a closed expression and there exists a type s such that $\emptyset \vdash e : s$, then we say that e* type checks*; informally we say that e* has type *s.*

A key property enjoyed by our type system is *subject reduction*, meaning an expression's type is preserved by reduction:

Lemma 2.2 (Subject Reduction) *If e is a closed expression, $\emptyset \vdash e \rightarrow r$, and $\emptyset \vdash e : s$, then $\emptyset \vdash r : s$.*

Note that this lemma is usually presented in a weaker form to make a proof by induction easier. We shall not give a proof of this lemma, since it is well known and follows from the soundness proofs in Appendices A and B, which are for more complicated type systems described in later chapters.

Using the subject reduction lemma, soundness follows immediately, since **err** has no type:

Theorem 2.3 (Type Soundness) *If e is a closed expression, $\emptyset \vdash e \rightarrow r$, and $\emptyset \vdash e : s$ for some s, then r is not **err**.*

Thus we see that type correct programs have the desirable property that they never reduce to **err**. Notice, however, that we have *not* shown that a program that type checks is "correct." For example, a program that type checks could fail to terminate, or it could produce an answer that has the right type but the wrong value. But the partial correctness guarantee of a type-correct program is still extremely valuable, because it ensures that the program is free from a large class of errors, allowing the programmer to spend their time and energy elsewhere. In subsequent chapters in this dissertation, we develop type qualifier systems to help the programmer eliminate still larger classes of application-specific errors.

2.2 Standard Type Inference

Given that type correct programs have the desirable property that they never reduce to **err**, we would like to be able to type check all programs. However, observe that to apply the type checking rules in Figure 2.4, we need some extra information besides the bare program in our source language. In particular, to apply (Lam) we somehow need to guess a type s for the function parameter in order to satisfy the hypothesis of (Lam). But where does this type come from?

One reasonable solution is to require that programmers annotate their programs with types. In particular, we extend the syntax for function definition to $\lambda x : s.e$, where s is the type of the parameter. Then, whenever we apply (Lam), we get the type of the parameter from the source code. This is the solution used in languages such as C, C++, and Java.

It turns out, however, that there is a well-known alternative solution: *type inference*. Instead of requiring that the programmer annotate each function parameter with a type, we can infer the types automatically. This is the solution used in languages such as ML and Haskell, and it has the advantage that the programmer is freed from the burden of writing down the types explicitly. ML and Haskell also support *(parametric) polymorphic* type inference, which allows the same piece of code to be automatically reused at different types. However, this dissertation focuses on monomorphic types, so we will not discuss polymorphism over standard types.

Figure 2.5 shows the rules for performing standard type inference on a program with no explicit type annotations. These rules prove judgments of the form $\Gamma \vdash' e : s$, meaning as before that in type environment Γ, expression e has type s. In this and the

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

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

$$\frac{\Gamma[x \mapsto \alpha] \vdash' e : s' \quad \alpha \text{ fresh}}{\Gamma \vdash' \lambda x.e : \alpha \longrightarrow s'} \text{ (Lam}')$$

$$\frac{\Gamma \vdash' e_1 : s_1 \quad \Gamma \vdash' e_2 : s_2 \quad s_1 = s_2 \longrightarrow \beta \quad \beta \text{ fresh}}{\Gamma \vdash' e_1\ e_2 : \beta} \text{ (App}')$$

$$\frac{\Gamma \vdash' e_1 : s_1 \quad \Gamma[x \mapsto s_1] \vdash' e_2 : s_2}{\Gamma \vdash' \texttt{let}\, x = e_1\, \texttt{in}\, e_2 : s_2} \text{ (Let}')$$

$$\frac{\Gamma \vdash' e : s}{\Gamma \vdash' \texttt{ref}\, e : ref\ (s)} \text{ (Ref}')$$

$$\frac{\Gamma \vdash' e : s \quad s = ref\ (\beta) \quad \beta \text{ fresh}}{\Gamma \vdash' \texttt{*}e : \beta} \text{ (Deref}')$$

$$\frac{\Gamma \vdash' e_1 : s_1 \quad \Gamma \vdash' e_2 : s_2 \quad s_1 = ref\ (s_2)}{\Gamma \vdash' e_1 \texttt{:=} e_2 : s_2} \text{ (Assign}')$$

Figure 2.5: Standard Type Inference System

$$
\begin{array}{rcll}
C \cup \{\alpha = s\} & \Rightarrow & C[\alpha \mapsto s] & \text{add } \alpha \mapsto s \text{ to solution} \\
C \cup \{s = \alpha\} & \Rightarrow & C[\alpha \mapsto s] & \text{add } \alpha \mapsto s \text{ to solution} \\
C \cup \{int = int\} & \Rightarrow & C & \\
C \cup \{ref\ (s_1) = ref\ (s_2)\} & \Rightarrow & C \cup \{s_1 = s_2\} & \\
C \cup \{s_1 \longrightarrow s_2 = s_1' \longrightarrow s_2'\} & \Rightarrow & C \cup \{s_1 = s_1'\} \cup \{s_2 = s_2'\} & \\
C \cup \{\text{other type eqn}\} & \Rightarrow & \text{unsatisfiable} &
\end{array}
$$

Figure 2.6: Type Equality Constraint Resolution

remainder of the dissertation we distinguish the inference version from the checking version of a type system by adding a prime to the judgment and to the rule labels. Our type inference rules are remarkably similar to the type checking rules of Figure 2.4, so we do not explain them in detail. There are two key differences. First, we add *type variables* α, which stand for unknown types that have yet to be determined, to our language of types:

$$s ::= \alpha \mid int \mid ref\ (s) \mid s \longrightarrow s'$$

We usually write type variables with Greek letters near the beginning of the alphabet (α, β, etc.). We call the set of types without type variables *ground types*. Whenever we encounter an expression whose type we need to guess (for instance, a function parameter in (Lam$'$)), we give as its type a fresh type variable.

Second, as we perform type inference, we discover constraints among certain types. For example, in (App$'$) when we see $e_1\ e_2$, we know that e_1 must be a function whose domain type matches the type of e_2. We write these conditions on the side with *type equality constraints* $s_1 = s_2$.

After applying the type inference rules in Figure 2.5, we have two things: a proof tree in exactly the shape we need to perform type checking and a set of type equality constraints on the type variables appearing in the proof. The last step we need is to solve the type equality constraints. Let C be a set of type equality constraints $s_1 = s_2$.

Definition 2.4 *A* solution σ *to a system of constraints* C *is a mapping from type variables to ground types (types without variables) such that for each constraint* $s_1 = s_2$ *in* C*, we have* $\sigma(s_1) = \sigma(s_2)$.

If σ is a solution to the constraints C, we write $\sigma \models C$. Figure 2.6 gives a set of resolution rules for checking whether a set of type equality constraints C has a solution. These rules

should be read as left-to-right rewrite rules, in which the system of constraints on the left is replaced by the (simpler) system of constraints on the right of each $\Rightarrow$. After we have applied the rules in Figure 2.6, we have either determined that the constraints are unsatisfiable, or we have computed a partial function σ assigning types to some of the variables in our program. The remaining type variables, and the remaining variables in the range of σ, are unconstrained, and hence we can set them arbitrarily—for example, to *int*. In fact, the rules in Figure 2.6 compute a most general solution—they constrain as few type variables as possible.

Lemma 2.5 *The rules in Figure 2.6 compute a solution σ to a system of constraints C if and only if a solution exists. Moreover, if $\sigma' \models C$, then there exists an R such that $\sigma' = \sigma \circ R$, i.e., σ is a most general solution.*

Thus we see that if any solution to a system of constraints exists, the rules in Figure 2.6 will succeed.

Standard type inference is very efficient. The rules in Figure 2.6 can be implemented using unification [3]. Given an initial, untyped program of size n, standard type inference takes $O(n\alpha(n))$ time, where $\alpha(n)$ is the inverse Ackerman's function.

In this dissertation we assume that programs are fully annotated with their standard types, either by the programmer or by a preliminary step of type inference.

2.3 Partial Orders and Lattices

In this dissertation we are concerned with adding type qualifiers to further expand the classes of bugs that type systems can prevent. In our framework, type qualifiers are related to each other by a partial order. In this section we define partial orders and lattices and give some of their basic properties. For an excellent introduction to the theory of partial orders and lattices, see Davey and Priestley [24].

Definition 2.6 *A* partial order *is a pair $(S, \leq)$ consisting of a set S and a relation $\leq$ on S such that $\leq$ is reflexive, anti-symmetric, and transitive.*

We write $a < b$ if $a \leq b$ and $a \neq b$. If it is clear from context we often refer to a partial order $(S, \leq)$ simply by the name S.

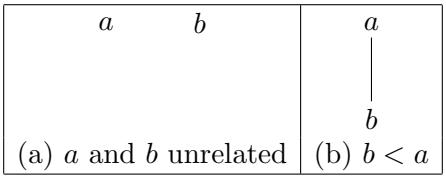

Figure 2.7: Two-point Partial Orders

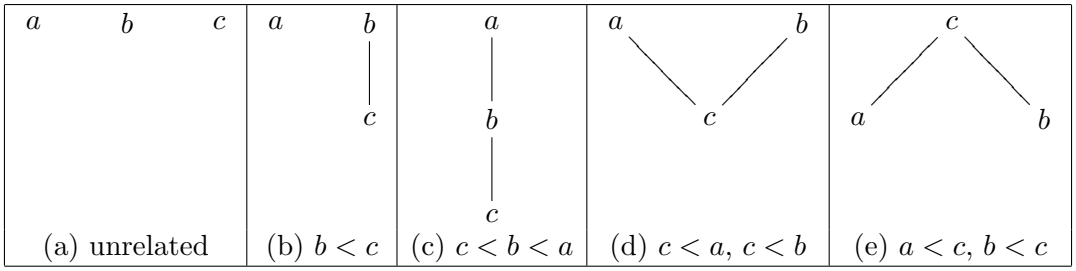

Figure 2.8: Three-Point Partial Orders

There are several basic partial orders we use in this dissertation. Given any set S, we can define the *discrete partial order* on S as the partial order $(S, \emptyset)$, meaning that no two elements are related. Let $S_2 = \{a, b\}$. Then there are exactly two partial orders on S_2: Either a and b are unrelated (the discrete partial order), or we have $a < b$. (The case $b < a$ is isomorphic to the second case.) Figure 2.7 contains a graphical representation of these two partial orders. Similar, Figure 2.8 contains a graphical representation of the five possible three-point partial orders. See Davey and Priestley [24] for a precise definition of such graphs.

Given two partial orders, one useful way of combining them to form a new partial order is to take their *cross product*.

Definition 2.7 *Let $(S_1, \leq_1)$ and $(S_2, \leq_2)$ be two partial orders. Then the partial order $(S_1, \leq_1) \times (S_2, \leq_2)$ is defined as $(S, \leq)$ where $S = S_1 \times S_2$, and $(a_1, a_2) \leq (b_1, b_2)$ iff $a_1 \leq_1 b_1$ and $a_2 \leq_2 b_2$.*

Given a partial order, we define two relations between its elements, the *least upper bound* or *join* $\sqcup$ and the *greatest lower bound* or *meet* $\sqcap$:

Definition 2.8 *If a and b are elements of a partial order, then $a \sqcup b$ is the element such*

that

1. $a \leq a \sqcup b$ *and* $b \leq a \sqcup b$
2. *If* $a \leq c$ *and* $b \leq c$, *then* $a \sqcup b \leq c$.

Definition 2.9 *If* a *and* b *are elements of a partial order, then* $a \sqcap b$ *is the element such that*

1. $a \sqcap b \leq a$ *and* $a \sqcap b \leq b$
2. *If* $c \leq a$ *and* $c \leq b$, *then* $c \leq a \sqcap b$

Note that it is not always the case that $a \sqcap b$ and $a \sqcup b$ are defined uniquely, and they may not even be defined at all if a and b are unrelated. If for any two elements $\sqcap$ ($\sqcup$) is always defined, we refer to the partial order a *meet (join) semilattice*. If a partial order is both a meet and a join semilattice, we call the partial order a *lattice*. For example, Figures 2.7b and 2.8c are lattices, Figure 2.8d is a meet semilattice, and Figure 2.8e is a join semilattice.

It is also useful to define two closure operations on elements of a partial order $(S, \leq)$.

Definition 2.10 *The* upward closure $\uparrow a$ *of an element* a *is defined as* $\uparrow a = \{b \mid a \leq b\}$. *The* downward closure $\downarrow a$ *of an element* a *is defined as* $\downarrow a = \{b \mid b \leq a\}$.

We extend $\uparrow$ and $\downarrow$ to sets of elements in the natural way, $\uparrow S = \bigcup_{s \in S} \uparrow s$ and $\downarrow S = \bigcup_{s \in S} \downarrow s$.

Chapter 3

Flow-Insensitive Type Qualifiers

In this section we present an extension to standard type systems that incorporates *flow-insensitive type qualifiers*. In general, type qualifiers can be added to any language with a standard type system. Throughout this chapter we use the language first introduced in Chapter 2 to illustrate the process. We assume that our input programs are type correct with respect to the standard type system of Chapter 2, and that function definitions have been annotated with standard types s. If that is not the case, we can perform a preliminary standard type inference pass.

3.1 Qualifiers and Qualified Types

As discussed in Chapter 1, in our system the user specifies a set of qualifiers Q and a partial order $\leq$ among the qualifiers. In practice, the user may wish to specify several sets $(Q_i, \leq_i)$ of qualifiers that do not interact, each with their own partial order. But then we can choose $(Q, \leq) = (Q_1, \leq_1) \times \cdots \times (Q_n, \leq_n)$, so without loss of generality we can assume a single partial order of qualifiers. For example, Figure 3.1 gives two independent partial orders and their equivalent combined, single partial order (in this case the partial orders are lattices). These particular qualifiers are described in more detail in Chapter 5. In Figure 3.1, as in the rest of this dissertation, we write elements of Q using *slanted text*. We sometimes refer to elements of Q as *type qualifier constants* to distinguish them from type qualifier variables introduced in Section 3.4.

For our purposes, *types* Typ are terms over a set Σ of n-ary type constructors.

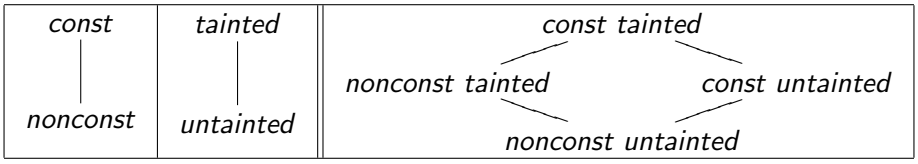

Figure 3.1: Example Qualifier Partial Order

Grammatically, types are defined by the language

$$Typ ::= c(Typ_1, \ldots, Typ_{arity(c)}) \qquad c \in \Sigma$$

In our source language, the type constructors are $\{int, ref, \longrightarrow\}$ with arities 0, 1, and 2, respectively. We construct the *qualified types* $QTyp$ by pairing each standard type constructor in Σ with a type qualifier (recall that a single type qualifier in our partial order may represent a set of qualifiers in the programmer's mind). We allow type qualifiers to appear on every level of a type. Grammatically, our new types are

$$Typ ::= Q\ c(Typ_1, \ldots, Typ_{arity(c)}) \qquad c \in \Sigma$$

For our source language, the qualified types are

$$\begin{aligned} \tau &::= Q\ \sigma \\ \sigma &::= int \mid ref\ (\tau) \mid \tau \longrightarrow \tau \end{aligned}$$

To avoid ambiguity, when writing down qualified function types we parenthesize them as $Q\ (\tau \longrightarrow \tau)$. Some example qualified types in our language are $\mathsf{tainted}\ \ int$ and $\mathsf{const}\ \ ref\ (\mathsf{untainted}\ \ int)$. We define the *top-level qualifier* of type $Q\ \sigma$ as its outermost qualifier Q.

So far we have types with attached qualifiers and a partial order among the qualifiers. A key idea behind our framework is that the partial order on type qualifiers induces a *subtyping* relation among qualified types. In a subtyping system, if type B is a subtype of type A, which we write $B \leq A$ (note the overloading on $\leq$), then wherever an object of type A is allowed an object of type B may also be used. Object-oriented programming languages such as Java and C++ are perhaps the most well known examples of subtyping systems (usually called *subclassing* in an object-oriented context).

Figure 3.2 shows how a given qualifier partial order is extended to a subtyping relation for our source language. In the first rule ($\text{Int}_{\leq}$) we have $Q\ \ int \leq Q'\ \ int$ if $Q \leq Q'$.

$$\frac{Q \leq Q'}{Q\ \ int \leq Q'\ \ int}\ (\text{Int}_{\leq})$$

$$\frac{Q \leq Q' \qquad \tau = \tau'}{Q\ \ ref\ (\tau) \leq Q'\ \ ref\ (\tau')}\ (\text{Ref}_{\leq})$$

$$\frac{Q \leq Q' \qquad \tau_2 \leq \tau_1 \qquad \tau_1' \leq \tau_2'}{Q\ (\tau_1 \longrightarrow \tau_1') \leq Q'\ (\tau_2 \longrightarrow \tau_2')}\ (\text{Fun}_{\leq})$$

Figure 3.2: Subtyping Qualified Types

This same rule generalizes to any nullary type constructor—for example, *char*, *float*, *double*, etc..

In the last rule ($\text{Fun}_{\leq}$), we constrain the outermost qualifiers as in ($\text{Int}_{\leq}$), and we require that functions are *contravariant* in their domain (the subtyping direction is reversed) and *covariant* in their range (the subtyping direction is preserved). For a discussion, see Mitchell [78].

In the middle rule ($\text{Ref}_{\leq}$), we again constrain the outermost qualifiers as in ($\text{Int}_{\leq}$), and we also require that the types of data stored in the references be equal (i.e., $\tau \leq \tau'$ and $\tau' \leq \tau$). At first glance this rule looks overly conservative—it would be more natural to only require

$$\frac{Q \leq Q' \qquad \tau \leq \tau'}{Q\ \ ref\ (\tau) \leq Q'\ \ ref\ (\tau')}\ (\text{Wrong})$$

Unfortunately, this turns out to be unsound. Consider the following code fragment (we omit the qualifiers on the references for this example):

```
let u : ref (untainted int) = ref 0 in      /* u points to untainted data */
let t : ref (tainted int) = u in            /* Allowed by (Wrong) */
    t := ⟨tainted data⟩                     /* Oops! Wrote tainted data into untainted u. */
```

According to (Wrong), we can bind t to u because *ref* (*untainted* *int*) $\leq$ *ref* (*tainted* *int*). But then $*t$ and $*u$ refer to the same object, yet they have different types. Therefore in the assignment we can store tainted data into u by writing through t, even though $*u$ is supposed to be untainted. This is a well-known problem, and the standard solution is to use the rule in Figure 3.2, which requires that the pointed-to types of an updatable reference are equal.[1]

[1] Java uses the rule (Wrong) for arrays. In Java, if S is a subclass of T, then $S[]$ is a subclass of $T[]$, where

e	::=	v	values
	|	$e_1\ e_2$	application
	|	$\texttt{let}\ x = e_1\ \texttt{in}\ e_2$	name binding
	|	$\texttt{ref}\ e$	allocation
	|	$\texttt{*}e$	dereference
	|	$e_1\ \texttt{:=}\ e_2$	assignment
	|	$\texttt{annot}(e, Q)$	**qualifier annotation**
	|	$\texttt{check}(e, Q)$	**qualifier check**
v	::=	x	variable
	|	n	integer
	|	$\lambda x\texttt{:}s.e$	function

Figure 3.3: Source Language with Qualifier Annotations and Checks

In general, for any $c \in \Sigma$ the rule

$$\frac{Q \leq Q' \qquad \tau_i = \tau_i' \qquad i \in [1..n]}{Q\ c(\tau_1, \ldots, \tau_n) \leq Q'\ c(\tau_1', \ldots, \tau_n')}$$

should be sound. Whether the equality can be relaxed for any particular position depends on the meaning of the type constructor c.

3.2 Qualifier Assertions and Annotations

Next we wish to extend our standard type system to work with qualified types. Thus far, however, we have supplied no mechanism that allows programmers to talk about which qualifiers are used in their programs. One place where this issue comes up is when constructing a qualified type during type checking. For example, if we see an occurrence of the integer `0` in the program, how do we decide which qualifier Q to pick for its type $Q\ \mathit{int}$? We wish to have a generic solution for this problem that allows programmers to talk about qualifiers without modifying the type rules.

In our system, we extend the syntax with two new forms, shown marked in boldface in Figure 3.3. A *qualifier annotation* $\texttt{annot}(e, Q)$ specifies the outermost qualifier Q to add to e's type. Annotations may only be added to expressions that construct a term, and whenever the user constructs a term our type system requires that they add an annotation.

$X[]$ is an array of X's. Java gets away with this by inserting run-time checks at every assignment into an array to make sure the type system is not violated. Since we seek a purely static system, Java's approach is not available to us.

Clearly this last requirement is not always desirable, and in Section 3.4 we describe an inference algorithm that allows programmers to omit these annotations if they like.

Dually, a *qualifier check* $\texttt{check}(e, Q)$ tests whether the outermost qualifier of e's type is compatible with Q. Notice that if we want to check a qualifier deeper in a type, we can do so by first applying our language's deconstructors. (For example, we can check the qualifier on the contents of a reference x using $\texttt{check}(*x, Q)$).

3.3 Flow-Insensitive Type Qualifier Checking

Finally, we wish to extend the original type checking system to a *qualified type system* that checks programs with qualified types, including our new syntactic forms $\texttt{annot}(\cdot, \cdot)$ and $\texttt{check}(\cdot, \cdot)$. Intuitively this extension should be natural, in the sense that adding type qualifiers should not modify the type structure (we make this precise below). We also need to incorporate a subsumption rule [78] into our qualified type system to allow subtyping.

We define a pair of translation functions between standard and qualified types and expressions. For a qualified type $\tau \in \mathit{QTyp}$, we define $\mathit{strip}(\tau) \in \mathit{Typ}$ to be τ with all qualifiers removed. Analogously, $\mathit{strip}(e)$ is e with any qualifier annotations or checks removed. In the other direction, for a standard type $s \in \mathit{Typ}$ we define $\mathit{embed}(s, q)$ to be the qualified type with the same shape as t and all qualifiers set to q. Analogously, $\mathit{embed}(e, q)$ is e with $\texttt{annot}(e', q)$ wrapped around every subexpression e' of e that constructs a term. Figure 3.4 gives formal definitions of *strip* and *embed*.

Figure 3.5 shows the qualified type system for our source language. Judgments are either of form $\Gamma \vdash_q e : \sigma$ (the first three rules of Figure 3.5a) or $\Gamma \vdash_q e : \tau$ (the remaining rules), meaning that in type environment Γ, expression e has unqualified type σ or qualified type τ. Here Γ is a mapping from variables to qualified types.

The rules (Int_q) and (Ref_q) are identical to the rules from the standard type checking system in Figure 2.4. (Lam_q) is also as before, plus we check that the parameter's qualified type τ has the same shape as the specified standard type. (This check is not strictly necessary—see Lemma 3.1 below.) Notice that these three rules produce types that are missing a top-level qualifier. The rule (Annot_q) adds a top-level qualifier to such a type, which is produced in our qualified type grammar by non-terminal σ. Inspection of the type rules shows that judgments of the form $\Gamma \vdash_q e : \sigma$ can only be used in the hypothesis of (Annot_q). Thus the net effect of the four rules in Figure 3.5a is that all constructed terms

$$
\begin{array}{rcl}
\mathit{strip}(Q\ \mathit{int}) & = & \mathit{int} \\
\mathit{strip}(Q\ \mathit{ref}\ (\tau)) & = & \mathit{ref}\ (\mathit{strip}(\tau)) \\
\mathit{strip}(Q\ (\tau \longrightarrow \tau')) & = & \mathit{strip}(\tau) \longrightarrow \mathit{strip}(\tau')
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{strip}(x) & = & x \\
\mathit{strip}(n) & = & n \\
\mathit{strip}(\lambda x.e) & = & \lambda x.\, \mathit{strip}(e) \\
\mathit{strip}(e_1\ e_2) & = & \mathit{strip}(e_1)\ \ \mathit{strip}(e_2) \\
\mathit{strip}(\texttt{let}\, x = e_1\, \texttt{in}\ e_2) & = & \texttt{let}\, x = \mathit{strip}(e_1)\, \texttt{in}\ \ \mathit{strip}(e_2) \\
\mathit{strip}(\texttt{ref}\, e) & = & \texttt{ref}\ \ \mathit{strip}(e) \\
\mathit{strip}(\texttt{*}e) & = & \texttt{*}\, \mathit{strip}(e) \\
\mathit{strip}(e_1\, \texttt{:=}\, e_2) & = & \mathit{strip}(e_1)\, \texttt{:=}\, \mathit{strip}(e_2) \\
\mathit{strip}(\texttt{annot}(e, Q)) & = & \mathit{strip}(e) \\
\mathit{strip}(\texttt{check}(e, Q)) & = & \mathit{strip}(e)
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{embed}(\mathit{int}, q) & = & q\ \ \mathit{int} \\
\mathit{embed}(\mathit{ref}\ (s), q) & = & q\ \ \mathit{ref}\ (\mathit{embed}(q, s)) \\
\mathit{embed}(s \longrightarrow s', q) & = & q\ (\mathit{embed}(s, q) \longrightarrow \mathit{embed}(s', q))
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{embed}(x, q) & = & x \\
\mathit{embed}(n, q) & = & \texttt{annot}(n, q) \\
\mathit{embed}(\lambda x.e, q) & = & \texttt{annot}(\lambda x.\, \mathit{embed}(e, q), q) \\
\mathit{embed}(e_1\ e_2, q) & = & \mathit{embed}(e_1, q)\ \ \mathit{embed}(e_2, q) \\
\mathit{embed}(\texttt{let}\, x = e_1\, \texttt{in}\ e_2, q) & = & \texttt{let}\, x = \mathit{embed}(e_1, q)\, \texttt{in}\ \ \mathit{embed}(e_2, q) \\
\mathit{embed}(\texttt{ref}\, e, q) & = & \texttt{annot}(\texttt{ref}\ \ \mathit{embed}(e, q), q) \\
\mathit{embed}(\texttt{*}e, q) & = & \texttt{*}\, \mathit{embed}(e, q) \\
\mathit{embed}(e_1\, \texttt{:=}\, e_2, q) & = & \mathit{embed}(e_1, q)\, \texttt{:=}\, \mathit{embed}(e_2, q)
\end{array}
$$

Figure 3.4: Definitions of $\mathit{strip}(\cdot)$ and $\mathit{embed}(\cdot, \cdot)$

must be assigned a top-level qualifier with an explicit annotation.

The rules (Var_q) and (Let_q) are identical to the standard type checking rules. The rules (App_q), (Deref_q), and (Assign_q) are similar to the standard type checking rules, except that they match the types of their subexpressions against qualified types. Notice that these three rules allow arbitrary qualifiers (denoted by Q) when matching a type. Only the rule (Check_q) actually tests a qualifier on a type.

Finally, the subsumption rule (Sub_q) allows us to use a subtype anywhere a supertype is expected. Notice that this is a non-syntactic rule that can be applied to any expression (the other rules apply only to one form of expression). While this is convenient

$$\frac{}{\Gamma \vdash_q n : \mathit{int}} \ (\mathrm{Int}_q)$$

$$\frac{\Gamma[x \mapsto \tau] \vdash_q e : \tau' \qquad \mathit{strip}(\tau) = s}{\Gamma \vdash_q \lambda x{:}s.e : \tau \longrightarrow \tau'} \ (\mathrm{Lam}_q)$$

$$\frac{\Gamma \vdash_q e : \tau}{\Gamma \vdash_q \mathtt{ref}\, e : \mathit{ref}\ (\tau)} \ (\mathrm{Ref}_q)$$

$$\frac{\Gamma \vdash_q e : \sigma}{\Gamma \vdash_q \mathtt{annot}(e, Q) : Q\ \sigma} \ (\mathrm{Annot}_q)$$

(a) Rules for unqualified types σ

$$\frac{x \in \mathit{dom}(\Gamma)}{\Gamma \vdash_q x : \Gamma(x)} \ (\mathrm{Var}_q)$$

$$\frac{\Gamma \vdash_q e_1 : Q\ (\tau \longrightarrow \tau') \qquad \Gamma \vdash_q e_2 : \tau}{\Gamma \vdash_q e_1\ e_2 : \tau'} \ (\mathrm{App}_q)$$

$$\frac{\Gamma \vdash_q e_1 : \tau_1 \qquad \Gamma[x \mapsto \tau_1] \vdash_q e_2 : \tau_2}{\Gamma \vdash_q \mathtt{let}\, x = e_1\, \mathtt{in}\ e_2 : \tau_2} \ (\mathrm{Let}_q)$$

$$\frac{\Gamma \vdash_q e : Q\ \ \mathit{ref}\ (\tau)}{\Gamma \vdash_q \mathtt{*}e : \tau} \ (\mathrm{Deref}_q)$$

$$\frac{\Gamma \vdash_q e_1 : Q\ \ \mathit{ref}\ (\tau) \qquad \Gamma \vdash_q e_2 : \tau}{\Gamma \vdash_q e_1\, \mathtt{:=}\, e_2 : \tau} \ (\mathrm{Assign}_q)$$

$$\frac{\Gamma \vdash_q e : Q'\ \sigma \qquad Q' \leq Q}{\Gamma \vdash_q \mathtt{check}(e, Q) : Q'\ \sigma} \ (\mathrm{Check}_q)$$

$$\frac{\Gamma \vdash_q e : \tau \qquad \tau \leq \tau'}{\Gamma \vdash_q e : \tau'} \ (\mathrm{Sub}_q)$$

(b) Rules for qualified types τ

Figure 3.5: Qualified Type Checking System

for explaining type checking, in Section 3.4 we incorporate this rule directly into the other rules for inference purposes.

Lemma 3.1 *Let e be a closed term.*

- *If $\emptyset \vdash e : s$, then for any qualifier q we have $\emptyset \vdash_q embed(e, q) : embed(s, q)$.*
- *If $\emptyset \vdash_q e : \tau$, then $\emptyset \vdash strip(e) : strip(\tau)$.*

This lemma formalizes our intuitive requirement that type qualifiers do not affect the underlying type structure.

3.4 Flow-Insensitive Type Qualifier Inference

As described so far, type qualifiers place a rather large burden on programmers wishing to use them: programmers must add explicit qualifier annotations to all constructed terms in their programs. We would like to reduce this burden by performing *type qualifier inference*, which is analogous to standard type inference. As with standard type inference, we introduce *type qualifier variables* $QVar$ to stand for unknown qualifiers that we need to solve for. We write qualifier variables with the Greek letter κ. In the remainder of this dissertation we use *type qualifier constants* to refer to elements of the given qualifier partial order, and we use *type qualifiers* to refer to either a qualifier constant or variable. We define a function $embed'(s)$ that maps standard types to qualified types by inserting fresh type qualifier variables at every level:

$$\begin{array}{rcll} embed'(int) & = & \kappa\ int & \kappa \text{ fresh} \\ embed'(ref\ (s)) & = & \kappa\ ref\ (embed'(s)) & \kappa \text{ fresh} \\ embed'(s \longrightarrow s') & = & \kappa\ (embed'(s) \longrightarrow embed'(s')) & \kappa \text{ fresh} \end{array}$$

The type qualifier inference rules for our source language are shown in Figure 3.6. In this system, we have eliminated qualifier annotations completely. Instead, whenever we assign a type to a term constructor, we introduce a fresh type qualifier variable to stand for the unknown qualifier on the term (see (Int'_q), (Lam'_q), and (Ref'_q)). We use $embed'$ in (Lam'_q) to map the given standard type to a type with fresh qualifier variables. To simplify the rules slightly we use our assumption that the program is correct with respect to the standard types to avoid some shape matching constraints. For example, in (App'_q)

$$\frac{x \in dom(\Gamma)}{\Gamma \vdash'_q x : \Gamma(x)} \ (\mathrm{Var}'_q)$$

$$\frac{\kappa \text{ fresh}}{\Gamma \vdash'_q n : \kappa \ \mathit{int}} \ (\mathrm{Int}'_q)$$

$$\frac{\Gamma[x \mapsto \tau] \vdash'_q e : \tau' \qquad \tau = \mathit{embed}'(s) \qquad \kappa \text{ fresh}}{\Gamma \vdash'_q \lambda x{:}s.e : \kappa \ (\tau \longrightarrow \tau')} \ (\mathrm{Lam}'_q)$$

$$\frac{\Gamma \vdash'_q e_1 : Q \ (\tau \longrightarrow \tau') \qquad \Gamma \vdash'_q e_2 : \tau_2 \qquad \tau_2 \leq \tau}{\Gamma \vdash'_q e_1 \ e_2 : \tau'} \ (\mathrm{App}'_q)$$

$$\frac{\Gamma \vdash'_q e_1 : \tau_1 \qquad \Gamma[x \mapsto \tau_1] \vdash'_q e_2 : \tau_2}{\Gamma \vdash'_q \texttt{let}\, x = e_1 \,\texttt{in}\, e_2 : \tau_2} \ (\mathrm{Let}'_q)$$

$$\frac{\Gamma \vdash'_q e : \tau \qquad \kappa \text{ fresh}}{\Gamma \vdash'_q \texttt{ref}\, e : \kappa \ \mathit{ref}\ (\tau)} \ (\mathrm{Ref}'_q)$$

$$\frac{\Gamma \vdash'_q e : Q \ \mathit{ref}\ (\tau)}{\Gamma \vdash'_q \texttt{*}e : \tau} \ (\mathrm{Deref}'_q)$$

$$\frac{\Gamma \vdash'_q e_1 : Q \ \mathit{ref}\ (\tau) \qquad \Gamma \vdash'_q e_2 : \tau' \qquad \tau' \leq \tau}{\Gamma \vdash'_q e_1 \texttt{:=} e_2 : \tau'} \ (\mathrm{Assign}'_q)$$

$$\frac{\Gamma \vdash'_q e : Q' \ \sigma \qquad Q' \leq Q}{\Gamma \vdash'_q \texttt{check}(e, Q) : Q' \ \sigma} \ (\mathrm{Check}'_q)$$

Figure 3.6: Qualified Type Inference System

$$
\begin{array}{rcl}
C \cup \{Q\ \ int \leq Q'\ \ int\} & \Rightarrow & C \cup \{Q \leq Q'\} \\
C \cup \{Q\ \ ref\ (\tau) \leq Q'\ \ ref\ (\tau')\} & \Rightarrow & C \cup \{Q \leq Q'\} \cup \{\tau \leq \tau'\} \cup \{\tau' \leq \tau\} \\
C \cup \{Q\ (\tau_1 \longrightarrow \tau_2) \leq Q'\ (\tau_1' \longrightarrow \tau_2')\} & \Rightarrow & C \cup \{Q \leq Q'\} \cup \{\tau_1' \leq \tau_1\} \cup \{\tau_2 \leq \tau_2'\}
\end{array}
$$

Figure 3.7: Subtype Constraint Resolution

we know that e_1 has a function type, but we do not know its qualifier, or the qualifiers on its parameter and result types. Finally, instead of having a separate subsumption rule, to make inference syntax-driven we use the standard technique of incorporating (Sub_q) directly into (App'_q) and (Assign'_q).

As we perform type qualifier inference, the rules in Figure 3.6 generate *subtyping constraints* of the form $\tau_1 \leq \tau_2$. Next we apply the rules of Figure 3.7, which are simply the rules of Figure 3.2 written as left-to-right rewrite rules, to reduce the subtyping constraints to *qualifier constraints* among type qualifier constants and variables. Notice that because we assume that the program we are analyzing type checks with respect to the standard types, we know that none of the structural matching cases in Figure 3.2 can fail. The rule (Check'_q) also generates qualifier constraints.

Thus after applying the rules in Figures 3.6 and 3.2, we are left with qualifier constraints of the form $L \leq R$, where L and R are type qualifier constants from Q or type qualifier variables κ. As with the type equality constraints in Section 2.2, we need to solve these qualifier constraints to complete type qualifier inference.

Definition 3.2 *A* solution σ *to a system of qualifier constraints* C *is a mapping from type qualifier variables to type qualifier constants such that for each constraint* $L \leq R$*, we have* $\sigma(L) \leq \sigma(R)$*.*

We write $\sigma \models C$ is σ is a solution to C. Note that there may be many possible solutions to C. There are two solutions in particular that we may be interested in.

Definition 3.3 *If* $\sigma \models C$*, then* σ *is a* least (greatest) solution *if for any other* σ' *such that* $\sigma' \models C$*, for all* $\kappa \in dom(\sigma)$ *we have* $\sigma(\kappa) \leq \sigma'(\kappa)$ *(*$\sigma(\kappa) \geq \sigma'(\kappa)$*).*

Even if C is satisfiable, least and greatest solutions may not always exist for a given partial order on Q. Clearly if C is satisfiable and Q is a meet semilattice then a least solution exists, and similarly if Q is a join semilattice then a greatest solution exists.

```
QUAL-SOLVE(C) =
  for all κ ∈ C do S(κ) ← Q
  for all q ∈ Q do S(q) ← {q}
  let C' = C
  while C' ≠ ∅ do
    remove an L ≤ R from C'
    let S'_L = S(L) ∩ ↓ S(R)
    let S'_R = S(R) ∩ ↑ S(L)
    if S'_L = ∅ or S'_R = ∅
      then return unsatisfiable
    if S(L) ≠ S'_L
      then S(L) ← S'_L
        Add each L' ≤ L and L ≤ R' in C to C'
    if S(R) ≠ S'_R
      then S(R) ← S'_R
        Add each L' ≤ R and R ≤ R' in C to C'
  return S
```

Figure 3.8: Qualifier Constraint Solving

A system of qualifier constraints is also known as an *atomic subtyping constraint system*, and there are well-known algorithms for solving such constraints efficiently if Q is a semilattice [93]. In general, solving atomic subtyping constraints over an arbitrary partial order is NP-hard, even with fixed Q [91]. Here we present a simple algorithm that works for semilattices, discrete partial orders, and arbitrary cross products of those.

Figure 3.8 gives our algorithm. The function QUAL-SOLVE(C) takes as input a system of qualifier constraints. It either returns **unsatisfiable** or it returns a mapping $S : QVar \to 2^Q$ that captures, in the sense described below, the possible solutions of C if Q is a semilattice, a discrete partial order, or a cross product of those. For any partial order the algorithm is sound, as shown in the following lemma:

Lemma 3.4 *Let* $S =$ QUAL-SOLVE(C). *Then for any* σ *such that* $\sigma \models C$ *and for any qualifier variable* $\kappa \in C$, *we have* $\sigma(\kappa) \in S(\kappa)$.

Proof: We show that this property is preserved by each step of the algorithm. Clearly it holds before the loop since $S(\kappa) = Q$ initially for all qualifier variables κ. So suppose that this property holds and we execute one step of the loop iteration. Let $L \leq R$ be the constraint we remove from C'. By assumption, $\sigma(L) \in S(L)$ and $\sigma(R) \in S(R)$ (where

we set $\sigma(q) = q$ for $q \in Q$). Then since $\sigma \models C$, we must have $\sigma(L) \leq \sigma(R)$. But then $\sigma(L) \in\downarrow \sigma(R)$ and $\sigma(R) \in\uparrow \sigma(L)$. Thus $\sigma(L) \in S'_L$ and $\sigma(R) \in S'_R$, and therefore the property holds after this iteration. □

Corollary 3.5 *If* QUAL-SOLVE(C) = **unsatisfiable**, *then C has no solution.*

Proof: Suppose for a contradiction that $\sigma \models C$. Then by Lemma 3.4, for all κ we have $\sigma(\kappa) \in$ QUAL-SOLVE(C). But since the algorithm returned **unsatisfiable**, there must be some κ such that $S(\kappa) = \emptyset$, a contradiction. □

If the algorithm returns **unsatisfiable**, then, no solution to the constraints exists. As mentioned above, solving atomic subtyping constraints is NP-hard; thus there are cases when the algorithm does not discover that the constraints are unsatisfiable even though they are. But we can prove that our algorithm is complete if Q is a semilattice, discrete partial order, or a cross product of those.

First we define a non-standard term to capture a key property of our algorithm.

Definition 3.6 *Let $(S, \leq)$ be a partial order. A set $S' \subseteq S$ is an* interval *if $x, z \in S'$ and $x \leq y \leq z$ implies $y \in S'$.*

Lemma 3.7 *If S_1 and S_2 are intervals, then $S_1 \cap S_2$ is an interval.*

Lemma 3.8 *The sets $S(\kappa)$ computed by* QUAL-SOLVE(C) *are intervals.*

Proof: This clearly holds at the beginning of the algorithm. Since $\downarrow S(R)$ and $\uparrow S(L)$ are intervals, this holds after each step of the algorithm by Lemma 3.7. □

Lemma 3.9 *Let $(S, \leq)$ be a finite meet (join) semilattice and let $S_1, S_2 \subseteq S$ be intervals and contain their least (greatest) elements. Suppose that $S_1 \cap S_2 \neq \emptyset$. Then $S_1 \cap S_2$ contains its least (greatest) element.*

Proof: We show this for a meet semilattice; the other case is similar. Let $s_1 = \bigsqcap S_1$, $s_2 = \bigsqcap S_2$, and $s = \bigsqcap(S_1 \cap S_2)$. By definition of $\sqcap$ we have $s_1 \leq s$ and $s_2 \leq s$. So pick a $q \in S_1 \cap S_2$, which exists by assumption. Then $q \in S_1$ and $q \in S_2$, so $s_1 \leq q$ and $s_2 \leq q$. But then by continuity $s \in S_1$ and $s \in S_2$, thus $s \in S_1 \cap S_2$. □

Lemma 3.10 *Suppose the elements of Q are in a semilattice, and let $S =$* QUAL-SOLVE(C). *Then if $S \neq$* **unsatisfiable**, *there exists σ such that $\sigma \models C$.*

Proof: Suppose that Q is a meet semilattice. Since greatest lower bounds always exist, we can view $S(\kappa)$ as an alternate representation of a solution σ where $\sigma(\kappa) = \bigsqcap S(\kappa)$. To make the mapping clear, observe that $\sigma(\kappa) \in S(\kappa)$ at every step of the algorithm. This clearly holds initially, and since $\downarrow S(R)$ and $\uparrow S(L)$ are intervals and contain their least elements, by Lemma 3.9 it holds after each iteration.

Further, pick any $q \in S(L) \cap \downarrow S(R)$ from the execution of the algorithm in Figure 3.8 and consider one additional step of iteration. Then $\bigsqcap S(L) \leq q$, hence $\bigsqcap S(L) \in \downarrow S(R)$. Thus $\bigsqcap S(L) \in S'_L$, i.e., $\bigsqcap S'_L \leq \bigsqcap S(L)$. But we already know $\bigsqcap S(L) \leq \bigsqcap S'_L$, and therefore $\bigsqcap S(L) = \bigsqcap S'_L$, i.e., intersecting with $\downarrow S(R)$ does not change the least solution.

For the other intersection, $S'_R = S(R) \cap \uparrow S(L)$, we know that $\bigsqcap S(R) \leq \bigsqcap S'_R$ and $\bigsqcap(\uparrow S(L)) \leq \bigsqcap S'_R$. But $\bigsqcap(\uparrow S(L)) = \bigsqcap S(L)$. Thus $\bigsqcap S'_R$ is an upper bound of $\bigsqcap S(L)$ and $\bigsqcap S(R)$, our solutions for L and R. Thus intersecting with $\uparrow S(L)$ increases the least solution of R to account for having L as a lower bound.

Thus we see that QUAL-SOLVE is just the standard least-fixpoint algorithm, and $\sigma \models C$. The case when Q is a join semilattice is similar, and in that case we can view QUAL-SOLVE as computing the greatest solution. □

Lemma 3.11 *Suppose the elements of Q are in the discrete partial order, and let $S =$* QUAL-SOLVE(C)*. Then if $S \neq$* **unsatisfiable***, there exists σ such that $\sigma \models C$.*

Proof: If Q is in the discrete partial order, the directionality of the constraints does not matter. Observe that at every step of the algorithm, $S(\kappa)$ is either Q or contains a single element. Clearly this holds initially and after each iteration of the algorithm, since $\uparrow q = \downarrow q = \{q\}$ for any partial order element q. Thus we can view QUAL-SOLVE as computing the standard most general solution of a set of equality constraints. □

Lemma 3.12 *Suppose that the partial order on Q is a cross product of semilattices and the discrete partial order, and let $S =$* QUAL-SOLVE(C)*. Then if $S \neq$* **unsatisfiable***, there exists σ such that $\sigma \models C$.*

Proof: Let $Q = Q_1 \times \cdots Q_n$. We can view the set $S(\kappa)$ computed by the algorithm is a tuple of solution sets $S_1(\kappa) \times \cdots \times S_n(\kappa)$. Then we can imagine we have n copies of the constraints C, and by Lemmas 3.10 and 3.11 and our assumptions we can compute a solution σ_i from each S_i for each component of the tuple. □

$$\frac{l \in dom(S)}{S \vdash_q (l, Q) \to (l, Q); S} \text{ [Var}_q\text{]}$$

$$\frac{}{S \vdash_q \texttt{annot}(n, Q) \to (n, Q); S} \text{ [Int]}$$

$$\frac{}{S \vdash_q \texttt{annot}(\lambda x\!:\!s.e, Q) \to (\lambda x\!:\!s.e, Q); S} \text{ [Lam]}$$

$$\frac{\begin{array}{c} S \vdash_q e_1 \to (\lambda x.e, Q); S' \qquad S' \vdash_q e_2 \to (v', Q'); S'' \\ S'' \vdash_q e[x \mapsto (v', Q')] \to (v'', Q''); S''' \end{array}}{S \vdash_q e_1\ e_2 \to (v'', Q''); S'''} \text{ [App]}$$

$$\frac{S \vdash_q e_1 \to (v, Q); S' \qquad S' \vdash_q e_2[x \mapsto (v, Q)] \to (v', Q'); S''}{S \vdash_q \texttt{let}\, x = e_1\, \texttt{in}\, e_2 \to (v', Q'); S''} \text{ [Let]}$$

$$\frac{S \vdash_q e \to (v, Q); S' \qquad l \notin dom(S')}{S \vdash_q \texttt{annot}(\texttt{ref}\, e, Q') \to (l, Q'); S'[l \mapsto (v, Q)]} \text{ [Ref]}$$

$$\frac{S \vdash_q e \to (l, Q); S' \qquad l \in dom(S')}{S \vdash_q *e \to S'(l); S'} \text{ [Deref]}$$

$$\frac{S \vdash_q e_1 \to (l, Q); S' \qquad S' \vdash_q e_2 \to (v, Q'); S'' \qquad l \in dom(S'')}{S \vdash_q e_1 := e_2 \to (v, Q'); S''[l \mapsto (v, Q')]} \text{ [Assign]}$$

$$\frac{S \vdash_q e \to (v, Q'); S' \qquad Q' \leq Q}{S \vdash_q \texttt{check}(e, Q) \to (v, Q'); S'} \text{ [Check]}$$

Figure 3.9: Big-Step Operational Semantics with Qualifiers

Given a system of constraints C of size n and a fixed set of k qualifiers, the algorithm in Figure 3.8 runs in $O(n2^k)$ time. To see this, observe that each constraint $L \leq R$ can only be added back to C' if the solution of L or of R changes. Since $S(\kappa)$ decreases monotonically for all κ, we can only change $S(\kappa)$ at most 2^k times. Thus we can only add a constraint back into C' a total of $2 \cdot 2^k$ times. Since we assume k is a small constant, the whole algorithm runs in time $O(n)$.

3.5 Semantics and Soundness

As with the standard type system for our language, we can prove that our qualified type system is sound under a natural semantics for type qualifiers. Figure 3.9 gives our semantic reduction rules. In this semantics, values are simply standard values paired with uninterpreted qualifiers. The rules are identical to the standard semantic rules of Figure 2.2, except that locations, integers, and function values are paired with qualifiers, and that deconstruction steps throw away the outermost qualifier. Rule [Check] tests the top-level qualifier of a value.

A full proof of soundness, using standard techniques, can be found in Appendix A. Here we simply state our soundness theorem, where r, a reduction result, is either a value pair (v, Q) or **err**.

Theorem 3.13 *If* $\emptyset \vdash_q e : \tau$ *and* $\emptyset \vdash_q e \to r; S'$*, then* r *is not* ***err****.*

3.6 Subtyping Under Non-Writable Pointer Types

As discussed in Section 3.1, we use a conservative rule ($\text{Ref}_{\leq}$) for pointer subtyping: the constraint $\mathit{ref}\,(\tau) \leq \mathit{ref}\,(\tau')$ is satisfiable only if $\tau = \tau'$. This rule can often lead to non-intuitive "backward" qualifier propagation. For example, consider the following code:

$$\begin{array}{l}\texttt{let } f = \lambda x\colon \mathit{ref}\,(\mathit{int}).\,\texttt{*}x \texttt{ in}\\ \quad f\ y;\\ \quad f\ z\end{array}$$

Ignoring the outermost qualifier, inference assigns the domain of f type $\mathit{ref}\,(\kappa\ \ \mathit{int})$. Assume that the types of y and z are $\mathit{ref}\,(\kappa'\ \ \mathit{int})$ and $\mathit{ref}\,(\kappa''\ \ \mathit{int})$, respectively. Then by ($\text{Ref}_{\leq}$), the first application requires $\kappa' = \kappa$, and the second application requires $\kappa'' = \kappa$. Putting the two together yields the rather counter-intuitive $\kappa' = \kappa''$. In other words, y's qualifier κ' is propagated from x into f and then backward to κ'' and z.

Notice, however, that f does not write through its parameter x. Therefore y and z cannot be modified by f, and we can soundly weaken our constraints to $\kappa' \leq \kappa$ and $\kappa'' \leq \kappa$. Think of an updatable reference x containing data of type τ_x as an object with two methods $\mathit{get}_x : \mathit{void} \longrightarrow \tau_x$ and $\mathit{set}_x : \tau_x \longrightarrow \mathit{void}$ to read and write the reference, respectively. Here

$$\frac{Q \leq Q' \quad \tau \leq \tau' \quad Q'\ ref\ (\tau') \text{ cannot be updated}}{Q\ ref\ (\tau) \leq Q'\ ref\ (\tau')}\ (\text{Ref}'_{\leq})$$

Figure 3.10: Subtyping Non-Writable References

void is a placeholder meaning "no parameter" or "no result." Notice that τ_x appears both co- and contravariantly (on the left and right sides of the function arrow). When we apply f to y in the above code, we generate two constraints:

$$\begin{array}{lcll} void \longrightarrow \tau_y & \leq & void \longrightarrow \tau_x & (1)\ get \text{ compatibility} \\ \tau_y \longrightarrow void & \leq & \tau_x \longrightarrow void & (2)\ set \text{ compatibility} \end{array}$$

These constraints require that the *get* and *set* methods of y and of f's parameter x be compatible. By ($\text{Fun}_{\leq}$) the constraint (1) yields $\tau_y \leq \tau_x$ and (2) yields $\tau_x \leq \tau_y$, which put together produce $\tau_y = \tau_x$, which is exactly what ($\text{Ref}_{\leq}$) requires. But if f does not write through x, then intuitively x does not have a *set* method. Thus by standard width subtyping in object-oriented type systems we do not generate constraint (2), and the result is that we only require $\tau_y \leq \tau_x$.

Thus we can use a new subtyping rule for references, as shown in Figure 3.10. We refer to this as *deep subtyping*. There are a number of techniques for checking whether a particular name is used to write to an updatable reference. In Section 5.2 we describe the approach used in our implementation.

3.7 Related Work

In this section we discuss work related to the basic concepts of type qualifiers. We delay discussion of most of the related program analysis systems and tools until Section 5.6. The flow-insensitive type qualifier system presented here was previously described by us [43].

Specific examples of flow-insensitive type qualifiers have been proposed to solve a number of problems. ANSI C contains the type qualifier *const* [6], discussed further in Section 6.1. Binding-time analysis [30] can be viewed as associating one of two qualifiers with expressions, either *static* for expressions that may be computed at compile time or *dynamic* for expressions not computed until run-time. The Titanium programming language

[124] uses qualifiers *local* and *global* to distinguish data located on the current processor from data that may be located at a remote node [73]. Solberg [106] gives a framework for understanding a particular family of related analyses as type annotation (qualifier) systems.

Several related techniques have been proposed for using qualifier-like annotations to address security issues. A major topic of recent interest is secure information flow [114], which associates *high* and *low* security levels with expressions and tries to prevent high-security data from "leaking" to low-security outputs. Other examples of security-related annotation systems are lambda calculus with trust annotations [87] and Java security checking [103]. For a discussion of a related technique using our framework, see Section 6.2.

Type qualifiers, like any type system, can be seen as a form of abstract interpretation [19]. Flow-insensitive type qualifiers can be viewed as a label flow system [80] in which we place constraints on where labels may flow. Control-flow analysis [102] is a label flow system in which labels decorate only functions. We believe that recent efficient techniques for polymorphic recursive label flow inference [37, 92] can be applied to flow-insensitive type qualifiers. Type qualifiers can also be viewed as refinement types [48], which have the same basic property: refinement types do not change the underlying type structure. The key difference between qualifiers and refinement types is that the latter is based on the theory of intersection types, which is significantly more complex than atomic subtyping. Refinement types also are not flow-sensitive (see Chapter 4).

Chapter 4

Flow-Sensitive Type Qualifiers and Restrict

In the discussion so far, type qualifiers, like the standard types, are *flow-insensitive*, meaning that qualifiers do not change from one program point to the next. For example, consider an assignment to x:

$$/^*\ x : \tau\ ^*/\ x := e\ /^*\ x : \tau\ ^*/$$

Notice that x has the same type—and the same qualifiers, since types τ contain qualifiers—before and after the assignment. While a flow-insensitive system is natural for many problems and useful in practice (see Chapter 6), many important program properties are *flow-sensitive*. Checking such properties requires associating different facts—in our system, different qualifiers—with a value at different program points.

In this chapter, we present monomorphic systems for flow-sensitive type qualifier checking and inference. With one exception our syntax and semantics are the same as in Chapter 3, but our type system is enhanced to track qualifiers more precisely across state changes. The one exception is `restrict`, a new language construct that can be used to enhance the precision of our flow-sensitive type qualifier system (Section 4.2). As in the previous chapter, type qualifiers do not affect the underlying type structure. This choice is critical for making a scalable flow-sensitive inference algorithm (Section 4.5).

As before, we assume that our input programs are annotated with standard types s and type check with respect to those types. In our implementation (Chapter 5) we support

```
fun f w =
  let x = ref 0
      y = ref annot(1, a)
      z = ref annot(2, b)
   in
      x := 3;
      w := 4;
      y := annot(5, c);
      if (···) {
        f z
      };
      check(*y, c)
```

Figure 4.1: Example Program

both flow-insensitive and flow-sensitive type qualifiers simultaneously, but in order to avoid confusion we assume in this chapter that all type qualifiers of interest are flow-sensitive.

Example 1. Figure 4.1 shows an example program we would like to check with our flow-sensitive type qualifier system. Here we use some syntactic sugar; for example, we write a recursive function in the natural way instead of using the primitive `Y` combinator, and we write `if` directly instead of encoding it with functions. In this example the qualifier constants a, b, and c are in the discrete partial order (they are incomparable). Just before f returns, we wish to check that y has the qualifier c. This check succeeds only if we can model the update to y as a strong update. The qualifiers on x and z will be used to demonstrate other features of our system. □

4.1 Designing a Flow-Sensitive Type Qualifier System

In this section, we give an informal overview of how our flow-sensitive type qualifier system works and what design choices went into it. Since we expect programmers to interact with our system, both when adding and when reviewing the results of inference (Section 4.5), we consciously seek a system that supports efficient inference and is straightforward for a programmer to understand and use.

4.1.1 Abstract Stores, Abstract Locations, and Linearities

In order to model type qualifiers flow-sensitively, we need to be able to talk about the qualifiers at a particular program point. In our system, we model the state at a particular point using an *abstract store*, which associates a qualified type with each location in the program. As locations are updated, we update their qualifiers in the abstract store to reflect their new state. For example, suppose that x, y, and z are updatable references (typically called *variables* in an imperative language such as C or Java), and assume for a moment that those are the only locations in the program. Then we can choose as abstract stores a mapping from x, y, and z to their qualified types:

$$\begin{gathered}\{x : q\ \ int,\ y : r\ \ int,\ z : s\ \ int\} \\ x := \texttt{annot}(e, q') \\ \{x : q'\ \ int,\ y : r\ \ int,\ z : s\ \ int\} \\ y := \texttt{annot}(e, r') \\ \{x : q'\ \ int,\ y : r'\ \ int,\ z : s\ \ int\}\end{gathered}$$

This is the approach taken by classical dataflow analysis [3, 67], which focuses on *intraprocedural* analysis (analyzing one function body at a time) of languages like FORTRAN. For such an analysis there is only a small, fixed set of locations, and *aliasing* can often be modeled very conservatively (for example, function calls may change any non-local variable) without greatly compromising the effectiveness of the analysis [3]. Aliasing occurs when there are multiple names for the same object—for example, via pointers or by-reference parameter passing. Because we want to perform *inter*procedural checking (modeling more than one function body at once) of languages such as C, C++, Java, and ML, where pointers and indirection are very common, we need to handle aliasing in a more sophisticated way. For example, suppose that p is a pointer to updatable reference x. Consider the following code (the outermost qualifier on p has been omitted):

$$\begin{gathered}\{x : q\ \ int,\ p : ref\ (q\ \ int),\ \ldots\} \\ *p := \texttt{annot}(e, q') \\ \{x :?\ \ int,\ p : ref\ (q'\ \ int),\ \ldots\}\end{gathered}$$

What happens when we indirectly update x through p? We need to know that both $*p$ and x are changed, and so far this information is not encoded in abstract stores.

Our solution is to introduce another level of indirection into abstract stores. Instead of program names, in our system abstract stores map *abstract locations* ρ to qualified

types. Intuitively, two expressions that evaluate to the same run-time location are assigned the same abstract location. Instead of types *ref* (τ) (again, omitting the qualifier on the *ref*), we use pointer types of the form *ref* (ρ), where ρ is the pointed-to location. Reads and writes to an object of type *ref* (ρ) access location ρ in the current abstract store. Our example above becomes

$$\{\rho : q\ \ int,\ \nu : ref\ (\rho),\ \ldots\}$$
$$*p := \texttt{annot}(e, q')$$
$$\{\rho : q'\ \ int,\ \nu : ref\ (\rho),\ \ldots\}$$

The write through $*p$ updates ρ, the location of x, and both $*p$ and x have the same qualified type.

To apply this idea of mapping abstract locations to types, we need a way to compute abstract locations and assign them to expressions. There are several issues in doing so. First, the program may have an unbounded number of run-time locations. For example, it may have a recursive function with a local variable, or it may use a data structure. Thus we need some way to represent all possible run-time locations in finite space. Second, determining whether two objects evaluate to the same location is undecidable [60]. Thus our assignment of abstract locations to expressions must be conservative. Finally, because our abstract location assignment is conservative, we may not always be able to track updates precisely.

The process of assigning abstract locations to expressions is a form of *alias analysis* [17, 69]. There are two basic kinds of aliasing information we can compute. If two expressions could evaluate to the same run-time location, then we say they *may alias*. If two expressions always evaluate to the same run-time location, then they *must alias*. Usually may alias information is used in the negative sense: if it's not the case that two expressions may alias, then we know they evaluate to different locations.

In practice, we need both may and must alias information to model flow-sensitive type qualifiers. In our system we encode may alias information in abstract locations—two expressions have the same abstract location if they may alias. In Section 4.3 we describe checking and inference systems for computing flow-insensitive may alias information. We choose flow-insensitivity for the abstract location computation to make inference as a whole efficient. We associate a *linearity* [21, 112] with each abstract location to encode must alias information. Informally, we say that a location ρ is *linear*, which we write ρ^1, if it corresponds to exactly one run-time location. Otherwise, ρ is *non-linear*, written ρ^ω. The

key feature of linear locations is that two expressions that refer to the same linear location ρ^1 must alias, since they always refer to run-time location ρ and there is only one of those. On the other hand, two expressions assigned the same non-linear location ρ^ω by alias analysis may—but not necessarily must—alias, since ρ may stand for multiple run-time locations. The linearities 1 and ω are ordered, with $1 < \omega$. Intuitively the ordering corresponds to the fact that two locations that must alias also may alias, but not vice-versa.

When we model updates to a location, we treat linear and non-linear locations differently. When a linear location is updated, we can track the update precisely, since we know which run-time location is changing. Suppose that p has type *ref* (ρ) and location ρ is linear. Consider the following code:

$$\{\rho^1 : q \ \ int, \ \ldots\}$$
$$*p := \texttt{annot}(e, q')$$
$$\{\rho^1 : q' \ \ int, \ \ldots\}$$

Here we have performed a *strong update* [17] on ρ, replacing its qualifier after the assignment, because we know precisely which location was updated. On the other hand, suppose ρ is non-linear. Then ρ may stand for multiple locations, but the assignment $*p :=$ ... only updates a single one of them. Thus we need to be conservative:

$$\{\rho^\omega : q \ \ int, \ \ldots\}$$
$$*p := \texttt{annot}(e, q')$$
$$\{\rho^\omega : q \sqcup q' \ \ int, \ \ldots\}$$

After the assignment, ρ refers to both the run-time location that was updated, which has qualifier q', and the run-time locations that were not updated, which still have qualifier q. Thus we conservatively say that ρ's qualifier is either q or q', which we represent with the least upper bound operator (Section 2.3). This is called a *weak update*.

In our system we model linearities flow-sensitively. Thus in their final form, abstract stores are mappings from abstract locations to types and linearities, which we write as follows:

$$\{\rho^1 : q \ \ int, \ \nu^\omega : r \ \ int, \ \ldots\}$$

As it turns out, while these abstract stores are useful for describing our system, they are an inefficient representation for type qualifier inference. In Section 4.5 we present a constructor-based formalism for describing such stores compactly and efficiently.

$$\begin{array}{cc} \{\rho^1 : \mathsf{a}\ \ int,\ \ldots\} & \{\rho^\omega : \mathsf{b}\ \ int,\ \ldots\} \\ f\ () & f\ () \\ \{\rho^\omega : \mathsf{a} \sqcup \mathsf{b}\ \ int\ \ldots\} & \{\rho^\omega : \mathsf{a} \sqcup \mathsf{b}\ \ int\ \ldots\} \end{array}$$

(a) Two calls to f with no effect information

$$\begin{array}{cc} \{\rho^1 : \mathsf{a}\ \ int,\ \ldots\} & \{\rho^\omega : \mathsf{b}\ \ int,\ \ldots\} \\ f\ () & f\ () \\ \{\rho^1 : \mathsf{a}\ \ int\ \ldots\} & \{\rho^\omega : \mathsf{b}\ \ int\ \ldots\} \end{array}$$

(b) Result if f has no effect on location ρ

Figure 4.2: Using Effects at Function Calls

4.1.2 Effects

Abstract stores represent the state at a particular program point. To model functions, we add abstract stores to their types to represent the state at the beginning and end of the function. We also add an *effect* [51, 74, 75, 121] to function types to capture all possible reads, writes, and allocations that may happen when a function is called.

We can use the effect of a function to improve the precision of our flow-sensitive qualifier system. Recall that the system we describe is monomorphic, meaning that all calls to a function share the same initial and final stores. For example, consider the state of location ρ after two distinct calls to f in the same program, shown in Figure 4.2a. Before the call on the left, ρ is linear and has qualifier a, and before the call on the right, ρ is non-linear and has qualifier b. But since f is monomorphic there is only one store representing the state following f, and in that store location ρ may have qualifier a or qualifier b. Moreover, in that store ρ must be non-linear, since ρ is non-linear before one of the two calls to f.

The most general solution to this problem is to introduce polymorphism over stores [21, 104]. Instead, we choose a simpler solution: we use effects to gain some of the benefits of polymorphism without the added complexity.

Observe that if f does not use location ρ, then we need not merge the qualifiers and linearity of ρ after the calls to f. Intuitively, we can simply flow the qualifiers and linearity of ρ "around" the calls to f, as shown in Figure 4.2b. Using effects this way makes functions fully polymorphic in locations they do not use. We can even do slightly better—if

f reads or allocates ρ but does not write ρ, then we can flow the qualifiers of ρ around calls to f. If f does not allocate ρ, we can flow the linearity of ρ around calls to f. We formalize these ideas in Section 4.4.

4.2 Restrict

Finally, the system described so far has a serious practical weakness. Type checking may fail because a location on which a strong update is needed may be non-linear. This is especially problematic for data structures, since our may alias analysis for computing abstract locations (Section 4.3) tends to conflate different elements of the same data structure. We address this issue by adding a new language construct $\texttt{restrict}\, x = e\, \texttt{in}\, e'$, which is inspired by the ANSI C type qualifier of the same name [6]. ANSI C's `restrict` qualifier, whose use is not checked for correctness, is designed to enable a compiler to optimize code more aggressively. In our system, `restrict` is used as a tool to document and check aliasing properties of the program, increasing the precision of flow-sensitive type qualifier checking. For a complete discussion of the relation of our system for `restrict` to ANSI C's type qualifier, see Section 5.5.

In our system, the construct $\texttt{restrict}\, x = e\, \texttt{in}\, e'$ binds x to the value of e, which must be a pointer, during evaluation of e'. The contract `restrict` enforces is that during evaluation of e_2, the only access to the object x points to is through the name x or through copies of x. We sometimes informally refer to x as a restricted pointer.

Example 2. Consider the following code:

```
restrict p = q in
  *p;      /* valid */
  *q;      /* invalid */
```

Here we may access the object p points to by dereferencing p but we may not access it via q. □

Example 3. The following code shows how restricted pointers can be passed from outer to inner scopes:

```
restrict p = ... in
  restrict r = p in {
    *r;      /* valid */
    *p;      /* invalid */
  }
  *p         /* valid */
```

We have added braces to make the grouping clear. In this example, within the scope of r we can dereference r but not p. When r goes out of scope we recover the ability to access p. □

Example 4. Consider the following code:

```
restrict p = ... in
  let r = ref p in
    *r := ...      /* valid */
```

Notice that the restricted pointer p is actually written to memory. In our system we may store and retrieve the value of p from memory as long as that memory does not escape the scope of p. Although as mentioned above the C standard contains a construct similar to our `restrict`, this particular example is not allowed in the standard [6]; see Section 5.5. □

We can use `restrict` to locally regain strong updates of flow-sensitive type qualifiers. Consider an instance of `restrict` $x = e$ `in` e', and suppose that expression e points to location ρ. Then because of the semantics of `restrict`, we know that out of all the objects location ρ may refer to, only one of them can be used within e', namely the one x is dynamically initialized to. Thus we give x a fresh abstract location ρ', and we can allow ρ' to be linear even if ρ is non-linear. Only when the scope of the `restrict` ends do we need to merge the state of ρ' with the state of ρ, which may require a weak update to ρ.

Example 5. Consider the following code to acquire and release a lock from a data structure (here $a[i]$ reads the ith element of array a and $x.f$ accesses field f of x):

```
restrict l = a[i].lock in
  lock(l)
  ...
  unlock(l)
```

In our implementation of type qualifiers for C (Chapter 5), we assign all elements of an array the same type and the same location. Thus the location ρ that l points to is non-linear. If l were bound with `let`, then we could not strongly update the state of l when the lock is acquired and released. But since l is bound with `restrict`, it can be assigned a fresh *linear* location ρ' (assuming the ... does not alias ρ' with other locations). Since ρ' is linear, we can track precisely that l is locked after it is acquired and unlocked at the end of the block. When the scope ends, we merge the state of ρ' back with the state of ρ. In this way we can check that this code adheres to a standard locking protocol (for example, `lock` is never called twice in a row on the same lock). □

In Section 4.3 we discuss `restrict` in more depth and show how to use effects to enforce the correctness of `restrict` expressions.

4.3 Aliasing, Effects, and Restrict

Now that we have described our flow-sensitive type qualifier framework at a high level, we begin developing the system more formally. Our type system is divided into two stages, a preliminary flow-insensitive step followed by flow-sensitive qualifier checking. The first, flow-insensitive step computes aliasing information and effects and checks `restrict`. In practice, this stage is combined with checking flow-insensitive type qualifiers, though we omit that here to avoid confusion. The second, flow-sensitive step uses the information from the first stage to build stores modeling the qualifiers and linearities at each program point.

In order to conveniently transmit information from the first stage to the second, we present the first stage as a translation system from unannotated programs to programs with location and effect annotations, shown in Figure 4.3. The target language extends the source language in two ways. First, every allocation site $\texttt{ref}^{\rho} e$ is annotated with the abstract location ρ that is allocated, and similarly each `restrict` site $\texttt{restrict}^{\rho} x = e_1 \texttt{ in } e_2$ is annotated with the abstract location that x is bound to (see below). Second, each function

$$
\begin{array}{rcl}
e & ::= & v \mid e_1\ e_2 \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid \texttt{ref}\ e \mid \texttt{*}e \mid e_1\ \texttt{:=}\ e_2 \\
& \mid & \texttt{annot}(e, Q) \mid \texttt{check}(e, Q) \mid \texttt{restrict}\ x = e_1\ \texttt{in}\ e_2 \\
v & ::= & x \mid n \mid \lambda x{:}s.e \\
s & ::= & \mathit{int} \mid \mathit{ref}\ (s) \mid s \longrightarrow s'
\end{array}
$$

(a) Source Language

$$
\begin{array}{rcl}
e & ::= & v \mid e_1\ e_2 \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid \texttt{ref}^{\rho}\ e \mid \texttt{*}e \mid e_1\ \texttt{:=}\ e_2 \\
& \mid & \texttt{annot}(e, Q) \mid \texttt{check}(e, Q) \mid \texttt{restrict}^{\rho}\ x = e_1\ \texttt{in}\ e_2 \\
v & ::= & x \mid n \mid \lambda^{L} x{:}t.e \\
L & ::= & \emptyset \mid \rho \mid \mathit{rd}(\rho) \mid \mathit{wr}(\rho) \mid \mathit{al}(\rho) \mid L_1 \cup L_2 \mid L_1 \cap L_2 \mid L_1 - \rho \\
t & ::= & \mathit{int} \mid \mathit{ref}\ (\rho) \mid t \longrightarrow^{L} t'
\end{array}
$$

(b) Target Language

Figure 4.3: Source Language and Target Language with Location and Effect Annotations

$\lambda^L x{:}t.e$ is annotated with the type t of its parameter and the effect L of calling the function. Effects are sets of three basic effects: reading $\mathit{rd}(\rho)$, writing $\mathit{wr}(\rho)$, and allocation $\mathit{al}(\rho)$. The effect ρ is shorthand for $\mathit{rd}(\rho) \cup \mathit{wr}(\rho) \cup \mathit{al}(\rho)$. When we write $\rho \notin L$ (see below), we mean $\mathit{rd}(\rho) \notin L$, $\mathit{wr}(\rho) \notin L$, and $\mathit{al}(\rho) \notin L$.

Integer types are as before, and function types contain the effect of calling the function. Notice that there are no qualifiers—in this system, all qualifiers are flow-sensitive, hence they are ignored during this first stage.

Intuitively, the fundamental difference between a flow-sensitive type system and a flow-insensitive type system is the choice between having a single, global model of the store and having a per-program point model of the store. To emphasize this distinction, here we write pointer types as $\mathit{ref}\ (\rho)$, and we maintain a single global abstract store S_I mapping locations ρ to types. If $S_I(\rho) = t$, then location ρ contains data of type t. In contrast, in the second, flow-sensitive stage of the algorithm, we use a per-program point model of the store.

We define a function $\mathit{strip}(\cdot)$ from types with locations and effects to standard types:

$$
\begin{array}{rcl}
\mathit{strip}(\mathit{int}) & = & \mathit{int} \\
\mathit{strip}(\mathit{ref}\ (\rho)) & = & \mathit{ref}\ (\mathit{strip}(S_I(\rho))) \\
\mathit{strip}(t \longrightarrow^{L} t') & = & \mathit{strip}(t) \longrightarrow \mathit{strip}(t')
\end{array}
$$

As with the previous type systems, we present both a system for checking whether a translation into our target language is correct and an inference system for constructing a correct translation from a bare program.

4.3.1 A Flow-Insensitive Checking System

Figure 4.4 presents our system for checking a translation while simultaneously checking the correctness of uses of `restrict`. This system proves judgments of the form $\Gamma \vdash e \Rightarrow e' : t; L$, meaning that in type environment Γ, expression e translates to annotated expression e' and has type t, and evaluating e has effect L. We define the set of locations and effects appearing in a type t as

$$\begin{aligned} \mathit{loceff}(\mathit{int}) &= \emptyset \\ \mathit{loceff}(\mathit{ref}\ (\rho)) &= \rho \cup \mathit{loceff}(S_I(\rho)) \\ \mathit{loceff}(t_1 \longrightarrow^L t_2) &= L \cup \mathit{loceff}(t_1) \cup \mathit{loceff}(t_2) \end{aligned}$$

We define $\mathit{loceff}(\Gamma) = \bigcup_{[x \mapsto t] \in \Gamma} \mathit{loceff}(t)$.

We discuss the rules the Figure 4.4.

- (Var_a) and (Int_a) translate variables and integers to themselves. Evaluating a variable or an integer has no effect—recall that in lambda calculus, a variable is an r-value, not an l-value.

- (Lam_a) translates a function by annotating it with the effect L of evaluating its body e and with the type t of its parameter. The type t must have the same shape as the specified standard parameter type s. Notice that L is added to the function type, and that the evaluation of the function definition itself has no effect, since the function does not execute until it is actually called.

- (App_a) translates an application, and the effect of evaluating an application is the union of the effect of evaluating e_1, the effect of evaluating e_2, and the effect of calling the function e_1. Notice here that the type of e_1's domain and the type of e_2 must match. Since those types may contain abstract locations, this rules enforces our aliasing requirement. The formal parameter x and the actual parameter e_2 may represent the same location, so they must contain identical abstract locations.

- (Let_a) translates a let binding.

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

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

$$\frac{\Gamma[x \mapsto t] \vdash e \Rightarrow e' : t'; L \qquad strip(t) = s}{\Gamma \vdash \lambda x{:}s.e \Rightarrow \lambda^L x{:}t.e' : t \longrightarrow^L t'; \emptyset} \ (\mathrm{Lam}_a)$$

$$\frac{\Gamma \vdash e_1 \Rightarrow e_1' : t \longrightarrow^L t'; L_1 \qquad \Gamma \vdash e_2 \Rightarrow e_2' : t; L_2}{\Gamma \vdash e_1\ e_2 \Rightarrow e_1'\ e_2' : t'; L_1 \cup L_2 \cup L} \ (\mathrm{App}_a)$$

$$\frac{\Gamma \vdash e_1 \Rightarrow e_1' : t_1; L_1 \qquad \Gamma[x \mapsto t_1] \vdash e_2 \Rightarrow e_2' : t_2; L_2}{\Gamma \vdash \mathtt{let}\, x = e_1\, \mathtt{in}\, e_2 \Rightarrow \mathtt{let}\, x = e_1'\, \mathtt{in}\, e_2' : t_2; L_1 \cup L_2} \ (\mathrm{Let}_a)$$

$$\frac{\Gamma \vdash e \Rightarrow e' : t; L \qquad S_I(\rho) = t}{\Gamma \vdash \mathtt{ref}\, e \Rightarrow \mathtt{ref}\, {}^{\rho} e' : ref\ (\rho); L \cup al(\rho)} \ (\mathrm{Ref}_a)$$

$$\frac{\Gamma \vdash e \Rightarrow e' : ref\ (\rho); L}{\Gamma \vdash {*}e \Rightarrow {*}e' : S_I(\rho); L \cup rd(\rho)} \ (\mathrm{Deref}_a)$$

$$\frac{\Gamma \vdash e_1 \Rightarrow e_1' : ref\ (\rho); L_1 \qquad \Gamma \vdash e_2 \Rightarrow e_2' : S_I(\rho); L_2}{\Gamma \vdash e_1 \mathtt{:=} e_2 \Rightarrow e_1' \mathtt{:=} e_2' : S_I(\rho); L_1 \cup L_2 \cup wr(\rho)} \ (\mathrm{Assign}_a)$$

$$\frac{\Gamma \vdash e \Rightarrow e' : t; L}{\Gamma \vdash \mathtt{annot}(e, Q) \Rightarrow \mathtt{annot}(e', Q) : t; L} \ (\mathrm{Annot}_a)$$

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

$$\frac{\begin{array}{c}\Gamma \vdash e_1 \Rightarrow e_1'; ref\ (\rho); L_1 \qquad\qquad S_I(\rho') = S_I(\rho) \\ \Gamma[x \mapsto ref\ (\rho')] \vdash e_2 \Rightarrow e_2' : t_2; L_2 \\ \rho \notin L_2 \qquad\qquad \rho' \notin loceff(\Gamma) \cup loceff(S_I(\rho)) \cup loceff(t_2)\end{array}}{\Gamma \vdash \mathtt{restrict}\, x = e_1\, \mathtt{in}\, e_2 \Rightarrow \mathtt{restrict}\, {}^{\rho'} x = e_1'\, \mathtt{in}\, e_2' : t_2; L_1 \cup L_2 \cup \rho} \ (\mathrm{Restrict}_a)$$

$$\frac{\Gamma \vdash e \Rightarrow e' : t; L \qquad \rho \notin loceff(\Gamma) \cup loceff(t)}{\Gamma \vdash e \Rightarrow e' : t; L - \rho} \ (\mathrm{Down}_a)$$

Figure 4.4: Alias, Effect, and Restrict Checking

- (Ref_a) translates an allocation by annotating it with the location ρ being allocated. We require that ρ's type in S_I matches t. Again, t may itself contain locations, so this condition also enforces our aliasing requirement. Finally, the effect of evaluating a `ref` includes the effect $al(\rho)$ of allocating it.

- (Deref_a) translates a dereference. To compute the type of $*e$, we look up the type of its location ρ in S_I. The effect of evaluating the dereference includes the effect $rd(\rho)$ of reading location ρ.

- (Assign_a) translates an assignment. The expression e_1 must be a pointer to some location ρ, and e_2's type must match ρ's type in S_I. Again, this matching condition enforces our aliasing requirement. The effect of the assignment is the union of the effects of evaluating the subexpressions and the effect $wr(\rho)$ of updating location ρ.

- (Annot_a) and (Check_a) translate type qualifier annotations and checks unchanged into the target language, since in this system all type qualifiers are flow-sensitive.

The most novel rule in this system, (Restrict_a), annotates `restrict` bindings with the location ρ' of x while simultaneously enforcing the semantics of `restrict`. This rule is similar to the rule for `let`, with four key differences:

- Recall that the semantics of `restrict` $x = e_1$ `in` e_2 state that during evaluation of e_2, the object x points to may only be accessed through x or copies of x. We enforce this requirement by binding x to an abstract location ρ' that may be different from the abstract location ρ of e_1. With this binding we can distinguish accesses through x and values derived from x, which have an effect on location ρ', from accesses through other aliases of e_1, which have an effect on ρ.

- The constraint $\rho \notin L_2$ forbids location ρ from being accessed during evaluation of e_2. (Recall that $\rho \notin L_2$ is shorthand for $rd(\rho) \notin L_2$, $wr(\rho) \notin L_2$, and $al(\rho) \notin L_2$.) Notice that dereferencing ρ' within e_2 is allowed, as long as ρ and ρ' are chosen to be different.

- Dually, the constraint $\rho' \notin \mathit{loceff}(\Gamma) \cup \mathit{loceff}(S_I(\rho)) \cup \mathit{loceff}(t_2)$ prevents the location ρ' of x from escaping the scope of e_2. Consider the following program:

```
let x = ref 0 in
let p = ... in
  restrict q = x in {p := q};
   /* 1 */
  restrict r = x in {**p}
```

Suppose x has type $\mathit{ref}\,(\rho_x)$. By (Restrict$_a$), the abstract locations pointed to by q and x can be different. Let q's type be $\mathit{ref}\,(\rho_q)$, where $\rho_x \neq \rho_q$. If the clause $\rho' \notin \mathit{loceff}\,(\Gamma) \cup \mathit{loceff}\,(S_I(\rho)) \cup \mathit{loceff}\,(t_2)$ were not in the hypothesis of (Restrict$_a$), the assignment p `:=` q would type check. But then, at program point 1, we would have two different names ρ_x and ρ_q for the same run-time location even though neither is restricted. Thus the dereference `**`p would type check even though the program is incorrect. We forbid ρ' escaping in (Restrict$_a$) to prevent this problem.

- The conclusion of (Restrict$_a$) contains the effect ρ, i.e., restricting a location is itself an effect. This naturally forbids the following program:

```
restrict y = x in
  restrict z = x in
    *y
```

If restricting a location had no effect on that location, it would be possible to restrict the same name twice and have both restricted names available for use in the same scope.

The final rule in our system, (Down$_a$), can be used to hide an effect [14, 51, 75], thereby increasing both the precision of `restrict` checking and the precision of the flow-sensitive analysis (Section 4.4). Suppose that our system proves a judgment

$$\Gamma \vdash e \Rightarrow e' : t; L$$

It may happen that the effect L contains locations that occur neither in Γ nor in t. While this behavior seems odd at first, observe that e may have subexpressions that allocate, read, and write from a temporary location ρ. But aside from (Down$_a$), the rules in Figure 4.4 only add to the effect of an expression. Thus as we move from the leaves to the root of the syntax tree, without (Down$_a$) effects can only grow. This behavior makes it difficult to

check `restrict` in the presence of recursive functions. For example, consider the following code:

```
fun foo x =
  let p = ref 0 in
    restrict y = p in {
      y := ...
      foo x
    }
```

Suppose that p has location ρ. Then without (Down$_a$), since y is initialized to p with `restrict` the effect of *foo* contains ρ. But then the recursive call to *foo* will not type check, since the hypothesis of (Restrict$_a$) requires that ρ is not in the effect of the application *foo* x.

Using (Down$_a$), however, we can remove effects on purely local state. The rule (Down$_a$) says that if location ρ is not visible outside the scope of e—i.e., if it is not bound in the environment Γ and is not in the type of e—then any effects on ρ can be removed from the effect of e [14, 75]. In the case of our example program, since ρ is purely local to the body of *foo*, it can be removed from the effect of *foo*, and thus with (Down$_a$) our example program type checks. By using (Down$_a$), we also increase the precision of the flow-sensitive qualifier system, since reducing the size of effect sets will increase the benefit of using effects to filter state at recursive function calls (recall Section 4.1.2).

Example 6. Figure 4.5 shows the translation of the example program in Figure 4.1 into our target language. The translation assigns x, y, and z distinct locations ρ_x, ρ_y, and ρ_z, respectively. Because f is called with argument z and our system is not polymorphic in locations, we require that the types of z and w match, and thus w is given the type *ref* (ρ_z). Finally, notice that since x and y are purely local to the body of f, using the rule (Down$_a$) we can hide all effects on ρ_x and ρ_y. The effect of f is $al(\rho_z) \cup wr(\rho_z)$ because f allocates z and writes to its parameter w, both of which have type *ref* (ρ_z). □

As given in Figure 4.4, (Down$_a$) is a non-syntactic rule. We can use the following lemma to construct a purely syntax-directed version of our system by incorporating (Down$_a$) into the other type rules.

```
fun al(ρz)∪wr(ρz) f w: ref (ρx) =
  let x = ref ρx 0
      y = ref ρy annot(1, a)
      z = ref ρz annot(2, b)
   in
      x := 3;
      w := 4;
      y := annot(5, c);
      if (···) {
        f z
      };
      check(*y, c)
```

Figure 4.5: Translation of Example Program in Figure 4.1

Lemma 4.1 *A proof of* $\Gamma \vdash e \Rightarrow e' : t; L$ *can be rewritten so that the only uses of (Down$_a$) are as the final step in the proof, or as the hypothesis of (Lam$_a$) or (Down$_a$).*

Proof: All rules except (Down$_a$) and (Lam$_a$) are monotonic in their effects, meaning that the set of effects in their conclusions is a superset of all the sets of effects in their hypotheses. All rules except (Restrict$_a$) place no conditions on the effects in their hypotheses. For any effect sets L and L' we have

$$(L - \rho) \cup L' = \begin{cases} (L \cup L') - \rho & \rho \notin L' \\ L \cup L' & \rho \in L' \end{cases}$$

Thus for any rules except (Down$_a$) and (Lam$_a$), we can move a use of (Down$_a$) above one of the hypotheses of an instance of a rule to below the conclusion of the rule. In (Restrict$_a$) we can clearly move uses of (Down$_a$) from above the e_1 hypothesis to below the conclusion. For the e_2 hypothesis, observe that if we could apply (Down$_a$) to remove ρ from the effect of e_2, then choosing the name ρ must have been arbitrary. Thus we can pick a fresh name ρ'' in place of ρ in the typing proof for e_2, and hence the constraint $\rho \notin L_2$ will be satisfied. (For details on the renaming step, see the proof of Theorem 4.4 below.) □

By applying Lemma 4.1 and combining sequences of (Down$_a$), we arrive at a purely syntax-directed type system by removing (Down$_a$) and replacing (Lam$_a$) by

$$\frac{\begin{array}{cc} \Gamma[x \mapsto t] \vdash e \Rightarrow e' : t'; L & \mathit{strip}(t) = s \\ \{\rho_1, \ldots, \rho_n\} \not\subseteq \mathit{loceff}(\Gamma) \cup \mathit{loceff}(t) \cup \mathit{loceff}(t') & L' = L - \{\rho_1, \ldots, \rho_n\} \end{array}}{\Gamma \vdash \lambda x{:}s.e \Rightarrow \lambda^L x{:}t.e' : t \longrightarrow^{L'} t'; \emptyset}$$

$$\frac{S \vdash e_1 \to l; S' \qquad S' \vdash e_2 \to v; S'' \qquad l \in dom(S'') \qquad S''(l) \neq \mathbf{err}}{S \vdash e_1 := e_2 \to v; S''[l \mapsto v]} \text{ [Assign]}$$

$$\frac{\begin{array}{c} S \vdash e_1 \to l; S' \qquad S'[l \mapsto \mathbf{err}, l' \mapsto S'(l)] \vdash e_2[x \mapsto l'] \to v, S'' \\ l \in dom(S') \qquad\qquad l' \notin dom(S') \end{array}}{S \vdash \texttt{restrict}\, x = e_1 \,\texttt{in}\, e_2 \to v; S''[l \mapsto S''(l'), l' \mapsto \mathbf{err}]} \text{ [Restrict]}$$

Figure 4.6: New Big-Step Operational Semantics Rules for Restrict

Finally, we can rewrite $L' = L - \{\rho_1, \ldots, \rho_n\}$ to $L' = L \cap (\mathit{loceff}(\Gamma) \cup \mathit{loceff}(t) \cup \mathit{loceff}(t'))$, which eliminates as many locations as possible:

$$\frac{\begin{array}{c} \Gamma[x \mapsto t] \vdash e \Rightarrow e' : t'; L \qquad \mathit{strip}(t) = s \\ L' = L \cap (\mathit{loceff}(\Gamma) \cup \mathit{loceff}(t) \cup \mathit{loceff}(t')) \end{array}}{\Gamma \vdash \lambda x{:}s.e \Rightarrow \lambda^L x{:}t.e' : t \longrightarrow^{L'} t'; \emptyset}$$

This is the final, syntax-directed rule for function definitions.

4.3.2 Semantics and Soundness of Restrict

In this section we sketch a proof of the soundness of `restrict` with respect to a precise semantics. Along the way we indirectly prove that the alias and effect computation by the type system is also correct, meaning that they are conservative approximations to the actual run-time behavior of the program. The complete proof of soundness can be found in Appendix B.

In order to prove soundness we need a semantics to make precise the meaning of `restrict`. We extend the standard semantics of Figure 2.2 by adding a new semantic reduction rule for `restrict` and modifying the rule for assignment slightly, as shown in Figure 4.6. The new rule for assignment [Assign] checks whether $S''(l)$ is **err** before allowing an update to location l. We need this modification because [Restrict] makes it possible for a store to contain locations mapped to **err** (see below), and normally [Assign] does not check the contents of l before overwriting it. We do not need to modify (Deref$_a$) to make this check because our semantics are strict in **err**.

The key rule is [Restrict], which uses copying to enforce `restrict`'s semantics. To evaluate $\texttt{restrict}\, x = e_1 \,\texttt{in}\, e_2$, we first evaluate e_1 normally, which must yield a pointer l. Within the body of e_2, the only way to access what l points to should be via the particular value that resulted from evaluating e_1. We enforce this by allocating a fresh location l'

initialized with the contents of l, and then binding l to **err** to forbid access through l. Recall that because our semantics is strict in **err**, and because of our modification to [Assign], any program that tries to read or write l within e_2 will reduce to **err**. The soundness of our checking system (see below) implies that no program evaluates to **err**, which in turn implies that an implementation can safely optimize `restrict` by eliding the copy of l. Instead, in an implementation `restrict` simply binds x to l.

Notice that it is not an error to use the value l, but only to dereference it. When e_2 has been evaluated, we re-initialize l to point to the value x points to, and then forbid accesses through l'. Forbidding access through l' corresponds to the requirement in the type rule (Restrict$_a$) that ρ' not escape. An alternative formulation is to rename occurrences of l' to l after e_2 finishes.

We next sketch a proof of soundness; the complete proof can be found in Appendix B. For purposes of our proof we have no need for the translation of expressions. Thus we abbreviate the judgment $\Gamma \vdash e \Rightarrow e' : t; L$ by $\Gamma \vdash e : t; L$. Locations l are represented in the proof as free variables, and thus their types are stored in Γ and they type check using (Var$_a$). We implicitly treat evaluated and unevaluated integers identically and use (Int$_a$) to type check both. Functions are represented not as closures but as syntactic functions, as in standard small-step semantics subject-reduction proofs [31, 122]. Thus evaluated functions are type checked using (Lam$_a$).

To show soundness we first show a subject-reduction result. We begin by introducing a notion of compatibility to capture when it is safe to evaluate an expression.

Definition 4.2 (Compatibility) *We say* Γ *and* L *are* compatible *with store* S*, written* $(\Gamma, L) \sim S$*, if*

1. $dom(\Gamma) = dom(S)$ *and*

2. *for all* $l \in dom(S)$*, there exists* ρ *such that* $\Gamma(l) = ref\ (\rho)$ *and*

$$\begin{cases} \Gamma \vdash S(l) : S_I(\rho); \emptyset & \text{if } S(l) \neq \textbf{\textit{err}} \\ \rho \notin L & \text{if } S(l) = \textbf{\textit{err}} \end{cases}$$

Intuitively, $(\Gamma, L) \sim S$ means an expression e that type checks in environment Γ and has effect L can execute safely in store S. Notice that the definition of compatibility requires $dom(\Gamma) = dom(S)$, i.e., that expressions typed in environment Γ contain locations but not

other free variables. This property is maintained during evaluation because in [App] we implement function calls with substitution.

As evaluation progresses in our proof we extend Γ with new locations allocated by `ref` expressions. It is a property of our semantics and type system that these extensions are safe, in the following sense:

Definition 4.3 (Safe Extension) *We say that* (Γ', S') *is a* safe extension *of* (Γ, S)*, written* $(\Gamma, S) \Rightarrow (\Gamma', S')$*, if*

1. $dom(\Gamma) = dom(S)$ *and* $dom(\Gamma') = dom(S')$*,*
2. $\Gamma'|_{dom(\Gamma)} = \Gamma$*,*
3. *for all* $l \in dom(S') - dom(S)$*, if* $S'(l) = \boldsymbol{err}$ *and* $\Gamma'(l) = ref\ (\rho)$*, then* $\rho \notin loceff(\Gamma)$*, and*
4. *for all* $l \in dom(S)$*, if* $S'(l) = \boldsymbol{err}$ *then* $S(l) = \boldsymbol{err}$*.*

Here $\Gamma'|_{dom(\Gamma)}(x)$ is the restriction of Γ' to the domain of Γ. Intuitively, $(\Gamma, S) \Rightarrow (\Gamma', S')$ means the **err**-bound locations in S' are either also **err**-bound in S, or if they are fresh (do not appear in Γ).

With these definitions we can state our subject reduction theorem. We use r to stand for a semantic reduction result, either a value v or **err**.

Theorem 4.4 (Subject Reduction) *If* $\Gamma \vdash e : t; L$ *and* $S \vdash e \rightarrow r; S'$*, where* $(\Gamma, L \cup L') \sim S$ *for some* L'*, then there exists* Γ' *such that*

1. $\Gamma' \vdash r : t; \emptyset$ *(which implies* $r \neq \boldsymbol{err}$*),*
2. $(\Gamma', L') \sim S'$*, and*
3. $(\Gamma, S) \Rightarrow (\Gamma', S')$

Proof (Sketch): By induction on the structure of the proof $S \vdash e \rightarrow r; S'$. The interesting case is $\texttt{restrict}\, x = e_1\, \texttt{in}\, e_2$. By assumption, we know

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : ref\ (\rho); L_1 \qquad\qquad S_I(\rho') = S_I(\rho) \\ \Gamma[x \mapsto ref\ (\rho')] \vdash e_2 : t_2; L_2 \\ \rho \notin L_2 \qquad \rho' \notin loceff(\Gamma) \cup loceff(S_I(\rho)) \cup loceff(t_2)\end{array}}{\Gamma \vdash \texttt{restrict}\, x = e_1\, \texttt{in}\, e_2 : t_2; L_1 \cup L_2 \cup \rho}$$

We also have a reduction $S \vdash \texttt{restrict}\, x = e_1 \,\texttt{in}\, e_2 \to r; S'$. By inspection of the semantic rules, to achieve this reduction we must have applied a reduction $S \vdash e_1 \to r_{e_1}; S'_{e_1}$ for e_1. Then by induction, there exists a Γ'_{e_1} satisfying

1. $\Gamma'_{e_1} \vdash r_{e_1} : \mathit{ref}\ (\rho); \emptyset$
2. $(\Gamma'_{e_1}, L_2 \cup \rho \cup L') \sim S'_{e_1}$
3. $(\Gamma, S) \Rightarrow (\Gamma'_{e_1}, S'_{e_1})$

After this step, though, we cannot apply induction directly to the evaluation

$$S'_{e_1}[r_{e_1} \mapsto \mathbf{err}, l' \mapsto S'_{e_1}(r_{e_1})] \vdash e_2[x \mapsto l'] \to r_{e_2}; S'_{e_2}$$

of e_2. The problem is that we would need to show the following compatibility:

$$(\Gamma'_{e_1}, L_2 \cup \rho \cup L') \sim S'_{e_1}[r_{e_1} \mapsto \mathbf{err}, l' \mapsto S'_{e_1}(r_{e_1})]$$

But of course this compatibility does not hold, because r_{e_1} maps to **err** in that store. We can solve this problem by simply removing ρ from the effect set for compatibility. But there is a deeper problem: although l' is fresh, ρ' may not be, and thus there may be some location l'' such that $\Gamma'_{e_1}(l'') = \mathit{ref}\ (\rho')$ and $S'_{e_1}(l'') = \mathbf{err}$. To solve this problem, we observe that the name ρ' is arbitrary. We construct a substitution $R = [\rho' \mapsto \rho'']$ for some fresh ρ'', with $S_I(\rho'') = S_I(\rho')$. Then from

$$\Gamma[x \mapsto \mathit{ref}\ (\rho')] \vdash e_2 : t_2; L_2$$

we conclude

$$R(\Gamma[x \mapsto \mathit{ref}\ (\rho')]) \vdash e_2 : Rt_2; RL_2$$

Using the hypothesis $\rho' \notin \mathit{loceff}(\Gamma) \cup \mathit{loceff}(S_I(\rho)) \cup \mathit{loceff}(t_2)$ in the type rule (Restrict$_a$), we derive

$$\Gamma[x \mapsto \mathit{ref}\ (\rho'')] \vdash e_2 : t_2; RL_2$$

Now we can show the following compatibility:

$$(\Gamma'_{e_1}, RL_2 \cup (L - \rho)) \sim S'_{e_1}[r_{e_1} \mapsto \mathbf{err}, l' \mapsto S'_{e_1}(r_{e_1})]$$

and thus apply induction to the evaluation of e_2. □

Given the subject-reduction theorem, soundness is easy to show:

Theorem 4.5 *If* $\emptyset \vdash e : t; L$ *and* $\emptyset \vdash e \to r; S$*, then* r *is not* ***err****.*

Proof: First observe that $(\emptyset, L) \sim \emptyset$. Then by Theorem B.10, there is a Γ' such that $\Gamma' \vdash r : t; \emptyset$. Thus r is not **err**. □

4.3.3 A Flow-Insensitive Inference System

In this section we give an efficient inference algorithm that produces a correct translation of an unannotated program into the target language of Figure 4.3. Our algorithm will be both sound and complete; if there is any proof that a program is correct according to the rules of Figure 4.4, the algorithm will find it.

As with the previous type inference systems we have discussed, our inference system for aliasing, effects, and `restrict` generates a system of constraints C. In this case, we generate equality constraints between types, inclusion constraints between effects, and disinclusion constraints between locations and effects:

$$\begin{array}{rcl} C & ::= & t_1 = t_2 \mid L \subseteq \varepsilon \mid \rho \notin L \\ t & ::= & \mathit{int} \mid \mathit{ref}\ (\rho) \mid t \longrightarrow^{\varepsilon} t' \\ L & ::= & \emptyset \mid \varepsilon \mid \rho \mid \mathit{rd}(\rho) \mid \mathit{wr}(\rho) \mid \mathit{al}(\rho) \mid L_1 \cup L_2 \mid L_1 \cap L_2 \end{array}$$

Here ε is an *effect variable* standing for an unknown set of effects. Notice that function types always have effect variables on their arrows, and inclusion constraints between effects have the special form $L \subseteq \varepsilon$, both of which make type equality and effect inclusion constraints particularly convenient to solve.

An important algorithmic consider is how we compute the sets of locations $\mathit{loceff}(t)$ and $\mathit{loceff}(\Gamma)$ for a type t and a type environment Γ, both of which are required by (Restrict_a) and the version of (Lam_a) with (Down_a) incorporated. In a program of size n, types may be of size $O(n)$ and type environments may have $O(n)$ variables in their domain. Thus we would like to avoid traversing types and environments as much as possible, since that alone would lead to a quadratic algorithm.

Our solution is to memoize the computation of $\mathit{loceff}(\cdot)$. Notice that we can view $\mathit{loceff}(t)$ and $\mathit{loceff}(\Gamma)$ simply as an effect, using our shorthand $\rho = \mathit{rd}(\rho) \cup \mathit{wr}(\rho) \cup \mathit{al}(\rho)$. To each type t used during inference we associate an effect variable ε_t to model $\mathit{loceff}(t)$. As part of our algorithm, we use the embed' function to map standard types to types with fresh locations and effects. As a side effect of applying embed', we generate constraints between

effect variables.

$$
\begin{array}{rcl}
\mathit{embed}'(\mathit{int}) & = & \mathit{int} \\
\mathit{embed}'(\mathit{ref}\ (s)) & = & \mathit{ref}\ (\rho) \qquad \rho \text{ fresh} \\
 & & \text{where } S_I(\rho) = \mathit{embed}'(s) \text{ and } \varepsilon_{S_I(\rho)} \cup \rho \subseteq \varepsilon_{\mathit{ref}\ (\rho)} \\
\mathit{embed}'(s \longrightarrow s') & = & t \longrightarrow^{\varepsilon} t' \quad \varepsilon \text{ fresh} \\
 & & \text{where } t = \mathit{embed}'(s), t' = \mathit{embed}'(s'), \text{ and } \varepsilon_t \cup \varepsilon \cup \varepsilon_{t'} \subseteq \varepsilon_{(t \longrightarrow^{\varepsilon} t')}
\end{array}
$$

We generate similar constraints when constructing types during type inference.

For type environments, observe that the type environment is empty at the root of the proof tree and then is only incrementally modified when type checking each subexpression. We use effect variables ε_Γ to contain the set of locations in environment Γ. When we extend Γ to $\Gamma' = \Gamma[x \mapsto t]$, we generate a fresh effect variable $\varepsilon_{\Gamma'}$ and the constraint $\varepsilon_\Gamma \cup \mathit{loceff}\,(t) \subseteq \varepsilon_{\Gamma'}$ In this way we succinctly model the set $\mathit{loceff}\,(\Gamma')$ without recomputing $\mathit{loceff}\,(\Gamma)$.

Figure 4.7 presents our type inference rules. Because the variables ε_Γ need to be communicated between adjacent steps of the proof, they are included to the left of the turnstile in the rules.

- (Var'_a) (Int'_a), (Annot'_a), and (Check'_a) are as before, except for the addition of ε_Γ to the left of the turnstile.

- (Lam'_a) incorporates (Down_a) as discussed at the end of Section 4.3.1, along with the computation of $\varepsilon_{\Gamma'}$ as described above. We use embed' to map the given standard parameter type to a type with fresh locations and effect variables. Notice that we always place an effect variable on the arrow of a function type. In this way when we solve equality constraints $t_1 = t_2$ between types, we produce equalities only between effect variables rather than between arbitrary effects. Since we are constructing a new type, we make the appropriate constraints so that $\varepsilon_{t \longrightarrow^{\varepsilon} t'}$ models $\mathit{loceff}\,(t \longrightarrow^{\varepsilon} t')$.

- (App'_a), (Ref'_a), (Deref'_a), and (Assign'_a) are written with explicit fresh locations and equality constraints between types where needed. Note that we assume that the program is correct with respect to the standard types, and so we avoid some shape matching constraints (for example, in (Assign'_a) we know that e_1 is a pointer type). In (Ref'_a) since we are constructing a new type we generate the appropriate constraints for its memoized location variable.

$$\frac{x \in dom(\Gamma)}{\Gamma, \varepsilon_\Gamma \vdash' x \Rightarrow x : \Gamma(x); \emptyset} \ (\mathrm{Var}'_a)$$

$$\frac{}{\Gamma, \varepsilon_\Gamma \vdash' n \Rightarrow n : int; \emptyset} \ (\mathrm{Int}'_a)$$

$$\frac{\begin{array}{c}\Gamma' = \Gamma[x \mapsto t] \quad t = embed'(s) \quad \varepsilon_\Gamma \cup \varepsilon_t \subseteq \varepsilon_{\Gamma'} \\ \Gamma', \varepsilon_{\Gamma'} \vdash' e \Rightarrow e' : t'; L \quad \varepsilon_t \cup \varepsilon \cup \varepsilon_{t'} \subseteq \varepsilon_{t \longrightarrow^\varepsilon t'} \\ L \cap (\varepsilon_{\Gamma'} \cup \varepsilon_{t'}) \subseteq \varepsilon \qquad \varepsilon_{\Gamma'},\ \varepsilon \text{ fresh}\end{array}}{\Gamma \vdash' \lambda x{:}s.e \Rightarrow \lambda^\varepsilon x{:}t.e' : t \longrightarrow^\varepsilon t'; \emptyset} \ (\mathrm{Lam}'_a)$$

$$\frac{\Gamma \vdash' e_1 \Rightarrow e'_1 : t \longrightarrow^L t'; L_1 \quad \Gamma \vdash' e_2 \Rightarrow e'_2 : t_2; L_2 \quad t = t_2}{\Gamma \vdash' e_1\ e_2 \Rightarrow e'_1\ e'_2 : t'; L_1 \cup L_2 \cup L} \ (\mathrm{App}'_a)$$

$$\frac{\begin{array}{c}\Gamma, \varepsilon_\Gamma \vdash' e_1 \Rightarrow e'_1 : t_1; L_1 \qquad \Gamma', \varepsilon_{\Gamma'} \vdash' e_2 \Rightarrow e'_2 : t_2; L_2 \\ \Gamma' = \Gamma[x \mapsto t_1] \qquad \varepsilon_\Gamma \cup \varepsilon_{t_1} \subseteq \varepsilon_{\Gamma'} \qquad \varepsilon_{\Gamma'} \text{ fresh}\end{array}}{\Gamma, \varepsilon_\Gamma \vdash' \mathtt{let}\, x = e_1 \,\mathtt{in}\, e_2 \Rightarrow \mathtt{let}\, x = e'_1 \,\mathtt{in}\, e'_2 : t_2; L_1 \cup L_2} \ (\mathrm{Let}'_a)$$

$$\frac{\Gamma, \varepsilon_\Gamma \vdash' e \Rightarrow e' : t; L \quad S_I(\rho) = t \quad \varepsilon_t \cup \rho \subseteq \varepsilon_{ref\ (\rho)} \quad \rho \text{ fresh}}{\Gamma, \varepsilon_\Gamma \vdash' \mathtt{ref}\, e \Rightarrow \mathtt{ref}^{\,\rho} e' : ref\ (\rho); L \cup al(\rho)} \ (\mathrm{Ref}'_a)$$

$$\frac{\Gamma, \varepsilon_\Gamma \vdash' e \Rightarrow e' : ref\ (\rho); L}{\Gamma, \varepsilon_\Gamma \vdash' *e \Rightarrow *e' : S_I(\rho); L \cup rd(\rho)} \ (\mathrm{Deref}'_a)$$

$$\frac{\Gamma, \varepsilon_\Gamma \vdash' e_1 \Rightarrow e'_1 : ref\ (\rho); L_1 \quad \Gamma, \varepsilon_\Gamma \vdash' e_2 \Rightarrow e'_2 : t_2; L_2 \quad S_I(\rho) = t_2}{\Gamma, \varepsilon_\Gamma \vdash' e_1 \mathtt{:=} e_2 \Rightarrow e'_1 \mathtt{:=} e'_2 : S_I(\rho); L_1 \cup L_2 \cup wr(\rho)} \ (\mathrm{Assign}'_a)$$

$$\frac{\Gamma, \varepsilon_\Gamma \vdash' e \Rightarrow e' : t; L}{\Gamma, \varepsilon_\Gamma \vdash' \mathtt{annot}(e, Q) \Rightarrow \mathtt{annot}(e', Q) : t; L} \ (\mathrm{Annot}'_a)$$

$$\frac{\Gamma, \varepsilon_\Gamma \vdash' e \Rightarrow e' : t; L}{\Gamma, \varepsilon_\Gamma \vdash' \mathtt{check}(e, Q) \Rightarrow \mathtt{check}(e', Q) : t; L} \ (\mathrm{Check}'_a)$$

$$\frac{\begin{array}{ccc}\Gamma, \varepsilon_\Gamma \vdash' e_1 \Rightarrow e'_1; ref\ (\rho); L_1 & & S_I(\rho') = S_I(\rho) \\ \Gamma' = \Gamma[x \mapsto ref\ (\rho')] & \varepsilon_{S_I(\rho')} \cup \rho' \subseteq \varepsilon_{ref\ (\rho')} & \varepsilon_\Gamma \cup \varepsilon_{ref\ (\rho')} \subseteq \varepsilon_{\Gamma'} \\ & \Gamma', \varepsilon_{\Gamma'} \vdash' e_2 \Rightarrow e'_2 : t_2; L_2 & \\ \rho \notin L_2 & \rho' \notin \varepsilon_\Gamma \cup \varepsilon_{S_I(\rho)} \cup \varepsilon_{t_2} & \varepsilon_{\Gamma'},\ \rho' \text{ fresh}\end{array}}{\Gamma, \varepsilon_\Gamma \vdash' \mathtt{restrict}\, x = e_1 \,\mathtt{in}\, e_2 \Rightarrow \mathtt{restrict}^{\,\rho'} x = e'_1 \,\mathtt{in}\, e'_2 : t_2; L_1 \cup L_2 \cup \rho} \ (\mathrm{Restrict}'_a)$$

Figure 4.7: Alias and Effect Inference and Restrict Checking

$$
\begin{array}{rcl}
C \cup \{int = int\} & \Rightarrow & C \\
C \cup \{ref\ (\rho) = ref\ (\rho')\} & \Rightarrow & C \cup \{\rho = \rho'\} \cup \{S_I(\rho) = S_I(\rho')\} \\
C \cup \{t_1 \longrightarrow^{\varepsilon} t_2 = t_1' \longrightarrow^{\varepsilon'} t_2'\} & \Rightarrow & C \cup \{t_1 = t_2\} \cup \{t_1' = t_2'\} \cup \{\varepsilon = \varepsilon'\} \\
C \cup \{\rho = \rho'\} & \Rightarrow & C[\rho \mapsto \rho'] \\
C \cup \{\varepsilon = \varepsilon'\} & \Rightarrow & C[\varepsilon \mapsto \varepsilon']
\end{array}
$$

(a) Type Unification

$$
\begin{array}{rcll}
C \cup \{\rho \notin L\} & \Rightarrow & C \cup \{\rho \notin \varepsilon\} \cup \{L \subseteq \varepsilon\} & \varepsilon \text{ fresh} \\
C \cup \{\emptyset \subseteq \varepsilon\} & \Rightarrow & C & \\
C \cup \{L_1 \cup L_2 \subseteq \varepsilon\} & \Rightarrow & C \cup \{L_1 \subseteq \varepsilon\} \cup \{L_2 \subseteq \varepsilon\} & \\
C \cup \{\emptyset \cap L \subseteq \varepsilon\} & \Rightarrow & C & \\
C \cup \{L \cap \emptyset \subseteq \varepsilon\} & \Rightarrow & C & \\
C \cup \{(L_1 \cup L_2) \cap L \subseteq \varepsilon\} & \Rightarrow & C \cup \{\varepsilon' \cap L \subseteq \varepsilon\} \cup \{L_1 \cup L_2 \subseteq \varepsilon'\} & \varepsilon' \text{ fresh} \\
C \cup \{L \cap (L_1 \cup L_2) \subseteq \varepsilon\} & \Rightarrow & C \cup \{L \cap \varepsilon' \subseteq \varepsilon\} \cup \{L_1 \cup L_2 \subseteq \varepsilon'\} & \varepsilon' \text{ fresh}
\end{array}
$$

(b) Constraint Normalization

Figure 4.8: Alias and Effect Constraint Resolution

- (Let$'_a$) is as before, with the added computation of $\varepsilon_{\Gamma'}$.
- (Restrict$'_a$) generates a fresh location ρ' for x, and we set $S_I(\rho') = S_I(\rho)$. Notice that we include ρ', and not necessarily ρ, in $\varepsilon_{\Gamma'}$. Finally, we generate two $\notin$ constraints requiring that ρ is not used in e_2 and that ρ' does not escape.

As in previous systems, after inference we are left with a system of constraints C we need to solve. We split the resolution of C into two phases. First we apply unification to solve the type equality constraints $t_1 = t_2$. Figure 4.8a gives the standard unification algorithm as a series of left-to-right rewrite rules. Because we assume that the program we are analyzing type checks with respect to the standard types, we know that none of the structural matching cases in Figure 4.8a can fail.[1] At this point, notice that we have assigned locations to each expression in the program, thereby completing our flow-insensitive may alias analysis.

After this first step, all that remain are *effect constraints* of the form $L \subseteq \varepsilon$ and $\rho \notin L$ (recall that the latter is shorthand for $rd(\rho) \notin L$ and $wr(\rho) \notin L$ and $al(\rho) \notin L$).

[1]When we unify t and t', we need not unify ε_t and $\varepsilon_{t'}$, which represent the locations appearing in the structure of t and t'. Resolving the constraint $t = t'$ unifies the locations appear in t and t', and thus it also unifies the locations contained in ε_t and $\varepsilon_{t'}$.

$$\begin{array}{rcl} C & ::= & L \subseteq \varepsilon \mid \rho \notin \varepsilon \\ L & ::= & M \mid M \cap M \\ M & ::= & rd(\rho) \mid wr(\rho) \mid al(\rho) \mid \varepsilon \end{array}$$

(a) Normal Form as Constraints

Constraints	Edge(s)
$M \subseteq \varepsilon$	$M \rightarrow \varepsilon$
$M \cap M' \subseteq \varepsilon$	$M \rightarrow_1 \cap, M' \rightarrow_2 \cap, \cap \rightarrow \varepsilon \quad \cap$ fresh

(b) Normal Form Inclusions as a Digraph

Figure 4.9: Effect Constraint Normal Form

Definition 4.6 *A* solution *to a system of effect constraints C is a mapping σ from effect variables to sets of locations such that for each $L \subseteq \varepsilon$ in C we have $\sigma(L) \subseteq \sigma(\varepsilon)$ and for each $\rho \notin L$ in C we have $\rho \notin \sigma(L)$, where we extend σ from effect variables to arbitrary effects in the natural way.*

If σ is a solution to the constraints C, we write $\sigma \models C$. If there is a σ such that $\sigma \models C$, then C is *saitsfiable*. Notice that abstract locations ρ are not in the domain of σ—intuitively, after applying the rules of Figure 4.8a, we treat abstract locations as constants.

Definition 4.7 *If $\sigma \models C$ and $\sigma' \models C$, then we define $\sigma \leq \sigma'$ if for all effect variables ε we have $\sigma(\varepsilon) \subseteq \sigma'(\varepsilon)$. If $\sigma \models C$, then σ is a* least solution *if $\sigma \leq \sigma'$ for any other solution σ'.*

Lemma 4.8 *If an effect constraint system C has a solution, then C has a least solution.*

Proof: Let Σ be any non-empty set of solutions of C. Let $\sigma = \bigcap_{\sigma' \in \Sigma} \sigma'$, where $\sigma(\varepsilon) = \bigcap_{\sigma' \in \Sigma} \sigma'(\varepsilon)$. We claim that $\sigma \models C$. Suppose C contains a constraint $\rho \notin L$. Then by assumption we know that for all $\sigma' \in \Sigma$ we have $\rho \notin \sigma'(L)$, and therefore $\rho \notin \sigma(L)$. Suppose C contains a constraint $L \subseteq \varepsilon$. Then by assumption we know that for all $\sigma' \in \Sigma$ we have $\sigma'(L) \subseteq \sigma'(\varepsilon)$. But then $\sigma(L) \subseteq \sigma(\varepsilon)$.

Clearly for and $\sigma' \in \Sigma$ we have $\sigma \leq \sigma'$. Thus the set of all solutions of C has a least element, and thus C has a least solution. □

To test satisfiability of an effect constraint system, we first apply the rules in Figure 4.8b to translate the constraints into the normal form shown in Figure 4.9a. Notice

```
EFFECT-SOLVE(C, ρ) =
  for all nodes n ∈ C do σ(n) ← ∅
  for all intersection nodes ∩ ∈ C do σ1(∩) ← σ2(∩) ← ∅
  let σ(rd(ρ)) ← {rd(ρ)}, σ(wr(ρ)) ← {wr(ρ)}, σ(al(ρ)) ← {al(ρ)}
  let W = {rd(ρ), wr(ρ), al(ρ)}, the set of nodes to visit
  while W ≠ ∅ do
    remove a node n from W
    for each edge n → ε do
      if σ(n) ⊈ σ(ε)
        then σ(ε) ← σ(ε) ∪ σ(n)
          W ← W ∪ {ε}
    for each edge n →i ∩ do
      if σ(n) ⊈ σi(∩)
        then σi(∩) ← σi(∩) ∪ σ(n)
          if σ1(∩) ∩ σ2(∩) ⊈ σ(∩)
            then σ(∩) ← σ(∩) ∪ (σ1(∩) ∩ σ2(∩))
              W ← W ∪ {∩}
  return σ
```

Figure 4.10: Solving Effect Constraint System with respect to Location ρ

that the rules in Figure 4.8b preserve least solutions but not arbitrary solutions. Also notice that in Figure 4.8b we do not consider the cases $(L_1 \cap L_2) \cap L_3 \subseteq \varepsilon$ and $L_1 \cap (L_2 \cap L_3) \subseteq \varepsilon$, since inspection of the rules of Figure 4.7 shows that constraints with nested intersections are never generated.

We view the inclusion constraints in a normal form effect constraint system as a directed graph, as shown in Figure 4.9b. The nodes n of the digraph are basic effects $rd(\rho)$, $wr(\rho)$, and $al(\rho)$ (with in-degree 0), effect variables ε (with arbitrary in-degree), and intersection nodes $\cap$ (with in-degree 2). We label the two edges into each $\cap$ node with either 1 or 2, marking whether they represent the left of the intersection or the right of the intersection. We generate a fresh $\cap$ node for each constraint $M \cap M' \subseteq \varepsilon$. Given a normal form effect constraint system, we test satisfiability by checking, for each constraint $\rho \notin \varepsilon$, whether $rd(\rho) \in \sigma(\varepsilon)$, $wr(\rho) \in \sigma(\varepsilon)$, or $al(\rho) \in \sigma(\varepsilon)$ in the least solution σ.

Figure 4.10 gives our algorithm for computing the least solution σ of the inclusion constraints for a particular location ρ. This algorithm is a simple extension of a standard graph traversal. For each intersection node $\cap$ in the graph, the algorithm maintains two sets $\sigma_1(\cap)$ and $\sigma_2(\cap)$ representing the current solution for the left and right parts of the

intersection, respectively. Intuitively, the algorithm in Figure 4.10 pushes the three basic effects $rd(\rho)$, $wr(\rho)$, and $al(\rho)$ transitively forward through the digraph.

Given a normal form effect constraint system, we test satisfiability by computing EFFECT-SOLVE(C, ρ) for each constraint $\rho \notin \varepsilon$. The constraint $\rho \notin \varepsilon$ is satisfiable if and only if none of the effects $rd(\rho)$, $wr(\rho)$, and $al(\rho)$ appear in $\sigma(\varepsilon)$.

Let n be the size of a program in the source language of Figure 4.3a, and let k be the number of occurrence of `restrict` in the program. Applying the type inference rules in Figure 4.7 takes $O(n)$ time and generates a system of constraints C of size $O(n)$. Applying the type unification rules in Figure 4.8a takes time $O(n\alpha(n))$, where $\alpha(n)$ is the inverse Ackerman's Function. Applying the rules in Figure 4.8b to normalize C takes $O(n)$ time and yields a normal form constraint system C' of size $O(n)$. Given that C' is size $O(n)$, the algorithm of Figure 4.10 takes $O(n)$ time each time it is invoked, and it is run twice for each `restrict` in the program, for a total of $O(kn)$ time. Summing up, the running time of the algorithm as a whole is $O(n\alpha(n) + kn)$.[2] In Section 4.5 we also use this algorithm to check whether $\rho \in L$. As indicated above, we can solve all such queries for location ρ in $O(n)$ time.

4.3.4 Subsumption on Effects

$$\frac{}{int \leq int}$$

$$\frac{}{ref\ (\rho) \leq ref\ (\rho)}$$

$$\frac{t_1' \leq t_1 \quad t_2 \leq t_2' \quad L \subseteq L'}{t_1 \longrightarrow^{L} t_2 \leq t_1' \longrightarrow^{L'} t_2'}$$

$$\frac{\Gamma \vdash e \Rightarrow e' : t; L \quad t \leq t'}{\Gamma \vdash e \Rightarrow e' : t'; L}\ (\mathrm{Sub}_a)$$

Figure 4.11: Subsumption Rule for Effects

We can extend the type and effect system in Figure 4.4 to admit a form of sub-

[2] Our source language is annotated with standard types, but even if we were to remove the standard types the running time of our algorithm is still the same—we can fold the rules of Figure 4.7 into standard type inference, and that whole phase runs in $O(n\alpha(n))$ time.

$$
\begin{array}{lcl}
\tau & ::= & Q\ \sigma \\
\sigma & ::= & int \mid ref\ (\rho) \mid (S, \tau) \longrightarrow^{L} (S', \tau') \\
S & ::= & \{\rho_1{}^{\eta_1} : \tau_1, \ldots, \rho_n{}^{\eta_n} : \tau_n\} \\
\eta & ::= & 0 \mid 1 \mid \omega
\end{array}
$$

Figure 4.12: Flow-Sensitive Qualified Types

typing among function types [51, 110], as shown in Figure 4.11. These rules can easily be incorporated into our inference algorithm. This form of subtyping increases the usefulness of the type system by allowing more programs with `restrict` annotations to type check with subtyping. To understand why, suppose we pass two functions f and g as parameters to h, and suppose that f has an effect on location ρ. Then without (Sub_a), the effects of f and g must be equal, meaning that g also appears to have an effect on ρ. But then we cannot call g in any context in which location ρ has been copied to a restricted pointer, even if it does not, in fact, access location ρ. With (Sub_a) we can avoid equating the effects of f and g, and thus we can avoid this problem.

4.4 Flow-Sensitive Type Qualifier Checking

In the second stage of our system, we perform flow-sensitive analysis, either inference or checking, on the qualifier-related annotations. In this section we present our checking system, and in the next section we present inference.

We take as input the target language of Figure 4.3, which has been decorated with types, locations, and effects. Throughout this stage we treat abstract locations ρ and effects L from the first step as constants.

We check the input program using the qualified types shown in Figure 4.12. As in Chapter 3, qualified types τ are standard types with qualifiers inserted at every level. The flow-sensitive system associates a store S with each program point, in contrast to the flow-insensitive alias and effect system, which uses a single global store S_I to assign types to locations. In Figure 4.12, function types are extended to $(S, \tau) \longrightarrow^{L} (S', \tau')$, where S describes the store the function is invoked in and S' describes the store when the function returns. As in Section 4.3, L is the effect of the function.

As discussed in Section 4.1.1, each location in each store has an associated *linearity*

η. In addition to the two linearities described before, 1 for linear locations (these admit strong updates) and ω for non-linear locations (which admit only weak updates), we also use a third linearity 0. In our system, the linearity 0 marks locations that are either unallocated or are in unreachable code. The three linearities form a lattice $0 < 1 < \omega$, and we define addition on linearities as expected: for any x we define $0 + x = x$, $1 + 1 = \omega$, and $\omega + x = \omega$.

An abstract store S is a vector assigning linearities and qualified types to the abstract locations appearing in the translated input program. To distinguish this form of store from the stores used in the next section, we refer to the stores in Figure 4.12 as *ground stores*. If S is an abstract store, we write $S(\rho)$ for ρ's type in S, and we write $S_{lin}(\rho)$ for ρ's linearity in S.

In our system, abstract stores assign a type and linearity to every abstract location in the target program, even if some of those locations are not directly accessible in the current scope. We make this choice so that handling hidden state is particularly simple. For example, consider the following program:

```
let f = (let x = ref 0 in λy.x := x + 1) in
  f(); f();
end
```

Here the function f increments x each time it is called and returns the new value of x. Thus the two calls to f yield 1 and then 2, respectively. Suppose that we want to track the value of x with a flow-sensitive analysis. Then although the name x is not visible outside the scope of f, in order to communicate the value of x from one call to f to the other we need to model the state of x at the top level, which our system does. As discussed in Section 4.1.2, we use effects to hide state that does not escape the scope of a function (see discussion of (Lam_f) below).

As in Chapter 3, we define a subtyping relation $\tau_1 \leq \tau_2$ between types. Whenever there is a control-flow branch from the state represented by abstract store S_1 to the state represented by abstract store S_2, we require that the types of the corresponding locations in S_1 and S_2 are compatible, which we write $S_1 \leq S_2$. Figure 4.13 gives the complete definition of $\tau_1 \leq \tau_2$ and $S_1 \leq S_2$, which are extensions of the rules of Figure 3.2. The rule ($\text{Int}_f \leq$) is as before. In ($\text{Ref}_f \leq$) we require that the locations on the left- and right-hand sides of the $\leq$ are the same. The translation step in Section 4.3 enforces this property, which corresponds to the standard requirement that subtyping becomes equality below a *ref* constructor (see

$$\frac{Q \leq Q'}{Q\ \ int \leq Q'\ \ int}\ (\text{Int}_f \leq)$$

$$\frac{Q \leq Q'}{Q\ \ ref\ (\rho) \leq Q'\ \ ref\ (\rho)}\ (\text{Ref}_f \leq)$$

$$\frac{Q \leq Q' \quad \tau_1' \leq \tau_1 \quad \tau_2 \leq \tau_2' \quad S_1' \leq S_1 \quad S_2 \leq S_2'}{Q\ (S_1, \tau_1) \longrightarrow^L (S_2, \tau_2) \leq Q'\ (S_1', \tau_1') \longrightarrow^L (S_2', \tau_2')}\ (\text{Fun}_f \leq)$$

$$\frac{\tau_i \leq \tau_i' \qquad \eta_i \leq \eta_i' \qquad i = 1..n}{\{\rho_1{}^{\eta_1} : \tau_1, \ldots, \rho_n{}^{\eta_n} : \tau_n\} \leq \{\rho_1'{}^{\eta_1'} : \tau_1', \ldots, \rho_n'{}^{\eta_n'} : \tau_n'\}}\ (\text{Store}_f \leq)$$

Figure 4.13: Subtyping and Store Compatibility Rules

Figure 3.2). We emphasize that in this phase we treat abstract locations ρ as constants, and we never attempt or need to unify two distinct locations to satisfy ($\text{Ref}_f \leq$). The rule ($\text{Fun}_f \leq$) states that functions are contravariant in their domain type and initial store, and covariant in their range type and final store. We require that the effects of the two function types match exactly; it is also sound to allow the effect of the left-hand function to be a subset of the effect of the right-hand function (Section 4.3.4). Finally, rule ($\text{Store}_f \leq$) says that two stores are compatible if their linearities and qualified types are compatible point-wise. Notice that here we use the fact that every location appears in every store.

Figure 4.14 presents our flow-sensitive type qualifier checking system. In this type system, judgments have the form $\Gamma, S \vdash e : \tau, S'$, meaning that in type environment Γ and in state S, evaluating e yields a result of type τ and a new state S'. In these rules, we construct partial abstract stores of the form $S|_L$. The partial store $S|_L$ maps all locations in L to their types and linearities as in S and is undefined elsewhere; we consider location ρ to occur in L if $rd(\rho) \in L$, $wr(\rho) \in L$, or $al(\rho) \in L$. We use $\neg L$ for the complement of the locations in L with respect to the set of all locations occurring in the input program. We use the operation $\oplus$ to combine two partial stores, defined in Figure 4.15. Notice that if we combine two partial stores using $\oplus$, the types of any common locations must match. Finally, as before the rules use a function $strip(\cdot)$ for removing qualifiers and stores from

$$\frac{x \in dom(\Gamma)}{\Gamma, S \vdash x : \Gamma(x), S} \ (\mathrm{Var}_f)$$

$$\frac{}{\Gamma, S \vdash n : int, S} \ (\mathrm{Int}_f)$$

$$\frac{\Gamma[x \mapsto \tau], S_\lambda \vdash e : \tau', S'_\lambda \qquad strip(\tau) = t}{\Gamma, S \vdash \lambda^L x{:}t.e : (S_\lambda, \tau) \longrightarrow^L (S'_\lambda, \tau'), S} \ (\mathrm{Lam}_f)$$

$$\frac{\begin{array}{c}\Gamma, S \vdash e_1 : Q\ (S_\lambda, \tau) \longrightarrow^L (S'_\lambda, \tau'), S' \qquad \Gamma, S' \vdash e_2 : \tau, S'' \\ S''|_L \oplus S''' \leq S_\lambda\end{array}}{\Gamma, S \vdash' e_1\ e_2 : \tau', S'_\lambda|_L \oplus S''|_{\neg L}} \ (\mathrm{App}_f)$$

$$\frac{\Gamma, S \vdash e_1 : \tau_1, S' \qquad \Gamma[x \mapsto \tau_1], S' \vdash e_2 : \tau_2, S''}{\Gamma, S \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau_2, S''} \ (\mathrm{Let}_f)$$

$$\frac{\Gamma, S \vdash e : S'(\rho), S'}{\Gamma, S \vdash \mathtt{ref}^{\,\rho} e : ref\ (\rho), S' \oplus \{\rho^1 : S'(\rho)\}} \ (\mathrm{Ref}_f)$$

$$\frac{\Gamma, S \vdash e : Q\ \ ref\ (\rho), S'}{\Gamma, S \vdash *e : S'(\rho), S'} \ (\mathrm{Deref}_f)$$

$$\frac{\begin{array}{c}\Gamma, S \vdash e_1 : Q\ \ ref\ (\rho), S' \qquad \Gamma, S' \vdash e_2 : \tau, S'' \\ \eta = S''_{lin}(\rho) \qquad\qquad \omega \leq \eta \Longrightarrow S''(\rho) \leq \tau\end{array}}{\Gamma, S \vdash e_1 \mathtt{:=} e_2 : \tau, S''|_{\neg\rho} \oplus \{\rho^\eta : \tau\}} \ (\mathrm{Assign}_f)$$

$$\frac{\Gamma, S \vdash e : \sigma, S'}{\Gamma, S \vdash \mathtt{annot}(e, Q) : Q\ \sigma, S'} \ (\mathrm{Annot}_f)$$

$$\frac{\Gamma, S \vdash e : Q'\ \sigma, S' \qquad Q' \leq Q}{\Gamma, S \vdash \mathtt{check}(e, Q) : Q'\ \sigma, S'} \ (\mathrm{Check}_f)$$

$$\frac{\begin{array}{c}\Gamma, S \vdash e_1 : Q\ \ ref\ (\rho), S' \\ \Gamma[x \mapsto Q\ \ ref\ (\rho')], S' \oplus \{\rho'^{\eta'} : S'(\rho)\} \vdash e_2 : \tau_2, S'' \\ \eta = S''_{lin}(\rho) \qquad\qquad \omega \leq \eta \Longrightarrow S''(\rho) \leq S''(\rho')\end{array}}{\Gamma, S \vdash \mathtt{restrict}^{\,\rho'} x = e_1\ \mathtt{in}\ e_2 : \tau_2, S''|_{\neg\rho} \oplus \{\rho^\eta : S''(\rho')\}} \ (\mathrm{Restrict}_f)$$

$$\frac{\Gamma, S \vdash e : \tau, S' \qquad \tau \leq \tau'}{\Gamma, S \vdash e : \tau', S'} \ (\mathrm{Sub}_f)$$

Figure 4.14: Flow-Sensitive Qualified Type Checking System

$$S \oplus S'(\rho) = \begin{cases} S(\rho) & S(\rho) = S'(\rho) \\ S(\rho) & \rho \in dom(S) \wedge \rho \notin dom(S') \\ S'(\rho) & \rho \notin dom(S) \wedge \rho \in dom(S') \end{cases}$$

$$S \oplus S'_{lin}(\rho) = \begin{cases} S_{lin}(\rho) + S'_{lin}(\rho) & \rho \in dom(S) \wedge \rho \in dom(S') \\ S_{lin}(\rho) & \rho \in dom(S) \wedge \rho \notin dom(S') \\ S'_{lin}(\rho) & \rho \in dom(S') \wedge \rho \notin dom(S) \end{cases}$$

Figure 4.15: $\oplus$ Operation on Partial Stores

flow-sensitive types:

$$\begin{aligned} strip(Q\ int) &= int \\ strip(Q\ ref\ (\rho)) &= ref\ (\rho) \\ strip(Q\ (S,\tau) \longrightarrow^L (S',\tau')) &= strip\,\tau \longrightarrow^L strip\,\tau' \end{aligned}$$

We discuss the rules in Figure 4.14:

- (Var_f) and (Int_f) are standard. As in Figure 3.5, when we assign a type to an integer we leave off the outermost qualifier, which forces the programmer to add a qualifier annotation (see (Annot_f) below).

- (Lam_f) type checks function body e in store S_λ with parameter x bound to a type with the same shape as t. (Note that this last check is not strictly necessary.) As with (Int_f), we leave the outermost qualifier off of the resulting function type, which forces the programmer to add an explicit qualifier annotation.

- (App_f) type checks e_1 followed by e_2—notice the left-to-right evaluation order enforced by using e_1's final abstract store as the initial store for checking e_2. We model the function call by requiring that the actual argument e_2 match the type of e_1's domain, and that the current state S'' is compatible with the initial state e_1 expects. As discussed in Section 4.1.2, we use the effect L of the function e_1 to avoid conflating hidden state. With the condition $S''|_L \oplus S''' \leq S_\lambda$, we require only that the locations from S'' that appear in L are compatible with the corresponding locations in S_λ. This constraint has no effect on the locations not in L, since S''' can be chosen arbitrarily.

  The state after the function call is $S'_\lambda|_L \oplus S''|_{\neg L}$, which combines S'_λ and S'' according to L. If e_1 accessed a location ρ, then after the function call we take ρ's linearity and

qualified type from S'_λ. Otherwise, we take ρ's linearity from S''. We can actually improve on this slightly by distinguishing the different kinds of effects in L; see the discussion of *Merge* in the next section.

- (Let_f) is standard. Notice the left-to-right order of evaluation.
- (Ref_f) type checks an allocation. The resulting store is the same as store S', except that location ρ has been allocated once more. The type of e is required to be compatible with ρ's type in S'. We make this choice to simplify inference slightly; see the discussion of (Ref'_f) in the next section.
- (Deref_f) type checks a memory read. The result of the dereference is ρ's qualified type in S'. As in Chapter 3, we allow any qualifier to appear on e's type; qualifiers are checked only be (Check_f), below.
- (Assign_f) type checks a memory update. The store after the assignment matches the store S'' except that location ρ now has type τ. If ρ is non-linear in S'' we require a weak update with the constraint $S''(\rho) \leq \tau$.
- (Annot_f) adds an outermost qualifier to a type, and (Check_f) tests the outermost qualifier of a type, just as in Chapter 3.
- (Restrict_f) type checks a `restrict` construct. First we evaluate e_1, which must be a pointer. Then we check e_2 in an environment where x has been bound to location ρ', as specified in the translated program. We check e_2 in a store like S' except that location ρ' has also been allocated, and its initial value must be compatible with ρ's value. The initial linearity of ρ' is η', which may be linear or non-linear independently of the linearity of ρ. When the scope of the `restrict` ends, we update ρ with the final value of ρ'; this is either a weak update or a strong update, depending on the linearity of ρ in S'' (which is the same as ρ's linearity in S', since e_2 cannot read, write, or allocate location ρ).
- (Sub_f) adds subsumption, as defined in Figure 4.13, to the system.

4.5 Flow-Sensitive Type Qualifier Inference

In this section we give an efficient algorithm for flow-sensitive type qualifier inference. The key to efficiency is to choose our representation of stores carefully. The ground stores in Figure 4.12 contain one occurrence of each location in the program. In a program of size n, the alias and effect inference of Section 4.3.3 may produce an annotated program with n locations. If we need to represent the type of n locations at n program points, that alone would lead to at least an n^2 algorithm.

Thus during inference, rather than explicitly associating a ground store with every program point, we represent stores using a constraint formalism. As the base case, we model an unknown store using a *store variable* ε. We define four *store constructors* that represent the differences between states, yielding the following grammar for stores:

$$S ::= \varepsilon \mid \mathit{Alloc}(S, \rho) \mid \mathit{Merge}(S, S', L) \mid \mathit{Filter}(S, L) \mid \mathit{Assign}(S, \rho{:}\tau)$$

Intuitively, the four store constructors model exactly the operations on stores we use in Figure 4.14. For example, in the rule (Ref_f) we build a store $S' \oplus \{\rho^1 : S'(\rho)\}$. We represent this store during inference with the constructed store $\mathit{Alloc}(S', \rho)$.

Our type inference rules generate *store constraints* of the form $S \leq \varepsilon$, as well as the subtyping constraints and qualifier constraints we have seen in Chapter 3.

Definition 4.9 *A* solution *to a system of store, type, and qualifier constraints is a mapping σ from store variables to ground stores and from qualifier variables to qualifier constants such that for each constraint $S \leq \varepsilon$ we have $\sigma(S) \leq \sigma(\varepsilon)$, for each constraint $\tau_1 \leq \tau_2$ we have $\sigma(\tau_1) \leq \sigma(\tau_2)$, and for each constraint $Q_1 \leq Q_2$ we have $\sigma(Q_1) \leq \sigma(Q_2)$, as defined in Figure 4.13.*

If C is a system of store, type, and qualifier constraints, as before we write $\sigma \models C$ if σ is a solution to C.

The meaning of each store constructor is given in Figure 4.16 by showing how a solution σ extends to constructed stores. Here the condition $\rho \in L$ means $rd(\rho) \in L$, $wr(\rho) \in L$, or $al(\rho) \in L$. We discuss the four store constructors:

- The store $\mathit{Alloc}(S, \rho)$ is the same as store S, except that location ρ has been allocated once more. Allocating location ρ does not affect the types in the store but increases the linearity of location ρ by one. In Figure 4.14, we wrote this store as $S \oplus \{\rho^1 : S(\rho)\}$.

$$\sigma(Alloc(S,\rho'))(\rho) = \sigma(S)(\rho)$$

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

$$\sigma(Filter(S,L))(\rho) = \sigma(S)(\rho) \qquad \rho \in L$$

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

(a) Types

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

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

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

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

(b) Linearities

$$\omega \leq \sigma(S)_{lin}(\rho) \implies \sigma(S)(\rho) \leq \tau \qquad \text{for all stores } Assign(S,\rho{:}\tau)$$

(c) Weak Updates

Figure 4.16: Extending a Solution to Constructed Stores

- The store $Merge(S, S', L)$ combines stores S and S' according to effect L. If L contains an allocation of location ρ, then $Merge(S, S', L)$ assigns ρ its type and linearity in S. If L contains a write to but not an allocation of location ρ, then $Merge(S, S', L)$ assigns ρ its type in S and its linearity in S'. Otherwise $Merge(S, S', L)$ assigns ρ its type and linearity in S'. We use *Merge* to model the state after a function call. In Figure 4.14 we used the less precise form $S|_L \oplus S'|_{\neg L}$ for a similar purpose. *Merge* is more precise because it distinguishes different kinds of effects in L. See the discussion of (App'_f) below.

- The store $Filter(S, L)$ assigns the same types and linearities as S to all locations in L. The types of any locations not in L are unconstrained, and their linearities are 0 since they have not been allocated. In Figure 4.14 we wrote $Filter(S, L)$ as $S|_L \oplus S'$, where S' is arbitrary. Note that because of the particular constraints our inference algorithm generates, our solution σ need not assign types or linearities for S', i.e., for the locations not in L.

- Finally, the store $Assign(S, \rho : \tau)$ is the same as store S, except that location ρ has been updated to type τ. If ρ is linear in S, then this is a strong update, so ρ's new type is τ. Otherwise, if ρ is non-linear in S, then in Figure 4.16c we require that the type of ρ in $Assign(S, \rho : \tau)$ be at least its type in S; this corresponds to a weak update.[3]

Figure 4.17 gives the rules for our flow-sensitive type qualifier inference system. As discussed above, these rules use our four store constructors to represent changes in state. We use our standard *embed'* function to add fresh qualifier variables and store variables to types:

$$\begin{array}{rcll} embed'(int) & = & \kappa\ int & \kappa \text{ fresh} \\ embed'(ref\ (\rho)) & = & \kappa\ ref\ (\rho) & \kappa \text{ fresh} \\ embed'(t \longrightarrow^L t') & = & \kappa\ (\varepsilon, embed'(t)) \longrightarrow^L (\varepsilon', embed'(t')) & \kappa,\ \varepsilon,\ \varepsilon' \text{ fresh} \end{array}$$

As in the previous inference systems, we assume that standard type inference has already been performed on the program, and so we simplify some of the matching conditions on the hypotheses of our type rules. We write $S(\rho)$ for the type associated with ρ in store S. If we wish to view our constraint generation algorithm as producing a linear-size system of

[3] In CQUAL we require equality here (Chapter 5).

$$\frac{x \in dom(\Gamma)}{\Gamma, S \vdash' x : \Gamma(x), S} \ (\text{Var}'_f)$$

$$\frac{\kappa \text{ fresh}}{\Gamma, S \vdash' n : \kappa\ \mathit{int}, S} \ (\text{Int}'_f)$$

$$\frac{\Gamma[x \mapsto \tau], \varepsilon \vdash' e : \tau', S' \qquad \tau = \mathit{embed}'(t) \qquad S' \leq \varepsilon' \qquad \varepsilon,\ \varepsilon',\ \kappa \text{ fresh}}{\Gamma, S \vdash' \lambda^L x{:}t.e : \kappa\ (\varepsilon, \tau) \longrightarrow^L (\varepsilon', \tau'), S} \ (\text{Lam}'_f)$$

$$\frac{\begin{array}{c}\Gamma, S \vdash' e_1 : Q\ (\varepsilon, \tau) \longrightarrow^L (\varepsilon', \tau'), S' \qquad \Gamma, S' \vdash' e_2 : \tau_2, S'' \\ \tau_2 \leq \tau \qquad\qquad \mathit{Filter}(S'', L) \leq \varepsilon\end{array}}{\Gamma, S \vdash' e_1\ e_2 : \tau', \mathit{Merge}(\varepsilon', S'', L)} \ (\text{App}'_f)$$

$$\frac{\Gamma, S \vdash' e_1 : \tau_1, S' \qquad \Gamma[x \mapsto \tau_1], S' \vdash' e_2 : \tau_2, S''}{\Gamma, S \vdash' \texttt{let}\, x = e_1\, \texttt{in}\, e_2 : \tau_2, S''} \ (\text{Let}'_f)$$

$$\frac{\Gamma, S \vdash' e : \tau, S' \qquad \tau \leq S'(\rho) \qquad \kappa \text{ fresh}}{\Gamma, S \vdash' \texttt{ref}^{\,\rho} e : \kappa\ \mathit{ref}\ (\rho), \mathit{Alloc}(S', \rho)} \ (\text{Ref}'_f)$$

$$\frac{\Gamma, S \vdash' e : Q\ \mathit{ref}\ (\rho), S'}{\Gamma, S \vdash' *e : S'(\rho), S'} \ (\text{Deref}'_f)$$

$$\frac{\begin{array}{c}\Gamma, S \vdash' e_1 : Q\ \mathit{ref}\ (\rho), S' \qquad \Gamma, S' \vdash' e_2 : \tau, S'' \\ \tau' = \mathit{embed}'(S_I(\rho)) \qquad\qquad \tau \leq \tau'\end{array}}{\Gamma, S \vdash' e_1 \texttt{:=} e_2 : \tau', \mathit{Assign}(S'', \rho{:}\tau')} \ (\text{Assign}'_f)$$

$$\frac{\Gamma, S \vdash' e : Q'\ \sigma, S' \qquad Q' \leq Q}{\Gamma, S \vdash' \texttt{check}(e, Q) : Q'\ \sigma, S'} \ (\text{Check}'_f)$$

$$\frac{\begin{array}{c}\Gamma, S \vdash' e_1 : Q\ \mathit{ref}\ (\rho), S' \qquad S'' = \mathit{Alloc}(S', \rho') \qquad S'(\rho) \leq S''(\rho') \\ \Gamma[x \mapsto Q\ \mathit{ref}\ (\rho')], S'' \vdash' e_2 : \tau_2, S'''\end{array}}{\Gamma, S \vdash' \texttt{restrict}^{\,\rho'} x = e_1\, \texttt{in}\, e_2 : \tau_2, \mathit{Assign}(S''', \rho{:}S'''(\rho'))} \ (\text{Restrict}'_f)$$

Figure 4.17: Flow-Sensitive Qualified Type Inference System

constraints, then we add $S(\rho)$ to our grammar for types, and instead of using the simplified matchings we should generate constraints to determine the shape of the hypothesis in the type inference rules, as in Figure 2.5. Similarly we need to add the $embed'(S_I(\rho))$ in (Assign'_f) (see below) to our type grammar. In practice we solve the constraints as they are generated, so this is not a concern; we discuss the computation of $S(\rho)$ in Section 4.5.1.

We discuss the rules in Figure 4.17:

- (Var'_f) and (Int'_f) are standard. As in Chapter 3, instead of using qualifier annotations we make a fresh qualifier variable for the outermost qualifier of an *int* type.

- (Lam'_f) makes fresh store variables to model the initial and final state of the function being defined. We add fresh qualifiers and store variables to t to yield the type of the parameter τ. Finally, we add a constraint that the state after e is evaluated is compatible with the final state ε' of the function. Notice that function types always have store variables rather than arbitrary stores in their domain and range. As in Section 4.3.3, this means that any generated subtyping constraints yield store constraints only among store variables.

- (App'_f) models a function call. We require that the actual argument be compatible with e_1's domain type, and we require that the state S'' before the function call be compatible with the state ε that the function expects. As in (App_f), here we use *Filter* so that only the types and linearities of locations appearing in L need be compatible. We similarly use *Merge* to merge the state after the function call.

  Recall that *Merge* is slightly refined over the construction used in (App_f) from the checking system. Using *Merge* in (App'_f), if the call to e_1 allocates a location ρ, then we must assume the worst, and after the function call we take ρ's linearity and qualified type from ε'. If e_1 writes a location but does not allocate it, then we must take ρ's qualified type from ε', but we can take ρ's linearity from S''. If neither of those two conditions hold (for example, if ρ was not accessed at all or was only read during the call), then we can take both ρ's linearity and qualified type from S''.

- (Let'_f), (Deref'_f), and (Check'_f) are straightforward.

- (Ref'_f) uses the *Alloc* constructor to build a new store in which location ρ has been allocated once more. We require that the type of τ be compatible with ρ's type in S'.

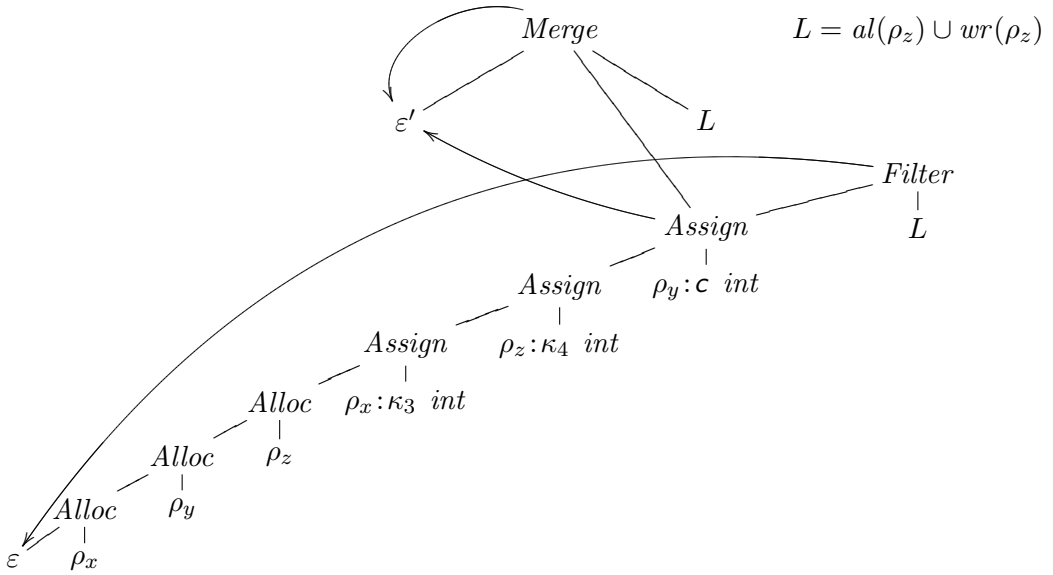

Figure 4.18: Store Constraints for Example in Figure 4.5

An alternative formulation would be to track the type τ as part of the constructed *Alloc* and only constrain τ to be compatible with $S'(\rho)$ if ρ is non-linear after the allocation. Although recording types in *Alloc* would be slightly more precise, we do not do so to make inference simpler: in the current system there is only one construct, *Assign*, that interacts with linearities.

- (Assign$'_f$) computes the type τ of e_2 and produces a new store representing the assignment of τ' to location ρ, where $\tau \leq \tau'$. Notice that we perform a subtyping step here. This corresponds to the subtyping in rule (Assign$'_q$) of Figure 3.6. Our definition of *Assign* in Figure 4.16 makes the assignment a strong or weak update, depending on ρ's inferred linearity in S''.

- (Restrict$'_f$) is exactly like (Restrict$_f$), except we use store constructors *Alloc* and *Assign* in the same way we do in (Ref$'_f$) and (Assign$'_f$).

Example 7. Figure 4.18 shows in graph form the stores and store constraints generated for the example program in Figure 4.5. This graph uses two kinds of edges. Store constructors are represented by undirected edges from the constructor to its arguments, and the store constraint $S \leq \varepsilon$ is represented with a directed edge from S to ε. We have slightly simplified the graph for clarity. Here ε is f's initial store and ε' is f's final store.

We step through constraint generation. We model the allocation of ρ_x with the store $Alloc(\varepsilon, \rho_x)$. Location ρ_x is initialized to 0, which is given the type $\kappa_0\ \ int$ for fresh qualifier variable κ_0. (Ref'_f) generates the constraint $\kappa_0\ \ int \leq \varepsilon(\rho_x)$ to require that the type of 0 be compatible with $\varepsilon(\rho_x)$. We model the allocation and initialization of ρ_y and ρ_z similarly. Then we construct three *Assign* stores to represent the assignment statements. We give 3 and 4 the types $\kappa_3\ \ int$ and $\kappa_4\ \ int$, respectively, where κ_3 and κ_4 are fresh qualifier variables.

For the recursive call to f, we construct a *Filter* store and add a constraint on ε. The *Merge* store represents the state when the recursive call to f returns. We join the two branches of the conditional by making edges to ε'. Notice the cycle, due to recursion, in which state from ε' can flow to the *Merge*, which in turn can flow to ε'. Finally, the qualifier check requires that $\varepsilon'(\rho_y)$ has qualifier c. □

4.5.1 Flow-Sensitive Constraint Resolution

As stated above, the rules of Figure 4.17 generate a constraint system C containing three kinds of constraints: qualifier constraints $Q \leq Q'$, subtyping constraints $\tau \leq \tau'$, and store constraints $S \leq \varepsilon$. In Chapter 3 we already discussed how to solve qualifier and subtyping constraints (the addition of stores to types changes the algorithm trivially, as shown in Figure 4.13). Thus in this section we focus on computing a solution σ to the store constraints. We assume that the alias and effect inference rules of Figure 4.7 have already been applied to the input program, and thus using the algorithm in Figure 4.10 we can ask queries of the form $\rho \in L$ in time $O(n)$, where n is the size of the input program.

Our analysis is most precise if as few locations as possible are non-linear. Recall that linearities naturally form a partial order $0 < 1 < \omega$. Thus, given a system of constraints C, we perform a least fixpoint computation to determine the linearity $\sigma(S)_{lin}(\rho)$ for each store S and location ρ in our solution σ. We initially assume that in every store, location

ρ has linearity 0. Then we exhaustively apply the rules in Figure 4.16b and the rule

$$\sigma(\varepsilon)_{lin}(\rho) = \max_{\{S|S \leq \varepsilon \in C\}} \sigma(S)_{lin}(\rho)$$

until we reach a fixpoint. This last rule is derived from Figure 4.13. In our implementation, we compute $\sigma(S)_{lin}(\rho)$ in a single pass over the store constraints using Tarjan's strongly-connected components algorithm [2] to find cycles in the store constraint graph. For each such cycle containing more than one allocation of the same location ρ, we set the linearity of ρ to ω in all stores on the cycle. Since linearities only affect assignments, we only compute $\sigma(S)_{lin}(\rho)$ if it is necessary to determine $\sigma(S')_{lin}(\rho)$ for some $Assign(S', \rho{:}\tau) \in C$.

Given this algorithm to compute $\sigma(S)_{lin}(\rho)$, in principle we can then solve the implied typing constraints using the following simple procedure. For each store variable ε, initialize the type component $\sigma(\varepsilon)$ of our solution σ to the map

$$\{\rho_1 : embed'(S_I(\rho_1)),\ \ldots,\ \rho_n : embed'(S_I(\rho_n))\}$$

thereby assigning fresh qualifiers to the type of every location at every program point. Replace uses of $S(\rho)$ in C with $\sigma(S)(\rho)$, using the logic in Figure 4.16. Then apply the following two closure rules until no more constraints are generated:

$$\begin{array}{rcll} C \cup \{S \leq \varepsilon\} & \Longrightarrow & C \cup \{S \leq \varepsilon\} \cup \{\sigma(S)(\rho) \leq \sigma(\varepsilon)(\rho)\} & \text{for all } \rho \\ C \cup \{\omega \leq \sigma(S)_{lin}(\rho)\} & \Longrightarrow & C \cup \{\omega \leq \sigma(S)_{lin}(\rho)\} \cup \{\sigma(S)(\rho) \leq \tau\} & Assign(S, \rho{:}\tau) \in C \end{array}$$

Given a program of size n, in the worst case this naive algorithm requires at least n^2 space and time to build $\sigma(\cdot)$ and generate the necessary type constraints. This cost is too high for all but small examples. We reduce this cost in practice by taking advantage of several observations.

Many locations are flow-insensitive. If a location ρ never appears on the left-hand side of an assignment, then ρ's type cannot change. Thus we can give ρ one global type instead of one type per program point. In imperative languages such as C, C++, and Java, function parameters are a major source of flow-insensitive locations. In these languages, because parameters are *l*-values, they have an associated memory location that is initialized but then often never subsequently changed.

Adding extra store variables trades space for time. To compute $\sigma(S)(\rho)$ for a constructed store S, we must deconstruct S recursively until we reach a store variable or an assignment to ρ (see Figure 4.16a). Because the effect inference algorithm represents effect constraints compactly, deconstructing $\mathit{Filter}(\cdot, L)$ or $\mathit{Merge}(\cdot, \cdot, L)$ may require a potentially linear time computation to check whether $\rho \in L$ (recall the algorithm in Figure 4.10). We recover efficient lookups by replacing S with a fresh store variable ε and adding the constraint $S \leq \varepsilon$. Then rather than computing $\sigma(S)(\rho)$ we compute $\sigma(\varepsilon)(\rho)$, which requires only a map lookup. Of course, we must use space to store ρ in $\sigma(\varepsilon)$. However, as shown below, we often can avoid this cost completely. We apply this transformation to each store $\mathit{Merge}(S, S', L)$ constructed during constraint inference.

Not every store needs every location. Rather than assuming $\sigma(\varepsilon)$ contains all locations, we add needed locations lazily. We add a location ρ to $\sigma(\varepsilon)$ the first time the analysis requests $\varepsilon(\rho)$ and whenever there is a constraint $S \leq \varepsilon$ or $\varepsilon \leq S$ such that $\rho \in \sigma(S)$. Stores constructed with *Filter* and *Merge* tend to stop propagation of locations, saving space. For example, if $\mathit{Filter}(S, L) \leq \varepsilon$ and $\rho \in \sigma(\varepsilon)$, but $\rho \notin L$, then we do not propagate ρ to S.

We can extend this idea further. For each qualifier variable κ, the qualifier constraint resolution algorithm in Figure 3.8 maintains a set of possible qualifier constants that are valid solutions for κ. If that set contains every qualifier constant, then κ is *uninteresting* (i.e., κ is constrained only by other qualifier variables). Otherwise κ is *interesting*. A type τ is interesting if any qualifier in τ is interesting, otherwise τ is uninteresting. We then modify the closure rules as follows:

$$\begin{array}{rcll} C \cup \{S \leq \varepsilon\} & \Longrightarrow & C \cup \{S \leq \varepsilon\} \cup \{\sigma(S)(\rho) \leq \sigma(\varepsilon)(\rho)\} & \\ & & \text{for all } \rho \text{ such that } \sigma(S)(\rho) \text{ or } \sigma(\varepsilon)(\rho) \text{ interesting} & \\ C \cup \{\omega \leq \sigma(S)_{lin}(\rho)\} & \Longrightarrow & C \cup \{\omega \leq \sigma(S)_{lin}(\rho)\} \cup \{\sigma(S)(\rho) \leq \tau\} & \mathit{Assign}(S, \rho{:}\tau) \in C \\ & & \sigma(S)(\rho) \text{ or } \tau \text{ interesting} & \end{array}$$

In this way, if a location ρ is bound to an uninteresting type, then we need not propagate ρ through the constraint graph.

Figures 4.19 and 4.20 give an algorithm for lazy location propagation. We associate a mark with each ρ in each $\sigma(\varepsilon)$ and with ρ in $\mathit{Assign}(S, \rho{:}\tau)$. Initially this mark is not set, indicating that location ρ is bound to an uninteresting type. If a qualifier variable κ appears in $\sigma(\varepsilon)(\rho)$, we associate the pair (ρ, ε) with κ, and similarly for stores constructed

```
PROPAGATE(ρ, S) =
  switch
    case S = ε :
      add ρ : embed'(S_I(ρ)) to σ(ε) if not already in σ(ε)
      if ρ is not marked in σ(ε)
        then  mark ρ in σ(ε)
          FORWARD-PROP(ε, ρ, σ(ε)(ρ))
          for each S' such that S' ≤ ε do
            BACK-PROP(S', ρ, σ(ε)(ρ))
    case S = Assign(S', ρ:τ) :
      if ρ is not marked in σ(Assign(S', ρ:τ))
        then  mark ρ in σ(Assign(S', ρ:τ))
          FORWARD-PROP(S, ρ, τ)
```

Figure 4.19: Lazy Constraint Propagation

with *Assign*. If during constraint resolution the set of possible solutions of κ changes, we call PROPAGATE(ρ, S) to propagate ρ, and in turn κ, through the store constraint graph.

If PROPAGATE(ρ, C) is called and ρ is already marked in C, we do nothing. Otherwise, BACK-PROP() and FORWARD-PROP() make appropriate constraints between $\sigma(S)(\rho)$ and $\sigma(S')(\rho)$ for every store S' reachable from S. This step may add ρ to S' if S' is a store variable, and the type constraints that BACK-PROP() and FORWARD-PROP() generate may trigger subsequent calls to PROPAGATE().

Example 8. Consider again the example from Figure 4.5. The constraints generated for this program are shown in Figure 4.18. Figure 4.21 shows how locations and qualifiers propagate through this store constraint graph. Dotted edges in this graph indicate inferred constraints.

The four type constraints in Figure 4.17 are shown as directed edges in Figure 4.21. For example, the constraint $\kappa_0\ \ int \leq \varepsilon(\rho_x)$ reduces to the constraint $\kappa_0 \leq \kappa_x$, which is a directed edge $\kappa_0 \rightarrow \kappa_x$. Adding this constraint does not cause any propagation; this constraint is among variables. Notice that the assignment of type $\kappa_3\ \ int$ to ρ_x also does not cause any propagation.

The constraint $\mathsf{a}\ \ int \leq Alloc(\varepsilon, \rho_x)(\rho_y)$ reduces to $\mathsf{a}\ \ int \leq \varepsilon(\rho_y)$, which reduces to $\mathsf{a} \leq \kappa_y$. This constraint does trigger propagation. PROPAGATE(ρ_y, ε) first pushes ρ_y

```
BACK-PROP(S, ρ, τ) =
  switch
    case S = ε :
      add ρ : embed′(S_I(ρ)) to σ(ε) if not already in σ(ε)
      σ(ε)(ρ) ≤ τ
    case S = Alloc(S′, ρ′) :
      BACK-PROP(S′, ρ, τ)
    case S = Merge(S′, S″, L) :
      if ρ ∈ L
        then BACK-PROP(S′, ρ, τ)
        else BACK-PROP(S″, ρ, τ)
    case S = Filter(S′, L) :
      if ρ ∈ L
        then BACK-PROP(S′, ρ, τ)
    case S = Assign(S′, ρ′:τ′) :
      if ρ = ρ′
        then τ′ ≤ τ
        else BACK-PROP(S′, ρ, τ)


FORWARD-PROP(S, ρ, τ) =
  for each ε such that S ≤ ε do
    add ρ : embed′(S_I(ρ)) to σ(ε) if not already in σ(ε)
    τ ≤ σ(ε)(ρ)
  for each S′ such that S′ is constructed from S do
    switch
      case S′ = Alloc(S, ρ′) :
        FORWARD-PROP(S′, ρ, τ)
      case S′ = Merge(S_1, S_2, L) :
        if ρ ∈ L and S = S_1
          then FORWARD-PROP(S′, ρ, τ)
        if ρ ∉ L and S = S_2
          then FORWARD-PROP(S′, ρ, τ)
      case S′ = Filter(S, L) :
        if ρ ∈ L
          then FORWARD-PROP(S′, ρ, τ)
      case S′ = Assign(S, ρ′:τ′) :
        if ρ ≠ ρ′
          then FORWARD-PROP(S′, ρ, τ)
```

Figure 4.20: Lazy Location Propagation Subroutines

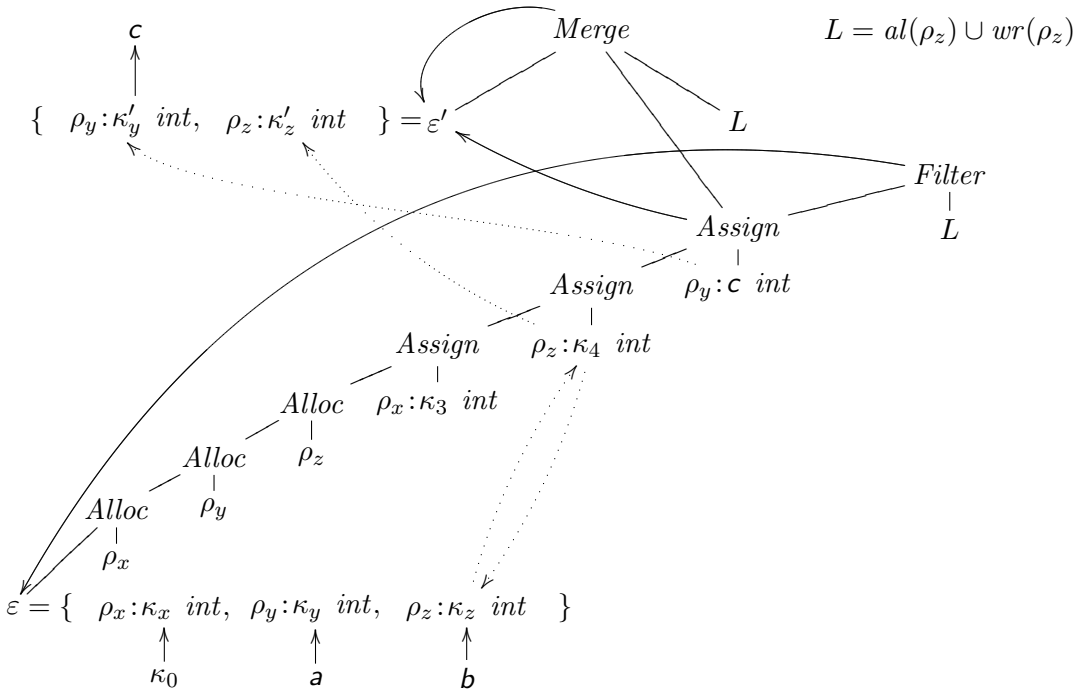

Figure 4.21: Constraint Resolution for Figure 4.18

backward to the *Filter* store. But since $\rho_y \notin L$, propagation stops. Next we push ρ_y forward through the graph and stop when we reach the store $Assign(\cdot, \rho_y : \mathsf{c}\ \ int)$; forward propagation assumes that this is a strong update.

Since $Assign(\cdot, \rho_y : \mathsf{c}\ \ int)$ contains an interesting type, ρ_y is propagated from this store forward through the graph. On one path, propagation stops at the *Filter* store. The other paths yield a constraint $\mathsf{c} \leq \kappa'_y$. Notice that the constraint $\kappa'_y \leq \mathsf{c}$ remains satisfiable.

The constraint $\mathsf{b} \leq \kappa_z$ triggers a propagation step as before. However, this time $\kappa_z \in L$, and during backward propagation when we reach *Filter* we must continue. Eventually we reach $Assign(\cdot, \rho_z : \kappa_4\ \ int)$ and add the constraint $\kappa_4 \leq \kappa_z$. This in turn triggers propagation from $Assign(\cdot, \rho_z : \kappa_4\ \ int)$. This propagation step reaches ε', adds ρ_z to $S(\varepsilon')$, and generates the constraint $\kappa_4 \leq \kappa'_z$.

Finally, we determine that in the *Assign* stores ρ_x and ρ_y are linear and ρ_z is non-linear. Thus the update to ρ_z is a weak update, which yields a constraint $\kappa_z \leq \kappa_4$. □

This example illustrates three kinds of propagation. The location ρ_x is never interesting, so it is not propagated through the graph. The location ρ_y is propagated, but propagation stops at the strong update to ρ_y and also at the *Filter*, because the (Down_a) rule in Figure 4.4 is able to prove that ρ_y is purely local to f. The location ρ_z, on the other hand, is not purely local to f, and thus all instances of ρ_z are conflated, and ρ_z admits only weak updates.

4.6 Related Work

In this section we discuss work related to our flow-sensitive type qualifier framework and inference system. We delay discussion of most of the related program analysis systems and tools until Section 5.6. The flow-sensitive type qualifier system presented here was previously described by us [45].

The unification-based alias analysis we use in this chapter is particularly simple. Many other, more precise alias analyses have been proposed in the literature, some of which scale to large programs [5, 22, 32, 69, 70, 108, 119] (to list only a few). Because our system includes `restrict`, we are able to recover from the relatively weak alias analysis we use. Nevertheless, we believe that any alias analysis system can be incorporated into our framework, at the cost of increased complexity. Research suggests that the usefulness

of more precise alias analysis may be mixed [44, 96, 126]. The Lackwit [85] and Ajax [84] tools use polymorphic unification-based alias analysis as the basis of program understanding tools.

Our flow-sensitive type qualifier system differs from classical dataflow analysis [3, 67] in several ways. First, we generate constraints over stores and use types to model the program. Thus there is no distinction between forward and backward analysis; information may flow in both directions during constraint resolution, depending on the specified qualifier partial order. Second, we explicitly handle pointers, heap-allocated data, aliasing, and strong/weak updates. Third, there is no distinction between interprocedural and intraprocedural analysis in our system.

Olender and Osterweil propose using dataflow analysis to check sequencing constraints [86]. Their system includes an interprocedural component but does not model aliasing. Horwitz et al [61, 97] and Duesterwald et al [28, 29] have proposed frameworks for demand-driven, interprocedural dataflow analysis. As we have also found, Horwitz et al and Duesterwald et al discovered that demand-driven (in our case, lazy) analysis is significantly more efficient than naive, exhaustive analysis.

The strong/weak update distinction was first described by Chase et al [17]. Several researchers have proposed techniques that allow strong updates for dataflow-based analysis of programs with pointers, among them Altucher and Landi [4], Emami et al [32], and Wilson and Lam [119]. Jagannathan et al [66] present a system for must-alias analysis of higher-order languages. The linearity computation in our system corresponds to their singleness computation, and they use a similar technique to gain polymorphism by flowing some bindings around function calls.

Our use of store constraints and linearities is inspired by the flow-sensitive type system for the calculus of capabilities [21] and the subsequent work on alias types [104, 115, 116]. The key difference between these systems and ours is that they are designed primarily for checking, while our system focuses on inference. Linearities have also been used to allow in-place updates in purely functional languages [11, 112].

The alias analysis step in our system can be seen as a form of region inference [111]. Intuitively, each abstract location determined by our alias analysis represents a region, and we can strongly update a location if it is linear, i.e., if its region contains only one run-time location.

For a comparison of `restrict` to ANSI C's type qualifier of the same name, see

Section 5.5. The Vault programming language includes "adoption" and "focus" language constructs, which are similar to a flow-sensitive version of `restrict` [36].

The type state system of NIL [109] is one of the earliest type systems to incorporate flow-sensitivity. NIL forbids aliasing, making the task of checking flow-sensitive properties somewhat different than in our system. Type systems for low-level programs [123] and for Java byte code [83, 107] also incorporate flow-sensitivity to check for initialization before use and to allow reuse of the same local variable at different types.

Igarashi and Kobayashi [65] propose a general framework for resource usage analysis, which associates a trace with each object specifying valid accesses to the object. The resource usage problem is to check that the program satisfies the trace specifications. The kinds of properties checkable in their framework [65] are similar to the checks possible with flow-sensitive type qualifiers. Igarashi and Kobayashi provide an inference algorithm that appears to be at least quadratic in practice. It also invokes as a sub-step an unspecified algorithm to check that a trace set is valid.

Chapter 5

CQual

To test the ideas described in the preceding chapters, we have built a tool called CQUAL that adds both flow-insensitive and flow-sensitive type qualifiers to the C programming language. To use CQUAL, programmers annotate their C programs with type qualifiers, and then CQUAL performs flow-insensitive and, if necessary, flow-sensitive type qualifier inference. Inference results are presented to the user with an Emacs-based user interface [56]. A web-based version of CQUAL is also available. In this chapter we discuss the issues involved in analyzing C code and some of the choices we made in designing CQUAL. We believe that the lessons learned while developing CQUAL are applicable to other languages as well. Chapter 6 describes a series of experiments using CQUAL.

Figure 5.1 gives an overview of the architecture of CQUAL. The input program is passed to a C front end that parses the program and performs standard C type checking. CQUAL uses the front-end from the RC region compiler [50]. The abstract syntax tree and a configuration file (described below) are fed into the first, flow-insensitive phase of the analysis, which performs flow-insensitive type qualifier inference (Section 3.4), alias analysis, effect constraint generation, and `restrict` checking (Section 4.3). The second, flow-sensitive phase of the analysis computes linearities and performs flow-sensitive type qualifier inference (Section 4.5).

Although in Chapters 3 and 4 we describe constraint resolution as happening after constraint generation, in practice we solve many of the constraints as they are generated. The exception is that in the flow-sensitive phase, we delay inferring linearities, and the weak update constraints produced for non-linear locations, until after all constraints have been generated. Whenever we generate an inconsistent constraint while analyzing a particular

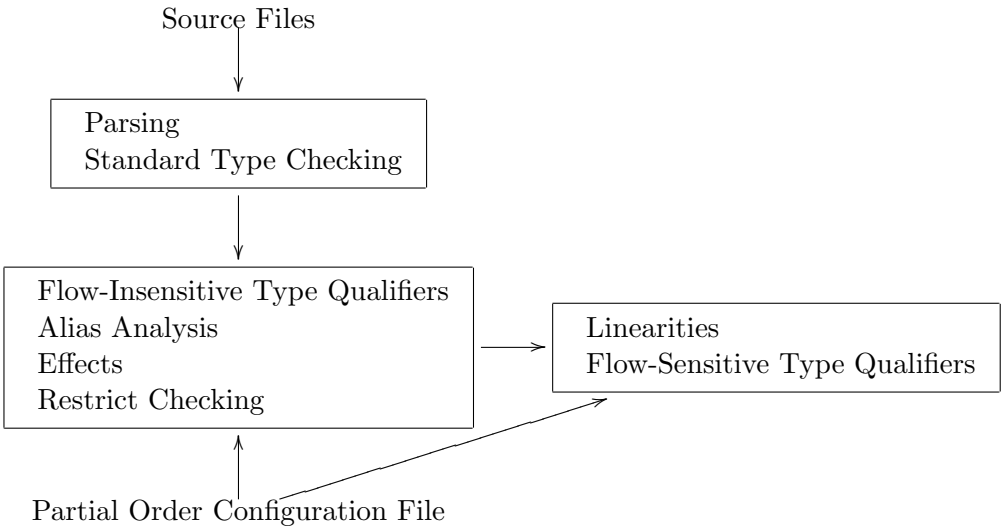

Figure 5.1: CQUAL System Architecture

expression in the source code, we report an error at the position of the expression.

5.1 Syntactic Issues and Partial Order Configuration Files

The first issue in adding type qualifiers to any language is incorporating them into the surface syntax. To avoid potential conflicts, we can require that all qualifiers begin with a reserved symbol, so that the lexer can unambiguously tokenize qualifiers. In CQUAL we require that all qualifiers except those standard in ANSI C begin with a dollar sign (for example, `$YYYY`, `$tainted`, `$locked`, etc.).[1] Many C compilers do not allow identifiers to begin with dollar signs, hence interpreting all such identifiers as qualifiers poses little problem for most source programs. We extend the grammar for types in our source language so that a set of qualifiers can appear on all levels of a type. (We allow a set instead of only a single qualifier to make CQUAL easier to use; see below.) For C adding qualifiers to types is particularly easy, as ANSI C contains three built-in type qualifiers `const`, `volatile`, and `restrict` already. Thus we simply extend the C grammar for qualifiers to include any identifiers beginning with a dollar sign. If we want to apply a standard language tool such as a compiler to a program annotated with qualifiers, we can simply remove, automatically, all of the qualifiers beginning with dollar signs. In our exposition we omit the dollar signs

[1] An alternative approach would be to use compiler-specific source code extensions, for example, `gcc`'s `attribute` syntax.

to make the text more readable. Instead we continue to use slanted text to denote qualifiers (for example, *yyyy*, *tainted*, *locked*, *const*, *volatile*, etc.)

In addition to adding qualifiers to the source language, we also add qualifier annotations and checks. Since C already requires that programs be annotated with types, new qualifier annotations and checks are largely unnecessary. Qualifier annotations roughly correspond to types in variable definitions. For example, a tainted integer `x` can be defined with

tainted `int x;`

Formally, type qualifier inference associates a qualifier variable x with `x`,[2] and we view the occurrence of *tainted* as placing a constraint on qualifier variable x (see below). Checks roughly correspond to type annotations on function parameters. For example, a function that requires untainted data can be declared as

`void f(`*untainted* `int y);`

For flow-sensitive analysis, sometimes it is convenient to specify qualifier properties at arbitrary program points in addition to declarations. We add two new kinds of statements to the language to make this easier:

```
assert_type(e, T);
change_type(e, T);
```

The first statement checks that at the current program point `e` has type `T`. The second statement models an assignment statement without giving a concrete right-hand side. The statement `change_type(e, T)` type checks exactly like the assignment `e:=`e' where e' is an expression of type `T`. In particular, we use this form in Section 6.3 to annotate Linux kernel locking functions, which are written with in-line assembly code.

Because all non-standard qualifiers must begin with a dollar-sign, it is easy to remove them automatically from a program so that it can be accepted by a standard C compiler. We can eliminate `assert_type(e, T)` and `change_type(e, T)` statements by simply `#define`-ing them away when the input file is passed to a standard C compiler. Usually this can be achieved with command-line options, requiring no changes to the actual source code.

[2]The variable `x` may actually have two inferred qualifiers; see Section 5.2.

```
po-defn   ::=  partial order [ po-opt* ]? { po-entry* }
po-opt    ::=  flow-insensitive
           |   flow-sensitive
           |   nonprop
po-entry  ::=  qual-name [ qual-opt* ]?
           |   qual-name < qual-name
qual-opt  ::=  color = "color-name"
           |   level = ref
           |   level = value
           |   sign = pos
           |   sign = neg
           |   sign = eq
```

Figure 5.2: Partial Order Configuration File Grammar

By itself, adding qualifiers to the surface syntax is not quite sufficient. We also need to know the partial order among the qualifiers and how to interpret occurrences of qualifiers in the surface syntax. For example, suppose we see a declaration `a int x`, where `a` is some qualifier constant. If x is `x`'s associated qualifier variable, how should x and `a` be related: $x \leq$ `a` or `a` $\leq x$, or both? To use CQUAL, the programmer must supply a *partial order configuration file* listing the qualifiers and their partial order. The partial order configuration file also declares, for each qualifier, its *variance*: whether it is *positive*, *negative*, or *non-variant*. The declaration `a int x` yields `a` $\leq x$ if `a` is positive; $x \leq$ `a` if `a` is negative; and $x =$ `a` if `a` is non-variant. Intuitively, positive qualifiers are used for annotations, and negative qualifiers are used for checks. Non-variant qualifiers may be used for both.

The user may specify several orthogonal sets of qualifiers, each with their own partial order, within the partial order configuration file. As mentioned in Chapter 3, we can combine these into a single partial order by taking their cross product. Conceptually, each qualifier variable x created during inference can be seen as a tuple with one component x_i for each of the specified partial orders. As mentioned above, we allow a set of qualifiers to appear at each level of a type. Suppose that qualifier a comes from the ith specified partial order. Then when we see a declaration `a int x`, we view the occurrence of `a` as placing a constraint on x_i, and placing no constraint on the other elements of the tuple.

Figure 5.2 gives the complete grammar for CQUAL's partial order configuration files. In this grammar, x^* means zero or more occurrences of `x`, and `[` x `]`$^?$ means either

```
partial order {
   const [level = ref, sign = pos]
   $nonconst [level = ref, sign = neg]

   $nonconst < const
}

partial order {
   $untainted [level = value, color = "pam-color-untainted", sign = neg]
   $tainted [level = value, color = "pam-color-tainted", sign = pos]

   $untainted < $tainted
}

partial order [flow-sensitive] {
   $locked [level = value, color = "pam-color-locked", sign = eq]
   $unlocked [level = value, color = "pam-color-unlocked", sign = eq]
}
```

Figure 5.3: Example Partial Order Configuration File

zero or one occurrence of `[` x `]`. Each partial order can be declared to contain flow-insensitive qualifiers (Chapter 3), flow-sensitive qualifiers (Chapter 4), or non-propagating qualifiers, which should not be inferred. The canonical example of a non-propagating qualifier is *restrict*, which has a special meaning in our system (see Section 5.5). For each partial order the users lists the qualifiers and their options. The `sign` option specifies the variance of a qualifier: positive (`pos`), negative (`neg`), or non-variant (`eq`). The `level` options are explained in Section 5.2, and the `color` option in Section 5.4. Finally, the partial order is specified by declarations $a < b$ for each pair of qualifiers so related in the partial order. We compute the reflexive transitive closure of the specified relations to yield the final partial order. Figures 5.3 and 5.4 give a partial order configuration file for the qualifiers discussed in Chapter 6.

5.2 Modeling C Types

In this section we discuss some of the issues in handling C types as they are used in C programs, which is somewhat messier than the idealized types given in Chapters 3

```
partial order [flow-sensitive] {
   $readwrite_unchecked [sign = eq, color = "pam-color-8"]
   $read_unchecked [sign = eq, color = "pam-color-8"]
   $write_unchecked [sign = eq, color = "pam-color-8"]
   $open_unchecked [sign = eq, color = "pam-color-8"]

   $readwrite [sign = eq, color = "pam-color-8"]
   $read [sign = eq, color = "pam-color-8"]
   $write [sign = eq, color = "pam-color-8"]
   $open [sign = eq, color = "pam-color-8"]

   $closed [sign = eq, color = "pam-color-8"]

   $readwrite_unchecked < $read_unchecked
   $readwrite_unchecked < $write_unchecked
   $read_unchecked < $open_unchecked
   $write_unchecked < $open_unchecked

   $closed < $readwrite_unchecked

   $readwrite < $read
   $readwrite < $write
   $read < $open
   $write < $open

   $open < $open_unchecked
   $read < $read_unchecked
   $write < $write_unchecked
   $readwrite < $readwrite_unchecked
}
```

Figure 5.4: Example Partial Order Configuration File (continued)

and 4.

***L*-Types and *R*-Types.** In C there is an important distinction between l-values, which correspond to memory locations, and r-values, which are ordinary values like integers. In the C type system, l-values and r-values are given the same type. For example, consider the following code:

```
int x;
x = ...;
...  = x;
```

The first line defines the variable `x` as a location containing an integer. On the second line `x` is used as an l-value: it appears on the left-hand side of an assignment, meaning that the location corresponding to `x` should be updated. On the third line `x` is used as an r-value. Here when we use `x` as an r-value we are not referring to the location `x`, but to `x`'s contents. In the C type system, `x` is given the type `int` in both places, and the syntax distinguishes integers that are l-values from integers that are r-values.

CQUAL takes a slightly different approach in which the types distinguish l-values and r-values. The variable `x` (ignoring qualifiers for a moment) is given the type *ref* (*int*), meaning that the name `x` is a location containing an integer. When `x` is used as an l-value its type stays the same—the left-hand side of an assignment is always a *ref* type. When `x` is used as an r-value the outermost *ref* is removed, i.e., `x` as an r-value has the type *int*.

But now the question again arises of how to interpret qualifiers in the surface syntax. Suppose we see a declaration *a* `int x;`. Then we assign `x` the type x *ref* (x' *int*), where x and x' are qualifier variables. Should we interpret the occurrence of *a* as constraining x or as constraining x'? In CQUAL, the user must specify this in the partial order configuration file. The qualifier *a* may be declared to constrain x, i.e., the *ref* level (`level = ref` in the configuration file), or it may be declared to constrain x', i.e., the *int* level (`level = value`). Most qualifiers constrain the value level of a type; for example, *tainted* and *untainted* behave this way. The canonical example of a qualifier that constrains the *ref* level of a type is *const*; see Section 6.1.

Finally, it is worth mentioning that arguments in C are passed by value, and hence function types contain the r-types of their declared parameters, even though within a function the parameter is treated as having an l-type. For example, given the declaration `void f(int x)`, we assign `f` the type x' *int* $\longrightarrow$ *void* (ignoring qualifiers except on `x`), and

within the body of `f` the variable `x` has type $x\ ref\ (x'\ int)$.

Structures. Aside from deciding how to model *l*- and *r*-values, there are other important considerations when modeling types in C programs. One of the key considerations in any whole-program analysis of C code is how structures (record types) are modeled [16, 57, 125]. Suppose that the user declares a structure:

```
struct foo {
    int x;
    int *y;
    ...
}
```

Then in theory, if we see two definitions `struct foo a` and `struct foo b` we can simply assign `a` and `b` two distinct copies of the type `struct foo`. Unfortunately, in practice this turns out to be prohibitively expensive. If `struct foo` has n fields and we assign each instance of `struct foo` fresh copies of the types of its fields, then we are doing $O(n^2)$ work. Since many C programs contain extremely long structure type declarations, n can be relatively large, causing a large slowdown in type qualifier inference. Instead, we choose to share structure fields among different instances of the same `struct`. Given the above declaration of `struct foo` and definitions `struct foo a` and `struct foo b`, we assign a single type to `a.x` and `b.x`, and similarly to `a.y` and `b.y`.

On the other hand, `typedef`s, which allow the programmer to name specific types, do not tend to be particularly large, and so we do not share qualifiers on `typedef`'d types.

Multiple Files. Very few C programs are contained within a single source file, thus CQUAL is designed to perform type qualifier inference on multiple files simultaneously. We require that globals declared in multiple files have the same type, which can be achieved by unifying their types. In ANSI C equivalence of `struct` types is by name; thus even if `struct foo` and `struct bar` are declared in a file with exactly the same fields, they are considered different types. In order to analyze multiple files, we must relax this restriction across files, and so instead we perform structural matching on `struct` and `union` types to determine equivalence. Note that we do not require that the programmer analyze all files of a program together. However, to get sound results when analyzing a single file CQUAL must be supplied with full qualified type declarations for any undefined globals.

Parametric Qualifier Polymorphism. *Parametric polymorphism* [77] is a powerful technique for increasing the precision of a type system. As discussed in Section 4.1.2, in our flow-sensitive analysis we use effects to gain some measure of polymorphism over locations. CQUAL also incorporates a limited form of polymorphism over type qualifiers, though only for flow-insensitive type qualifier inference. To understand the benefit of polymorphism over qualifiers, suppose we have qualifier constants a and b with partial order $b < a$, and consider the following two function definitions:

```
a int id1(a int x) { return x; }
b int id2(b int x) { return x; }
```

We would like to have only a single copy of this function, since both versions behave identically and in fact compile to the same code. Unfortunately, without polymorphism we need both. An object of type `b int` can be passed to `id1`, but the return value has qualifier a. An object of type `a int` cannot be passed to `id2` without a type cast. The problem here is that the type of the identity function on integer is $Q\ \mathit{int} \longrightarrow Q\ \mathit{int}$ with Q appearing both covariantly and contravariantly (recall our rule for subtyping function types in Figure 3.2).

We solve this problem by observing that the identity function behaves the same for any qualifier Q. We specify this in type notation with the polymorphic type signature $\forall\kappa.\kappa\ \mathit{int} \longrightarrow \kappa\ \mathit{int}$. When we apply a function of this type to an argument, we first *instantiate* its type at a particular qualifier, in our case either as $a\ \mathit{int} \longrightarrow a\ \mathit{int}$ or $b\ \mathit{int} \longrightarrow b\ \mathit{int}$. Intuitively instantiation corresponds exactly to function inlining, except we perform the inlining in our type system rather than in the source language.

In CQUAL, we allow the programmer to assign a flow-insensitive polymorphic type signature to a function. The signature is ignored during flow-sensitive type qualifier inference. The polymorphic type signature is assumed to be correct, and any function with a polymorphic type signature is not type checked. In its most general form, a type with polymorphic qualifiers is made up of a base type along with a set of subtyping constraints on the qualifier variables in the base type. For example, consider the C standard library function `char *strcat(char *dest, char *src)`, which appends `src` to the end of `dest` and returns `dest`. We can assign `strcat` the polymorphic type

$$\forall\kappa, \kappa'.\, \mathit{ref}\ (\kappa\ \mathit{char}) \times \mathit{ref}\ (\kappa'\ \mathit{char}) \longrightarrow \mathit{ref}\ (\kappa\ \mathit{char}) \text{ where } \kappa' \leq \kappa$$

which means that the qualifier on `strcat`'s second argument must be a subtype of its first argument, and that its first argument is returned. We omit the top-level *ref* qualifiers for

clarity. Here we use $\times$ to build a product type; we could also have written this as a curried function.

In the surface syntax, we declare this function with

```
$_1_2 char *strcat($_1_2 char *, $_1 char *);
```

The `$_1_2` and `$_1` are explicit qualifier variables represented as an unordered set of numbers. We use the names of the qualifier variables to encode the subtyping constraints. We generate the constraint $\kappa \leq \kappa'$ if the set encoded in the name of κ is a subset of the set encoded in the name of κ'. The existence of the Dedekind-MacNeille Completion [24] implies that any set of subtyping constraints can be encoded this way. While this is not the most transparent representation of subtyping constraints, it has the advantage of requiring no changes to the surface syntax.

Note that these limitations on polymorphism for flow-insensitive type qualifiers are not fundamental. By viewing type qualifiers as a label flow system, we can apply well-known techniques [80, 92] to perform fully automatic polymorphic flow-insensitive type qualifier inference. We defer such a system to future work.

Subtyping Under Pointer Types with const. In Section 3.6 we argued that if there are no updates through a reference, we can use the deep subtyping rule ($\text{Ref}'_{\leq}$) in Figure 3.10 rather than the conservative ($\text{Ref}_{\leq}$) in Figure 3.2. In ANSI C, programmers use *const* (Section 6.1) to annotate l-values that are never written. Thus in CQUAL we perform deep subtyping on locations explicitly annotated with *const* by the programmer. Although we could do so, we do not use the effect inference of Section 4.3 or the *const* inference of Section 6.1 to infer additional l-values that are not written to. Since *const* annotations are not available during the flow-sensitive portion of the analysis in our implementation, we do not apply this technique to flow-sensitive type qualifiers.

Flow-Sensitivity. We briefly mention a few special considerations in applying our flow-sensitive type qualifier inference to C. First, in CQUAL we do not allow strong updates to locations containing functions. This improves the efficiency of inference. Without this restriction, transforming a strong update on a location containing a function into a weak update could generate a store constraint, since function types contain stores. Then we

would need to recompute linearities (Section 4.5.1). By forbidding such strong updates, we increase the efficiency of the algorithm at a slight cost in precision.

Second, C programs can contain definitions of global variables that are allocated and initialized at load time. Thus our inference algorithm builds a global store S_G modeling the state of globals when the input program is loaded. If the programmer declares a `main` function, which is called by the operating system when the program is executed, then we generate a constraint $S_G \leq S$, where S is `main`'s initial store. If there is no such function, CQUAL prints a message warning that the initial state is discarded.

Third, recall that our alias and effect system in Figure 4.4 is more precise if we can show disinclusions of the form $\rho \notin \mathit{loceff}(\Gamma)$, where ρ is an abstract location and Γ is a type environment. Unfortunately, in C there is a single top-level environment containing the types of all functions declared in the program, and every statement of the program is type checked in an environment that extends Γ. Thus in our system it seems we cannot check $\rho \notin \mathit{loceff}(\Gamma)$ for any location ρ that is passed to a function.

However, ANSI C does not contain closures: functions may only be defined in the top-level scope, and not in internal scopes [6]. Intuitively, $\rho \notin \mathit{loceff}(\Gamma)$ is satisfied if none of the names in the domain of Γ provide a method for accessing location ρ. And without closures, functions—whether or not they mention ρ in their argument, result types, or effects—do not retain pointers to location ρ.[3] Thus when checking $\rho \notin \mathit{loceff}(\Gamma)$ in CQUAL, we safely ignore locations appearing in function types. We can formalize this by observing that all C functions are fully polymorphic [104] in the locations appearing in their type. Our system is monomorphic, however, in the sense that we always instantiate the same bound location in a particular function type to the same location.

5.3 Unsafe Features of C

The C programming language contains many features that allow the programmer to violate memory and type safety. Some of the major ones are type casts, unions, variable-argument functions, and arbitrary pointer arithmetic. CQUAL is based on the C types, and as such CQUAL obeys the type annotations in the program. As a result, CQUAL is sound only up to the unsafe features of C. For example, casting one type to another also casts away any type qualifiers and the abstract locations used in the alias analysis of Chapter 4.

[3] Local variables defined as `static` are treated as globals.

To be sound, the programmer should supply qualifier information whenever an unsafe feature of C is used. For example, at a type cast the programmer should explicitly annotate the cast-to type with qualifiers. For some qualifiers this is desirable behavior. For example, some type casts are added to programs exactly to cast away *const* qualifiers; hence it would be a bad idea to ignore such a cast. For other qualifiers, however, we should model the unsafe features of C more conservatively. For example, if we are tracking taintedness of data (see Section 6.2), tainting should not necessarily be removed at casts. In the remainder of this section, we discuss some of the unsafe features of C, and some techniques to model them more conservatively. Alternatively, CQUAL could be made sound by combining it with a system for enforcing memory safety, such as CCured [81].

Type Casts. Type casts allow a C programmer to treat a value as having any type they choose, which lets the programmer bypass limitations of the C type system. For example, a pointer to any type can legally be cast to and from a pointer to the special type *void*. Such casts are commonly used for generic functions on data structures. For example, a programmer may define a list data structure whose elements have type pointer to *void*, and then the same code for list operations can be used for lists of pointers to objects of any type.

The programmer can tell CQUAL to model such type casts by propagating qualifiers "through" the cast.[4] For example, consider the following code:

```
a char *y;
void *x = (void *) y;
```

This code declares `y` to be a pointer to character, where the character has qualifier *a*, and then initializes `x`, which is a pointer to *void*, with `y`. If the programmer tells CQUAL to propagate qualifiers through type casts, `x` is inferred to have type *a* `void *`. More generally, if we cast type τ to type τ', we generate a constraint $\tau \leq \tau'$ to propagate qualifiers through casts, where we allow matching between base types like *void* and *char*. If the programmer does not enable qualifier propagation through casts, this constraint is not generated, and type casts will then discard qualifiers.

[4] In CQUAL this does not apply to abstract locations, and thus our implementation of the alias analysis of Chapter 4 does not model casts. However, because all instances of the same `struct` type share field types (Section 5.2), the common case of the same pointer-to-structure type being cast to and from `void *` is modeled correctly.

Recall, however, that any pointer type may legally be cast to `void *`. For example, a programmer might write

```
a char **s;
char **t;
void *v = (void *) s;
t = (char **) v;
```

Here `s` and `t` are pointers to pointers to character. Notice that the type structure of `v` and the type structures of `s` and `t` do not even have the same shape. To model these kinds of casts, CQUAL "collapses" the mis-matched levels at type casts by equating their qualifiers. In resolving the constraint $\tau \leq \tau'$ generated at a cast from type τ to type τ', we allow τ and τ' to have different shapes. We add rules to conservatively equate qualifiers at shape mis-matches, such as

$$\frac{\tau = Q'\ void \qquad Q = Q'}{Q\ ref\ (\tau) \leq Q'\ void}$$

For our example above, inference determines that `v` has type `a void *`, and both `s` and `t` have type `a char *a *` (both levels of pointers get qualifier `a`).

Note that while this rule models casts to and from pointer types soundly, due to standard subtyping rules it does not model casts to and from base types soundly. For example, consider the following code:

```
char *x;                /* x :  x ref (x' char) */
char *y;                /* y :  y ref (y' char) */
int a;                  /* a :  a int */
int b;                  /* b :  b int */

a = (int) x;            /* (1) */
b = a;                  /* (2) */
y = (char *) b;         /* (3) */
```

We have given the qualified r-types for `x`, `y`, `a`, and `b`; we do not present the full l-types since they do not matter for this example. From line (1), using our modeling of casts we generate the constraints $x = x' = a$. From line (2) we generate the constraint $a \leq b$. Finally, from line (3) we generate the constraints $y = y' = b$. Putting these together, notice we have $x' \leq y'$ but not $y' \leq x'$—yet our rule for subtyping updatable references requires both, since

`x` and `y` refer to the same memory location. The problem is that we have cast an updatable reference type `char *` to an atomic type `int`, and our standard rule for subtyping atomic types assumes that `int`s do not allow indirect updates to memory. We could solve this problem by requiring equality rather than standard inclusion for `int` types, i.e., generating the constraint $a = b$ instead of $a \leq b$ at line (2). However, we have found in practice that the benefit of using standard inclusion for `int` types far outweighs the unsoundness introduced by such casts.

By the same token, we do not extend our modeling of type casts to equate qualifiers of structure fields at type casts, since if we do so the analysis becomes much too conservative in practice. However, recall that instances of the same structure type share fields (Section 5.2), hence casts to and from the same structure type are modeled soundly.

Sometimes casts to discard qualifiers are useful. We assume that any cast to a type that contains an explicit qualifier should stop qualifier propagation. For example, in the following code

```
a char *y;
void *x = (b void *) y;
```

the variable `x` is inferred to have qualifier *b* but not *a*. Such "trusted casts" are essential for making CQUAL usable in practice. There are always places where type systems are too conservative, and it is important to allow the programmer some mechanism for bypassing the type system.

Unions and Pointer Arithmetic. We make the same assumptions as the C standard about unions and pointer arithmetic. Namely, we model unions in the same way we model structures, and we assume that values of a union type are always accessed at the correct type with the correct qualifiers. We assume that pointer arithmetic does not violate object bounds, i.e., if p is a pointer to type τ, then we assume $p + i$ for any integer i also has type τ.

Libraries. Most C programs make some use of the extensive set of standard C libraries. Unfortunately, we do not necessarily have source code for library functions. Thus we require that the programmer supply a model for any library function that has an effect on the qualifiers. This model is usually a small stub function that mimics the behavior of the

library function with respect to the qualifiers. For flow-insensitive type qualifier inference, programmers may also supply polymorphic type signatures (Section 5.2) for functions in lieu of a stub function. In order to make it easy to identify library functions, CQUAL provides the programmer with a list of all globals that are used but never defined.

Variable-Argument Functions. In C, functions can be declared to take a variable number of arguments using the *varargs* language feature. One major problem with varargs functions is that there is no way to specify types for the variable arguments. CQUAL extends the grammar for C types to allow a qualifier constant or a polymorphic qualifier variable (Section 5.2) to be associated with the variable arguments. When the varargs function is called, we make constraints between that qualifier constant or polymorphic qualifier variable and all qualifiers on all levels of the actual arguments. To avoid unnecessary conservatism, we only generate such constraints for varargs functions that have explicitly marked varargs qualifiers. CQUAL provides a list of all undefined varargs functions to the user.

5.4 Presenting Qualifier Inference Results

Unlike traditional optimizing compiler technology, in order to be useful the results of the analysis performed by CQUAL must be presented to the user. We have found that in practice this often-overlooked aspect of program analysis is critically important—a user of CQUAL needs to know not only what was inferred but why it was inferred, especially when the analysis detects an error. To address this issue, CQUAL presents type qualifier inference results to the user via Program Analysis Mode (PAM) for Emacs [56]. PAM was developed concurrently with CQUAL, based on an earlier version that was part of the BANE toolkit [35]. PAM is a generic system for adding color markups and hyperlinks to program source code in Emacs. The ideas behind PAM can be adapted to many environments, and an experimental web-based client-server interface is also available.

After CQUAL analyzes the source programs, the user is presented with a buffer containing a list of the files that were analyzed and a list of any errors. Each file name in the buffer is a hyperlink to the start of the source file, and each error is a hyperlink to the line and column in the source code where the error was discovered, i.e., where the constraints generated by inference became unsatisfiable. When the user clicks on a hyperlink to bring up a file, the preprocessed source code of the file is colored according to the inferred qualifiers.

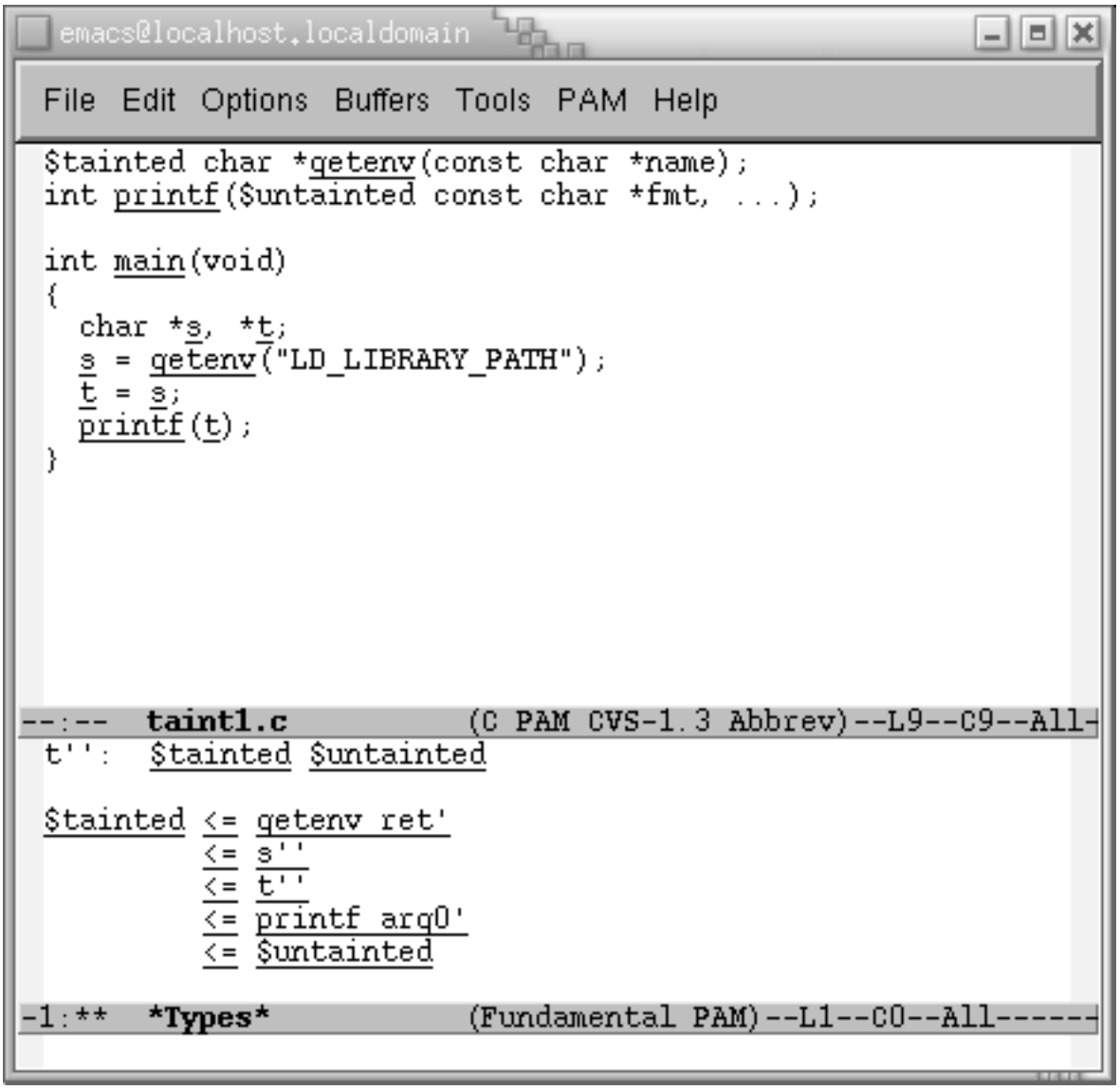

Figure 5.5: Sample Run of CQUAL

In particular, each qualifier can have an associated color in the partial order configuration file. If an identifier is inferred to have a particular qualifier, it is given that qualifier's color. CQUAL presents preprocessed source code because otherwise, due to C preprocessor macro expansions, jumping to particular line and column positions and marking up identifiers would not always be possible (for example, macro expansion can introduce new identifiers not present in the original source).

For each identifier in the program, CQUAL tries to show the user how its qualifiers

were inferred. Clicking on an identifier brings up its type and qualifier variables. Assuming the qualifier partial order is a lattice, clicking on a qualifier variable shows a path through the qualifier constraint graph that entails the inference result. Figure 5.5 shows a screen shot of CQUAL displaying such a path. In this example the result of `getenv` is annotated as *tainted*, and `printf` is annotated as taking an *untainted* first argument (see Section 6.2 for a discussion of these particular qualifiers). The result of `getenv` is passed to `s`, which is copied to `t`, which is passed as the first argument to `printf`. The screen shot in Figure 5.5 shows what happens when the user clicks on one of `t`'s qualifier variables t'': CQUAL presents the user with a path from *tainted* to t'' and from t'' to *untainted*. In this particular case this path indicates an error, since *tainted* $\not\leq$ *untainted* in our partial order. To make the paths even more useful, in CQUAL each element of the path, which represents a constraint, is hyperlinked to the position in the source code where that constraint was generated. In this way the programmer can step through a path one constraint at a time, viewing each line of source code that led to a particular inference result.

In general, for a given qualifier variable x, CQUAL presents the user with the shortest transitive paths (possibly bidirectional for non-variant qualifiers) from x to any qualifier constants appearing in x's solution. Clearly there could be many paths, some of which may be cyclic, from x to its bounds. We settled on presenting the shortest path as a way of reducing the burden on the user. In our experience, this heuristic is very important for usability.

One of the main problems in presenting analysis results is that for a large input program, there is a correspondingly large amount of information we may wish to present to the user. This information is usually represented compactly during analysis, but if represented textually it becomes extremely unwieldy. Clearly this is the case here: the constraint graph is relatively compact, but writing out all paths from qualifier variables to their bounds would be prohibitively expensive.

PAM sidesteps this problem completely by using a client-server architecture. PAM runs CQUAL as a subprocess. As the user clicks on hyperlinks in PAM buffers, PAM passes the click events to the CQUAL subprocess, which then sends commands to PAM to move the cursor position, display additional screens of information, and so on. In this way CQUAL maintains the inference results in its internal, compact form, and the results are presented verbosely only on demand by the user.

5.5 Comparison of Restrict to ANSI C

As mentioned earlier, our syntactic construct `restrict` $x = e_1$ `in` e_2 is inspired by the ANSI C type qualifier of the same name. In this section we discuss the differences between our construct and ANSI C's.

In CQUAL, we write `restrict` $x = e_1$ `in` e_2 using the same syntax as ANSI C:

```
{
    T *restrict x = e1;
    e2;
}
```

Here `x` is a pointer to type `T`. Note that this is why in CQUAL the *restrict* qualifier is non-propagating (Section 5.1)—it is really a kind of syntactic binding.

As described in Chapter 4, the user can add *restrict* to improve the precision of flow-sensitive type qualifier inference. In the ANSI C standard, in contrast, *restrict* is used to help the compiler optimize code [6]: if two pointers are declared *restrict*, they are guaranteed never to point to the same object, and hence reads and writes through the two pointers can be freely permuted.

The major difference between our type system and ANSI C is that in ANSI C *restrict* is not checked—the programmer is assumed to have added the *restrict* qualifier correctly. In addition to checking our version of *restrict*, the type system of Figure 4.4 can be used to check *restrict* as described in the ANSI C standard. If we wish to do so, several issues come up. Many of these issues have more to do with particular features of C than with programming language fundamentals.

Names. In C, most names refer to *l*-values, that is, *ref*-types in our notation. For example, if we declare

```
int *restrict p = ...;
```

then `p` may change during evaluation, which could change what object `p` points to. (Recall that in lambda calculus notation, the name x in `restrict` $x = e_1$ `in` e_2 is an *r*-value.) The standard is imprecise on this issue, suggesting that while it is invalid to update `p` to point to a different restricted value, it may be permissible to update `p` to a different but non-restricted value.

There are several solutions to this problem. The simplest solution is to require that all *restrict*-qualified pointers also be annotated with *const*, so that they cannot change. This is the solution followed in CQUAL. Equivalently, we can forbid writes to *restrict*-qualified pointers with effect constraints of the form $wr(\rho) \notin L$. A third solution, and the one most likely taken in a C compiler, is to perform a flow-sensitive analysis limited to a single function body to determine whether `p` is updated to point within the same object or whether it is updated with a new pointer (for example, `p++` versus `p = q`). The former may be allowed, and the latter should be forbidden.

Initialization. Our lambda calculus syntax for *restrict* forces the user to initialize restricted pointers as soon as they are declared. ANSI C has no such requirement. However, we feel that requiring restricted pointers to be initialized is not much of a burden, because the common case when *restrict* is used is for function parameters, and function parameters are always initialized.

Over-Estimation of Effects. The ANSI C standard defines *restrict* in a completely dynamic fashion. The accesses to object `X` within a block `B` are those that occur at run-time when `B` is executed. Since our analysis is static, we may over-estimate the set of locations accessed during evaluation, and hence we may fail to type check a program that executes correctly according to the standard.

Arrays. The ANSI C standard contains the following example of a valid use of *restrict* ([6], page 111):

```
void f(int n, int *restrict p, int *restrict q) {
    while (n-- > 0)
        *p++ = *q++;
}
void g(void) {
    extern int d[100];
    f(50, d + 50, d); // ok
}
```

In this example, the user has implicitly split the array `d` into two disjoint smaller arrays, and then called `f` knowing that as `f` traverses the arrays `p` and `q` it only accesses the first 50 elements of each. In our type system this program fails to type check, because this property—accessing only 50 elements of each array—cannot be deduced from the type of `f`. This application of *restrict* is useful in C and must be allowed. We feel that the best way to handle this situation in a manner consistent with C is to force the programmer to insert some kind of type cast at the call to *f()* to tell the compiler that `d[0..49]` and `d[50..99]` should be treated as disjoint objects. In our implementation, for example, this can achieved by calling `f(50, (int *) d + 50, (int *) d)`, since our alias analysis for checking *restrict* does not propagate abstract locations through type casts.

Escaping Pointers. The ANSI C standard explicitly allows certain pointers annotated with *restrict* to escape the scope of their declaration. Specifically, a function whose body declares a restricted pointer `p` may return the value of `p`. The only example of this in the standard is the definition of a function that returns a pointer to a structure, one of whose fields in annotated with *restrict*. Thus the motivation for allowing escaping restricted pointers seems to be to handle this case of *restrict* in a structure declaration. We believe that annotating structure fields with *restrict* is not well-defined in general (see below). Thus we do not support this usage, and our type system forbids restricted pointers from escaping entirely.

Data Structures. The ANSI C standard contains an example in which a `struct` contains a *restrict* qualifier ([6], page 112):

```
typedef struct {int n; float *restrict v;} vector;
```

This particular use of *restrict* can be encoded in our system and is semantically well-defined. The type `vector` is shorthand for a pair, and thus we can think of all operations on `vector`s as syntactic sugar for operations on the individual elements of the pair. Thus we can rewrite

```
vector x = { 3, a };
...  x.n ...  x.v ...
```

as

```
int x_n = 3;
float *restrict x_v = a;
...  x_n ...  x_v ...
```

On the other hand, uses of *restrict* in defining recursive data structures are problematic. For example, consider

```
struct list { int x; struct list *restrict next; };
```

What does *restrict* mean here? The problem is that the name `next` refers to a set of (possibly distinct) objects rather than a single object in memory. For example, if we construct a circular list `p`, then `p->next` and `p->next->next` may be the same. Is that forbidden by the *restrict* annotation? It seems not, because both accesses go through the name `next`. Clearly, though, a compiler cannot use the name `next` to infer anything about potential aliasing of list elements. Thus we feel that *restrict* annotations on recursively-defined data structures are not useful, and our implementation does not currently support *restrict* annotations on `struct`s.

A compiler needs stronger information than *restrict* to infer non-aliasing of heap objects. One such property is uniqueness of list elements [11, 112]. A *unique* object has at most one pointer to it at any time.[5] Thus if the `next` field of `struct list` were annotated as being unique, then a compiler could assume (and enforce) that each element of `p` is distinct, and a cyclic `struct list` would be forbidden.

Modified Objects. Our definition of *restrict* is slightly different than ANSI C's definition. Suppose that `p` is declared `int *restrict p` and that `p` points to object `X`. Then the ANSI C standard states that the *restrict* qualifier is only meaningful if `X` is modified within the scope of `p`. We refer to this as the *mod semantics* of *restrict*.

We consider the mod semantics of *restrict* unnecessarily complicated. The main reason we see to have the mod semantics is that for optimization purposes there is no benefit to *restrict* for locations that are not modified—optimizations must preserve read-write and write-write dependencies, but read-read dependencies can be safely ignored.

[5] This differs from the notion of a linear location defined in Chapter 4 because in our system the names of abstract locations are global. Thus a location can be linear no matter how many pointers to it there are. The downside is that the number of abstract locations in our system is finite.

However, from a language design point of view, it is undesirable to have a construct that is sometimes ignored. This is especially problematic if we think of *restrict* as an annotation to aid the programmer. For example, suppose we have the C declaration `f(int *restrict a, int *restrict b)`. Is it safe to call `f(x,x)`? Under the mod semantics we cannot tell without either knowing the effects of `f()` or looking at `f()`'s source code. We believe that rather than use the mod semantics, we should use our semantics and simply remove *restrict* qualifiers when they are unnecessary.

However, it is relatively easy to relax our type system in the case that a location is *restrict*ed but not written to, resulting in a system similar to the mod semantics. We replace the constraint $\rho \notin L_2$ in Figure 4.4 with constraints $wr(\rho) \notin L_2$ and $wr(\rho') \in L_2 \Rightarrow \rho \notin L_2$. The first constraint requires that ρ is never modified in e_2. This is because in the mod semantics if ρ is modified in e_2 then we require $\rho \notin L_2$, which is an immediate contradiction. The second constraint is satisfiable if either $wr(\rho') \notin L_2$ or if $\rho \notin L_2$. In other words, we only require that ρ is not read in e_2 if ρ' is modified. Note that we still do not allow ρ' to escape the scope of e_2, and note that ρ is still added to the effect of the *restrict*, so even with this change *restrict*ing non-written locations is not a no-op, unlike in the mod semantics.

Type inference proceeds as before, with the addition of replacing the constraint $wr(\rho') \in L_2 \Rightarrow \rho \notin L_2$ by $wr(\rho') \in \varepsilon \Rightarrow \rho \notin \varepsilon$ and $L_2 \subseteq \varepsilon$ for fresh ε. After checking satisfiability of the $\rho \notin \varepsilon$ constraints, we check the conditional constraints. For each constraint $wr(\rho') \in \varepsilon \Rightarrow \rho \notin \varepsilon$, we use the algorithm of Figure 4.10 to check whether $wr(\rho')$ reaches ε. If it does not, then the constraint is satisfiable. Otherwise, we check $\rho \notin \varepsilon$ with another call to the algorithm of Figure 4.10. Since there are $O(k)$ conditional constraints where k is the number of occurrences of *restrict* in the program, this step takes time $O(kn)$, where n is the size of the input program, and as before inference as a whole takes $O(n\alpha(n) + kn)$ time.

5.6 Related Work

In this section we discuss a number of related program analysis systems and techniques for improving software quality.

Two recent language proposals, Vault [26, 36] and Cyclone [52, 53], are extended safe variants of C that allow a programmer to enforce conditions on how resources are used in programs. For example, Vault has been used to check the safety of a floppy disk

driver [26]. Both Vault and Cyclone are inspired by the same flow-sensitive type systems that our framework is inspired a body of work by Crary, Morrisett, Smith, and Walker [21, 104, 115]. The key difference between our approach and Vault and Cyclone is that the latter are based on type checking and require programmers to annotate their programs with types. To make the annotation task easier, the languages are carefully designed to include compact notations and useful default annotations. In contrast, we propose a simpler and less expressive monomorphic type system that is designed for efficient type inference of both new and legacy code. Our system uses effects (Section 4.1.2) to gain a measure of polymorphism.

Several systems based on dataflow analysis have been proposed to statically check properties of source code. Evans's LCLint [34] introduces a number of additional qualifier-like annotations to C as an aid to debugging memory usage errors, and Evans found LCLint to be extremely valuable in practice [34]. LCLint has also been used to check for buffer overruns [71]. LCLint uses intraprocedural dataflow to analyze a function, using the programmer's annotations at function calls. LCLint uses a number of heuristics to model loops [71].

Meta-level compilation [33, 54] is a system for finding bugs in programs. The programmer specifies a flow-sensitive property as a finite state automaton. A program is analyzed by traversing its control paths and triggering state transitions of the automata on particular actions in program statements. The system warns of potential errors when an automaton enters an error state. Meta-level compilation includes an interprocedural dataflow component [54] but does not model aliasing. Meta-level compilation has been used to find many different kinds of bugs in programs.

Neither LCLint nor meta-level compilation is designed to be sound or complete; the goal of these tools is to find bugs, not prove the absence of bugs. In contrast, the ESP system [23] is based on sound dataflow analysis. Similarly to our flow-sensitive type qualifier system, ESP incorporates a conservative alias analysis. ESP also includes a path-sensitive symbolic execution component to model predicates. ESP has been used to check the correctness of C stream library usage in `gcc` [23]. See Section 6.4 for a discussion of checking file operations using CQUAL.

The Extended Static Checking (ESC) system [27, 42, 72] is a theorem-proving based tool for finding errors in programs. Programmers add preconditions, postconditions, and loop invariants to their program, and ESC uses sophisticated theorem proving tech-

nology to verify the annotations. ESC includes a rich annotation language; the Houdini assistant [41] can be used to reduce the burden of adding annotations.

SLAM [8, 9] and BLAST [59] are tools that verify software using model checking techniques. Both tools can track program state very precisely by modeling all paths separately. They are both based on predicate abstraction followed by successive refinement to make this process more tractable. Both SLAM and BLAST have been used to check properties of device drivers.

A number of techniques that are less easy to categorize have also been proposed. The AST toolkit provides a framework for posing user-specified queries on abstract syntax trees annotated with type information. The AST toolkit has been successfully used to uncover many bugs [118]. The PREfix tool [13], based on symbolic execution, is also highly effective at finding bugs in practice [88].

A number of systems have been proposed to check that implementations of data structures are correct. Graph types [68, 79] allow a programmer to specify the shape of a data structure and then check, with the addition of pre- and postconditions and loop invariants, that the shape is preserved by data structure operations. Shape analysis with three-valued logic [98] can also model data structure operations very precisely. Both of these techniques are designed to run on small inputs, and neither scales to large programs.

Chapter 6

Experiments

In this chapter we describe a number of experiments applying CQUAL to particular analysis and checking problems.

6.1 Const Inference

In ANSI C, the *const* qualifier can be added to types to specify that certain updatable references cannot, in fact, be updated. For example, if the programmer defines

```
const int x = 42;
```

then any assignment to `x`, such as `x = 3`, is forbidden. In other words, the left-hand side of an assignment must not be *const*. In this section we discuss checking and inferring ANSI C *const* qualifiers with CQUAL.

The main use of *const* is annotating the types of pointer-valued function parameters. Below is a table listing which assignments are allowed by the four possible placements of *const* on the type pointer to integer. Recall that C types are most easily read from right to left; thus, for instance, the second example below can be read as defining a pointer to a constant integer.

Definition	`p = ...;`	`*p = ...;`
`int *p`	valid	valid
const `int *p`	valid	invalid
`int *`*const* `p`	invalid	valid
const `int *`*const* `p`	invalid	invalid

Suppose that the programmer declares a function `void f(const int *p)`. Looking at our table, we see that the caller of `f` knows that, up to casting, `f` does not write through its argument `p`. This annotation is quite useful, since it means that one may freely pass pointers as arguments to `f` without fearing that the data they point to will be modified through `p`. Note that *const* does not guarantee that `*p` is not modified through other aliases, but only that it is not modified through `p`. Thus annotating `p` with *const* is different than saying there is no write effect (Section 4.3) on `*p`, since the latter does take aliasing into account.

To make *const* even more useful, ANSI C incorporates subtyping: non-*const* types can be used where *const* types are expected. In our system we express this by choosing *nonconst* $<$ *const* as the qualifier partial order, where *nonconst* is an explicit qualifier constant (written as whitespace in ANSI C) marking writable l-values. For example, consider again the definition of `f` above. With subtyping, both a *nonconst* `int *` and a *const* `int *` may be passed to `f`, and neither can be written to by `f` via `p`.

This subtyping on *const* pointer types is intuitive, but it seems quite suspicious on closer examination. If we pass a *nonconst* `int *` to a *const* `int *` position, then it looks like we are performing subtyping under a *ref*—and yet Section 3.1 contains a discussion explaining why such subtyping is unsound. A related question is what exactly a *const* `int` is. After all, `3` has type `int`, yet `3` cannot be updated; shouldn't `3` also be a *const* `int`?

The key to clearing up this confusion is to realize that *const* is an annotation on l-types (Section 5.2). Recall that if the programmer defines `int x`, then CQUAL assigns `x` the type (ignoring qualifiers) *ref* (*int*), meaning that `x` names an updatable reference containing an integer. In our system, if the programmer defines *const* `int x`, then `x` is assigned the type *const ref* (*int*)—not *ref* (*const int*). The *const* qualifier never appears on anything other than a *ref* constructor. Expressions that are only r-values, like the integer `3`, do not have a *ref* type, and thus *const* does not apply to them.

With this understanding of *const*, we see that the suspicious-looking subtyping under a pointer is completely standard. Figure 6.1 shows a small C program that assigns a pointer to *nonconst* to a pointer to *const* and the program's typing proof. Note that we add an explicit dereference of `x` in the typing proof; this dereference is implicit in the C program where `x` is used as an r-value. The main thing to observe in this proof is that in the subtyping step, there is no subtyping under a *ref* constructor.

To enforce the semantics of *const*, we can transform the input program by replacing each assignment statement e_1 `:=` e_2 with `check`$(e_1, \mathit{nonconst})$ `:=` e_2, which requires that the

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

(a) Example C Program

$$\dfrac{\dfrac{\Gamma \vdash_q x : \mathit{nc}\ \ \mathit{ref}\ (\mathit{nc}\ \ \mathit{ref}\ (\mathit{int}))}{\Gamma \vdash_q *x : \mathit{nc}\ \ \mathit{ref}\ (\mathit{int})} \quad \dfrac{\mathit{nc} \leq \mathit{c} \qquad \mathit{int} = \mathit{int}}{\Gamma \vdash_q *x : \mathit{c}\ \ \mathit{ref}\ (\mathit{int})}}{\Gamma \vdash_q y : \mathit{nc}\ \ \mathit{ref}\ (\mathit{c}\ \ \mathit{ref}\ (\mathit{int})) \qquad \Gamma \vdash_q y := *x : \mathit{nc}\ \ \mathit{ref}\ (\mathit{int})}$$

$$\Gamma = \{x \mapsto \mathit{nc}\ \ \mathit{ref}\ (\mathit{nc}\ \ \mathit{ref}\ (\mathit{int})), y \mapsto \mathit{nc}\ \ \mathit{ref}\ (\mathit{c}\ \ \mathit{ref}\ (\mathit{int}))\}$$
$$\mathit{c} = \mathit{const}, \mathit{nc} = \mathit{nonconst}$$

(b) Typing Proof

Figure 6.1: Const Subtyping with Pointers

left-hand side of every assignment is not *const*. Alternatively, we can modify the rule for assignment to require the same thing; here we present the modified inference rule:

$$\frac{\Gamma \vdash'_q e_1 : Q\ \ \mathit{ref}\ (\tau) \qquad \Gamma \vdash'_q e_2 : \tau' \qquad \tau' \leq \tau \qquad Q \leq \mathit{nonconst}}{\Gamma \vdash'_q e_1 := e_2 : \tau}$$

The latter is in fact what we do in CQUAL.

6.1.1 Experiments

Given an input program, we can assume that every position without a *const* qualifier has an implicit *nonconst* qualifier, just like a C compiler, and using our new rule for assignment CQUAL can check that a program uses *const* correctly. Unlike an ordinary C compiler, however, CQUAL can do better: we can use flow-insensitive type qualifier inference to infer *const* annotations.

A system that performs *const* inference has many benefits for the programmer. Although use of *const* is considered good programming style, it is well-known folklore that *const* is difficult to use in practice. Often an attempt to add a single *const* annotation to a program requires adding many other *const*s throughout the code. For the same reason, it

Name	Description	Lines	Preproc.	Declared	Inferred	Max
woman-3.0a	Manual page viewer	1496	8611	50	67	95
patch-2.5	Apply a diff	5303	11862	84	99	148
m4-1.4	Macro preprocessor	7741	18830	88	249	370
diffutils-2.7	Find diffs between files	8741	23237	153	209	372
ssh-1.2.26[2]	Secure shell	18620	127099	147	316	547
uucp-1.04	Unix to unix copy	36913	272680	433	1116	1773

Figure 6.2: Const Inference Results

can be difficult to mix code that uses *const* with code that does not. The result is that it is often easiest to simply omit *const* annotations altogether.

To perform *const* inference using CQUAL, we do not assume that any position without a *const* qualifier is *nonconst*. Instead we make no constraint on the qualifier variables in such positions. Then, given our new rule for assignment, we infer which qualifier variables must be *nonconst*. All of the remaining qualifier variables may be set to *const*. Note that we are actually computing the greatest solution of the generated qualifier constraints, since $\mathit{nonconst} < \mathit{const}$.

In Section 5.3 we describe some techniques for handling unsafe features of C. For purposes of these experiments we simply model unsafe features unsafely. We allow type casts to remove *const* qualifiers—we must do this, since many such casts are added precisely to remove *const*–and we do not perform any type checking on the extra arguments passed to varargs functions. We supply program stubs for library functions, and we make the conservative assumption that positions not marked *const* are indeed *nonconst* for such functions, and all fields of structures used by library functions as *nonconst*. In general library functions are annotated with as many *const*s as possible,[1] and so lack of *const* really does mean *nonconst*. We did not use any polymorphic qualifier annotations for these experiments, since our goal is to infer *const* annotations that can be checked by an ordinary C compiler.

We performed our *const* inference experiments using an earlier version of CQUAL based on the BANE constraint resolution library [35]. We selected six programs, listed to the

[1]...and sometimes more. For example, the `strchr` function is declared `char *strchr(const char *s, int c)`. The call `strchr(s, c)` returns a pointer somewhere in `s`, and yet the return type lacks *const*. This implicit cast is a way to emulate parametric qualifier polymorphism.

[2]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.

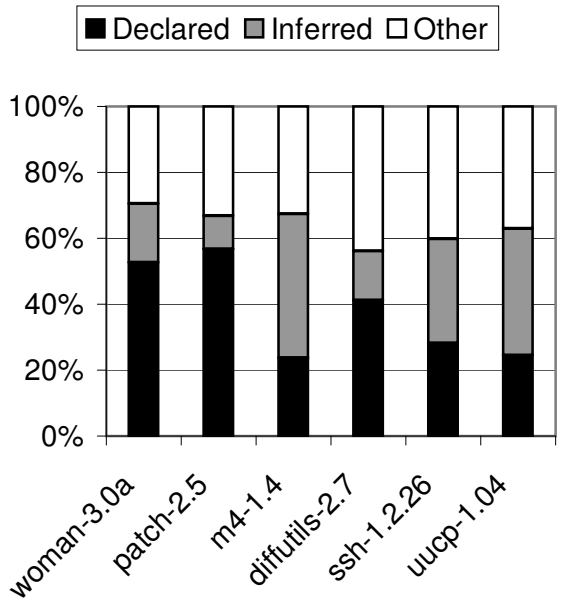

Figure 6.3: Graph of Const Inference Results

left in Figure 6.2, that make a significant effort to use *const*. Several of these "programs" are actually collections of programs that share a common code base. We analyzed each set of programs simultaneously, which occasionally required renaming as distinct certain functions that were defined in several files. For each benchmark, we measured the number of interesting *const*s inferred by CQUAL, where an *interesting* *const* is one that decorates a pointer r-type of a function parameter or result—these are the *const* annotations most likely to be useful to a programmer. For example, the function `int foo(int x, int *y)` has one interesting location where a *const* may be inferred, namely on the contents of `y`.

Figure 6.3 shows our results, which are tabulated in Figure 6.2. The fourth column of the table in Figure 6.2 lists the number of *const*s declared by the programmer in interesting positions. The fifth column lists the number inferred by *const* inference (which includes the explicitly specified ones), and the last column lists the total number of interesting positions. These measurements show that many more *const*s can be inferred than are typically present in a program, even one that makes a significant effort to use *const*. For some programs the results are quite dramatic, notably for `uucp-1.04`, which can have 2.5 times more *const*s than are actually present. An inspection of the code suggests that *const* is used consistently only in certain portions of the code, and that other parts of the code make no use of *const*. Additionally, the program uses several `typedef`s to define new

names for pointer types. Because we allow different instances of the same named type to have different qualifiers, we are able to infer that some uses of those pointer types can have *const* annotations. Clearly this is a case where *const* inference is very desirable. Faced with a program that heavily uses a single named type, few programmers would attempt to introduce a new type name with *const* annotations, but inference makes that process easy.

6.2 Format-String Vulnerabilities

Systems security is an ever more important problem as more critical services are connected to the Internet. Systems written in C are a particularly fruitful source of security problems, due to the tendency of C programmers to sacrifice safety for efficiency and the sometimes subtle interactions of C language and library features. One recently discovered class of C security problems is the so-called format-string vulnerability, which arises from the combination of unchecked variable argument (varargs) functions and standard C library implementations.

The standard ANSI C libraries contain a number of varargs functions that take as an argument a format specifier that gives the number and types of the additional arguments. For example, the standard printing function is declared as

```
int printf(const char *format, ...);
```

When `printf(format, a1, a2, ...)` is called, the string `format` is displayed with the ith format specifier replaced by extra argument `a`i. For example, here is the typical, correct way to print a string `buf`:

```
printf("%s", buf);
```

But for simply printing a string, the above construction appears at first to be unnecessarily verbose. A programmer can save themselves five characters—and possibly some whitespace—if they instead write

```
printf(buf);    /* may be incorrect */
```

Unfortunately, this innocuous-looking change may lead to security problems. If `buf` contains a format specifier (for example, `%s` or `%d`), perhaps supplied by a malicious adversary, `printf` attempts to read the corresponding argument off of the stack. Since there is no

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

int main(void)
{
    char *s, *t;
    s = getenv("LD_LIBRARY_PATH");
    t = s;
    printf(t);
}
```

Figure 6.4: Program with a Format-String Vulnerability

corresponding argument, `printf` will mostly likely crash, either when reading off the end of the stack or when it incorrectly interprets the garbage in the extra argument position as a pointer to a string in memory and attempts to display the string.

It turns out that format-string vulnerabilities are even worse than they first appear. Many implementations of the C standard libraries support the `%n` format specifier, which is now part of the ANSI C standard [6]. When a `printf`-like function encounters a `%n` format specifier, it writes through the corresponding argument, which must be a pointer, the number of characters printed so far. Given the ability to write to memory, a clever adversary can often exploit format-string vulnerabilities to completely compromise security—for example, to gain remote root access [82]. Since the ability to exploit format-string vulnerabilities was discovered in 2000, security experts and malicious attackers have discovered many such vulnerabilities in widely deployed, security-critical systems. Unfortunately, it is too restrictive to merely forbid non-constant format-strings, and clearly the `%n` specifier cannot be eliminated.

Format-string vulnerabilities are one of a wider class of security bugs that can occur in any language. When programmers write security-conscious programs, they should distinguish two different classes of data: untrusted data read from the network should never be passed unchecked to functions requiring trusted data. In our case, untrusted data should never be used directly as a format specifier. We can track the trust level of data in CQUAL by introducing the qualifier *tainted* to mark untrusted data and *untainted* to mark trusted positions. It is safe to interpret untainted data as tainted but not vice-versa, hence we choose *untainted* $<$ *tainted* as our partial order.

As an example use of these qualifiers, consider the following simple program shown in Figure 6.4. This program calls `getenv` to return the value of an environment variable, which is then stored successively in `s` and then `t`, and finally is passed as a format specifier to `printf`. Assuming we do not trust the user's environment variables, this program has a format-string vulnerability. Indeed, on the author's system, setting `LD_LIBRARY_PATH` to a string of eight `%s`'s causes this program to have a segmentation fault.

To detect this format-string vulnerability, we annotate this program as shown in the top half of Figure 5.5 on page 101. Since *tainted* is positive, marking the result of `getenv` with *tainted* produces the constraint $\textit{tainted} \leq \textit{getenv\_ret}'$. Since *untainted* is negative, marking the format-string argument of `printf` as *untainted* produces the constraint $\textit{printf\_arg0}' \leq \textit{untainted}$. Notice that we need not annotate the types of `s` or `t`. When CQUAL performs inference on this program, the generated qualifier constraints are inconsistent, meaning that *tainted* data is passed to an *untainted* argument, i.e., that this program may have a format-string vulnerability. The bottom half of Figure 5.5 displays the set of inconsistent constraints, and as mentioned before the user can explore this error path to discover why type qualifier inference failed.

6.2.1 Experiments

We used CQUAL to check for format-string vulnerabilities in ten popular C programs. For this experiment, we enable flow of qualifiers through casts (Section 5.3) to model taint propagation conservatively. We add *tainted* and *untainted* to programs by supplying a header file that contains declarations of the standard C library functions with the appropriate qualifiers and the appropriate parametric polymorphic types (Section 5.2). We use the same file of annotated library functions for all of our benchmarks. For one benchmark we also annotated two application-specific memory allocation and deallocation functions as polymorphic.

Figure 6.5 lists our benchmarks. All of these programs read data from the network, possibly controlled by a malicious adversary, and hence all could potentially have format-string vulnerabilities. For each program we list the numbers of lines of source code, both before and after preprocessing. The results of applying CQUAL are shown in Figure 6.6. In this figure, the second column lists the analysis time on a 550MHz dual processor Pentium III Xeon with 2GB of memory. The third column lists the memory usage, and the last

Name	Description	Lines	Preproc.
identd-1.0.0	Network id service	223	1224
mingetty-0.9.4	Remote terminal controller	270	1599
bftpd-1.0.11	FTP server	2323	6032
muh-2.05d	IRC proxy	3039	19083
cfengine-1.5.4	Sysadmin tool	26852	141863
imapd-4.7c	UW IMAP4 server	21796	78049
ipopd-4.7c	UW POP3 server	20159	78056
mars_nwe-0.99	Novell Netware emulator	21199	72954
apache-1.3.12	HTTP server	32680	135702
openssh-2.3.0p1[4]	Secure shell	25907	218947

Figure 6.5: Format-String Vulnerability Detection Benchmarks

column lists the number of warnings reported by CQUAL and the number of actual format-string bugs discovered.

For most of these programs CQUAL issues no warning, indicating that the presence of a format-string bug is unlikely. This is especially interesting for two of our test cases, `mars_nwe` and `mingetty`, which contain suspicious looking calls to a function that accepts format-strings [63, 64]. Since we originally studied these programs, the `mars_nwe` code has been patched, and the suspicious looking call has been said to be fully exploitable [46]. Because CQUAL does not model internal compiler functions to read variable arguments,[3] we believe CQUAL may be wrong in this case, though the patch for `mars_nwe` did not give any details about an exploit and stated there were no working exploits yet [47]. The `mingetty` program has also been patched in some distributions, although at least one patch says that the code cannot be abused "to the best of [the writer's] knowledge" [62].

CQUAL finds potential format-string vulnerabilities in three of the programs. Because of the nature of the constraint resolution algorithm, once CQUAL finds an inconsistent constraint it is likely to produce a large number of warnings, as can be seen in the `cfengine` case.

For `muh`, we knew beforehand [58] that these vulnerabilities were present in the code. In the cases of `cfengine` [100] and `bftpd` [7], the vulnerabilities were unknown to us at the time, although we subsequently discovered that these bugs had been previously reported. Nevertheless, this suggests that our tool is effective in finding unknown format-

[3] In `gcc` the important function is `__builtin_next_arg`; in other compilers different techniques, such as pointer arithmetic, are used to access varargs.

[4] We checked for vulnerabilities in the SSH daemon portion of the code.

Name	Time (s)	Mem (kB)	Warn./Bugs
identd-1.0.0	0.087	6444	0/0
mingetty-0.9.4	0.114	7228	0/0
bftpd-1.0.11	0.429	12292	2/1
muh-2.05d	1.093	24252	54/2
cfengine-1.5.4	8.782	132020	>1000/3
imapd-4.7c	7.426	113912	0/0
ipopd-4.7c	7.262	113908	0/0
mars_nwe-0.99	4.445	71112	0/0
apache-1.3.12	8.793	139928	0/0
openssh-2.3.0p1	13.952	221828	0/0

Figure 6.6: Format-String Vulnerability Detection Results

string vulnerabilities.

For `muh` and `bftpd`, fixing the format-string vulnerabilities also eliminates all warnings from CQUAL. For `cfengine`, after fixing the format-string vulnerabilities, we needed to do a little more work to eliminate the remaining warnings. We needed to fix an incorrect call to `sprintf` that was simply a bug. We also needed to make two functions take *const* parameters, declare one function to be polymorphic, and add three typecasts removing tainting from a single character extracted from a *tainted* string

6.2.2 Related Work

Using CQUAL to find format-string vulnerabilities was first described by us [101]. Our approach to finding format-string vulnerabilities is conceptually similar to Perl's taint mode [117], but with a key difference: unlike Perl, which tracks tainting dynamically, CQUAL checks tainting statically without ever running the program. Moreover, CQUAL's results are conservative over all possible runs of the program. This gives us a major advantage over dynamic approaches for finding security flaws. Often security bugs are in the least-tested portions of the code, and a malicious adversary is actively looking for just such code to exploit. Using static analysis, we conceptually analyze all possible runs of the program, providing complete code coverage.

Several lexical techniques have been proposed for finding security vulnerabilities. Pscan [25] searches the source code for calls to `printf`-like functions with a non-constant format string. Thus pscan cannot distinguish between safe calls when the format string is variable and unsafe calls. Lexical techniques have also been proposed to find other security

vulnerabilities [12, 113]. The main advantage of lexical techniques is that they are extremely fast and can analyze non-preprocessed source files. However, because lexical tools have no knowledge of language semantics there are many errors they cannot find, such as those involving aliasing or function calls.

Another approach to eliminating format-string vulnerabilities is to add dynamic checks. The `libformat` library intercepts calls to `printf`-like functions and aborts when a format string contains `%n` and is in a writable address space [94]. A disadvantage to `libformat` is that, to be effective, it must be kept in synchronization with the C libraries. Another dynamic system is FormatGuard, which injects code to dynamically reject bad calls to `printf`-like functions [20]. The main disadvantage of FormatGuard is that programs must be recompiled with FormatGuard to benefit. Another downside to both techniques is that neither protect against denial-of-service attacks.

It is important to realize that while CQUAL is successful at finding format-string vulnerabilities, it can never find all such bugs. One reason is that, as discussed in Section 5.3, CQUAL is sound only up to certain features of C (recall, for example, that while we flow taintedness through casts, this is not sound for casts to integers). However, there is a more fundamental reason that CQUAL can never find "all" security bugs. For example, suppose the programmer performs a branch based on a tainted value. Then conceptually the program counter has become tainted, and any result computed by the rest of the computation is suspect. There is a large body of work on a dual problem to tainting called *secure information flow*, which attempts to prevent just these kinds of security problems [1, 105, 114]. We feel that trying to model all possible information flow can often lead to an overly conservative analysis. For example, the `sendmail` program is a network daemon that waits for data from the network and then performs various tasks depending on the data. If taint propagates to the program counter, then all of `sendmail`'s computation must be tainted, which, while sound, is not a useful result.

6.3 Linux Kernel Locking

Locking is a basic technique for building multi-threaded programs, but it is well-known that locking is easy to get wrong. In this section, we describe experiments using flow-sensitive type qualifiers to prevent a particular problem with locks. We look at simple deadlocks that occur when a thread attempts to acquire the same lock twice in sequence

without an intervening release. Suppose that *lock* and *unlock* are functions that acquire and release locks, respectively. Then we introduce two qualifiers to track the state of locks. We use *locked* to annotate locks that the current thread owns, and we use *unlocked* to annotate locks that may be owned by another thread. Let τ be the type of locks. Using notation from flow-sensitive type qualifier inference (Section 4.5), we assign the following types to the primitive locking functions:

$$lock : (S, ref\ (\rho)) \longrightarrow^{\rho} (Assign(S, \rho{:}\,\textit{locked}\ \tau), void)$$
$$\text{where } S(\rho) \leq \textit{unlocked}\ \tau$$

$$unlock : (S, ref\ (\rho)) \longrightarrow^{\rho} (Assign(S, \rho{:}\,\textit{unlocked}\ \tau), void)$$
$$\text{where } S(\rho) \leq \textit{locked}\ \tau$$

Here we omit uninteresting qualifiers. The function *lock* takes a pointer to a lock as a parameter and requires that the lock have the *unlocked* qualifier in the initial state. The function *lock* changes its parameter to have the *locked* qualifier upon returning. The type of *unlock* is the dual. With these two type signatures we can use our system to statically discover simple deadlocks. Note that although we are describing locking, which is used in multi-threaded code, our system models only a single thread.

There are two natural choices for the partial order on our qualifiers. We can choose the discrete partial order, in which case *locked* and *unlocked* are incomparable. In such a system we signal an error whenever we attempt to join two states in which a lock is inconsistent. Alternatively, we can introduce a third qualifier $\top$ to stand for a lock in an unknown state, with partial order *locked* $< \top$ and *unlocked* $< \top$. In this case we do not signal an error at inconsistent joins, and instead we signal an error if we attempt to acquire or release a lock in the $\top$ state.

6.3.1 Experiments

We used CQUAL to check for simple deadlocks in the Linux kernel device drivers. We can apply the system described above by annotating two primitive locking functions in the Linux kernel:

```
void spin_lock(unlocked spinlock_t *lock) {
    /* inline assembly code */
    change_type(*lock, locked spinlock_t);
}

void spin_unlock(locked spinlock_t *lock) {
    /* inline assembly code */
    change_type(*lock, unlocked spinlock_t);
}
```

Here we insert explicit `change_type` statements because the body of these functions is inline assembly code, which CQUAL cannot analyze. We also add an annotation so that locks are initially *unlocked*.

Because our flow-sensitive type qualifier system is monomorphic, while our annotations are correct they lead to an extremely conservative analysis. Since every lock in the program is passed to `spin_lock` and `_unlock`, given the above definitions our alias analysis determines that all locks may alias a single location ρ. But then our linearity computation determines that ρ is non-linear, and thus ρ cannot be strongly updated, making it unlikely we could type check any realistic kernel code. Thus in practice we replace the above function definitions with macros, effectively inlining the functions.

With the inlined definitions of `spin_lock` and `spin_unlock`, we used CQUAL to check for simple deadlocks in the device drivers in a standard build of the Linux 2.4.9 kernel. Each driver in the Linux kernel is made up of a number of files linked together to form a driver module,[5] which is an object that can be loaded dynamically by the kernel [95]. We performed two experiments on the drivers. In both cases, we do not collapse qualifiers at casts, i.e., we trust the C types.

In the first experiment, we analyzed each of 892 compiled driver source files separately. We chose the discrete partial order for *locked* and *unlocked*, with no $\top$ qualifier; in this model, an attempted join of *locked* and *unlocked* results in a type qualifier error. We make optimistic assumptions about the environment in which each file is invoked. In particular, we assume that the body of each undefined function is empty. As a result our effect inference determines that calls to undefined functions have no effect on the state, and

[5] A few drivers do not come in modular form.

hence the state of all locks flows "around" undefined functions (Section 4.1.2). Finally, we need some mechanism to connect the initial state of locks with the initial state of the appropriate functions. We assume that any function that is either `extern` (has global scope) or has its address taken in a non-calling context is an *interface* function, meaning it may be called directly from the kernel. Let S_G be the initial top-level environment modeling any allocations and initializations of globals in the driver file (Section 5.2). For each interface function f of type $(S, \tau) \longrightarrow^L (S', L')$, we generate the constraints $S_G \leq S$ and $S' \leq S_G$, i.e., each interface function may be called from the top level environment and must leave the state of locks as they found it.

In the second experiment, we analyzed 513 device driver modules, which are made up of the 892 individual files plus some other files linked in from different parts of the kernel. For this experiment we chose the three-point partial order with $\mathit{locked} < \top$ and $\mathit{unlocked} < \top$; thus we allow locks to enter the unknown state, but any such lock cannot be used later. In each case, we simultaneously analyze all files linked together to form a module m and any modules m depends on. Note that this definition of a whole module means that there are many overlapping files between different whole modules. To model the kernel, we construct a `main` function that first invokes the module initialization function, then non-deterministically loops calling each possible driver function, and finally calls the module cleanup function.

We examined the results for all of the 892 separately analyzed driver files and for 64 of the 513 whole modules. In total we found 14 apparently new locking bugs, including one which spans multiple files. In five of the apparent bugs, a particular function is sometimes called with a lock held and sometimes without. For example, the emu10k1 module contains the following deadlock:

```
void emu10k1_mute_irqhandler(struct emu10k1_card *card) {
    struct patch_manager *mgr = &card->mgr;
    ...  spin_lock_irqsave(&mgr->lock, flags);
    emu10k1_set_oss_vol(card, ...); ...
}
void emu10k1_set_oss_vol(struct emu10k1_card *card, ...)  {
    ...  emu10k1_set_volume_gpr(card, ...); ...
}
emu10k1_set_volume_gpr(struct emu10k1_card *card, ...)  {
    struct patch_manager *mgr = &card->mgr;
    ...  spin_lock_irqsave(&mgr->lock, flags); ...
}
```

Here `&mgr->lock` is acquired in the topmost function, which calls the middle functions, which calls the bottom function, which attempts to acquire `&mgr->lock` again. Notice that detecting this error requires interprocedural analysis. The remaining bugs are cases when the programmer forgot to release a lock on a particular path, which causes a deadlock the next time the lock is acquired.

As described by us previously [45], we discussed these 14 bugs with others quite extensively, and all appear to be real problems. The bugs were reported on the Linux kernel mailing list, and as a result at least two were fixed, including the bug shown above (which we reported directly to the author of the module). We suspect that social reasons prevented all of the bugs from being fixed. First, convincing someone to look at 14 somewhat tricky bugs is difficult. Second, if a bug fix is not obvious, developers are reluctant to make changes. For example, in the emu10k1 code shown above, fixing the deadlocks requires deciding why the `set_volume_gpr` function is sometimes called with a lock and sometimes not. Finally, some of the deadlocks could be in code that is already scheduled for removal or a major overhaul, hence fixing such bugs may not be a kernel developer's top priority.

One of our goals is to understand how often, and why, our system fails to type check real programs. We have categorized every type qualifier error in the separate files analysis of the 892 driver files. In this experiment, of the 52 files that fail to type check, 11 files have locking bugs (sometimes more than one) and the remaining 41 files have type errors. Half of these type errors are due to our model of interface functions. Recall that we

generate constraints that require that all interface functions leave locks in the same state. It turns out, however, that for a class of driver files this requirement is simply not met—some of the functions in a file assume that locks are unlocked on entry, and some assume the opposite. These type errors are eliminated by moving to whole module analysis.

The remaining type errors in the individual driver file analysis fall into two main categories. In many cases the problem is that our alias analysis is not strong enough to type check the program. Another common class of type errors arises when locks are conditionally acquired and released. In this case, a lock is acquired if a predicate P is true. Before the lock is released, P is tested again to check whether the lock is held. Our system is not path sensitive, and since for these experiments *locked* and *unlocked* are incomparable, our tool signals a type error at the point where the path on which the lock is acquired joins with the path on which the lock is not acquired. (In the whole module analysis, this error is detected later on, when there is an attempt to acquire or release the a lock in the $\top$ state.) Many of these examples could be rewritten with little effort to pass our type system. In our opinion, this would usually make the code clearer and safer—the duplication of the test on P invites new bugs when the program is modified.

Of the 513 whole modules, 196 contain type errors, many of which are duplicates from shared code. We examined 64 of the type error-containing modules and discovered that a major source of type errors is when there are multiple aliases of a location, but only one alias is actually used in the code of interest. Not surprisingly, larger programs, such as whole modules, have more problems with spurious aliasing than the optimistic single-file analysis. To overcome this limitation *restrict* was added by hand to the 64 modules we looked at, including the emu10k1 module, which yielded the largest number of such false positives. Using *restrict* eliminated all of the type errors that occurred in these modules because non-linear locations could not be strongly updated. This supports our belief that *restrict* is the right tool for dealing with (necessarily) conservative alias analysis. Currently adding *restrict* by hand is burdensome, requiring a relatively large number of annotations. We leave the problem of automatically inferring *restrict* annotations as future work.

Finally, the algorithm for flow-sensitive type qualifier inference described in Section 4.5.1 is carefully designed to limit resource usage. Figure 6.7 shows the running time and Figure 6.8 shows the memory usage of the whole module analysis versus preprocessed lines of code for the 513 whole Linux kernel modules. All experiments were done on a dual processor 550 MHz Pentium III with 2GB of memory.

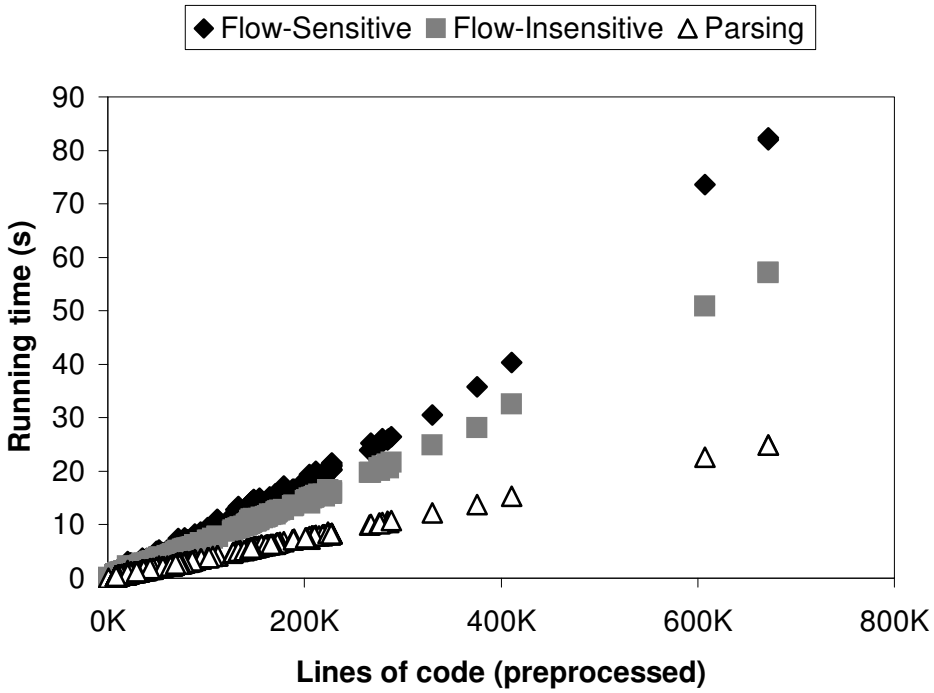

Figure 6.7: Running Time for Whole Module Analysis of Locks

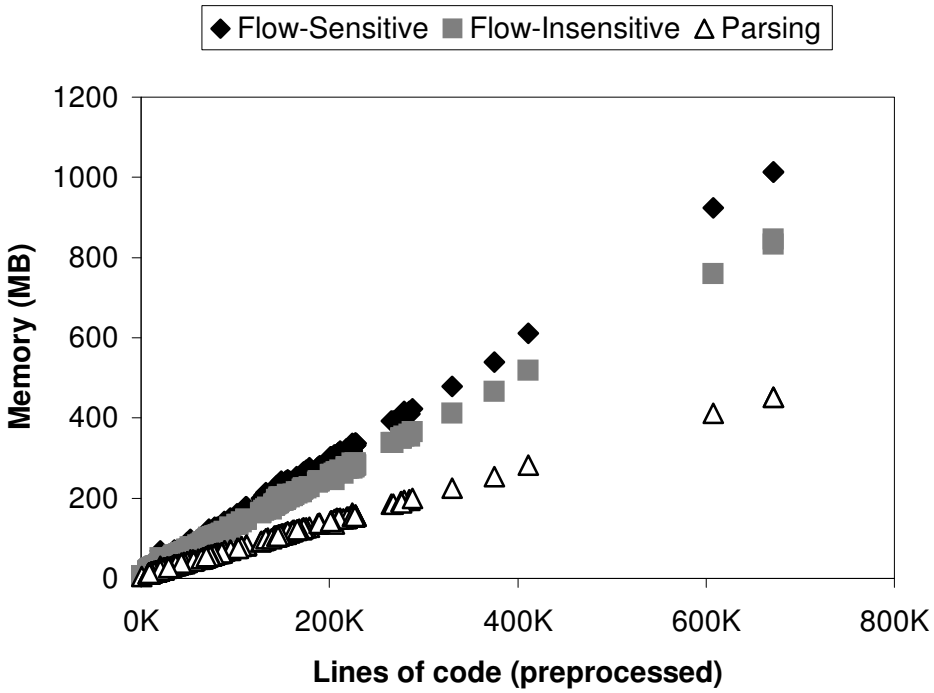

Figure 6.8: Memory Usage for Whole Module Analysis of Locks

We divide the resource usage into three components: C parsing and type checking, flow-insensitive analysis, and flow-sensitive analysis (see Figure 5.1). In the graphs, the reported time and space for each phase includes the time and space for the previous phases. The graphs show that the space overhead of flow-sensitive analysis is relatively small and appears to scale well to large modules. For all modules the space usage for the flow-sensitive analysis is within 35% of the space usage for the flow-insensitive analysis. The running time of the analysis is more variable, but the absolute running times are within a factor of 1.5 of the flow-insensitive running times.

6.3.2 Related Work

The difficulty of using locks correctly is well-known, and a number of techniques have been developed to check lock usage. The Eraser system [99] and Choi et al [18] can check whether locks are correctly acquired and released dynamically at run time. As with any dynamic technique, these tools can find errors only on particular test cases.

Flanagan and Abadi have proposed type systems for checking correct use of locks statically [38, 39]. Flanagan and Freund [40] use a type checking system to verify Java locking behavior. In all three of these type systems, locks are acquired and released according to a lexical discipline. To model locking in the Linux kernel, we must allow non-lexically scoped lock acquires and releases.

Meta-level compilation [33, 54] has been used to find a number of locking bugs in the Linux kernel, including the same kind of simple deadlocks found by our tool. The Linux kernel we analyzed above was a newer version than that checked by Engler et al [33], and many of the deadlocks discovered by meta-level compilation had been removed, so a direct comparison is not possible. Our tool found a bug previously found by meta-level compilation but not yet fixed, and the remaining bugs were found in code that had been changed enough to make a comparison difficult. We believe that, because our system is sound, our approach can find strictly more bugs than meta-level compilation.

6.4 File Operations

Virtually all operating systems include a file system interface. One of the critical features of this interface is that certain functions must be called in a particular order. For example, a file must be opened for reading before it is read, it must be opened for writing

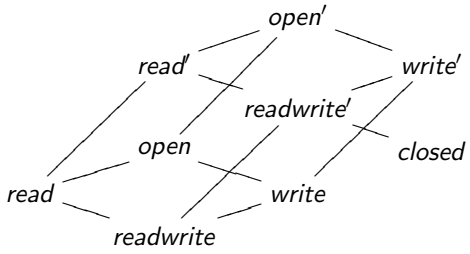

Figure 6.9: Subtyping Relation among C Stream Library Qualifiers

before it is written to, and once closed a file cannot be accessed. We can enforce these rules using flow-sensitive type qualifiers. We introduce five qualifiers to track the state of files. We use qualifiers *open*, *read*, *write*, and *readwrite* to mark files that are open for, respectively, undetermined access, reading, writing, and both reading and writing. We use the qualifier *closed* to mark files that are not open. We also need to account for the fact that opening a file may fail. In particular, in ANSI C the result of a call to the file opening function `fopen` may return `NULL`. Thus we introduce four additional qualifiers *open′*, *read′*, *write′*, and *readwrite′*, to stand for files that the programmer attempted to open but have not yet been checked again `NULL`.

Our nine qualifiers naturally form the partial order shown in Figure 6.9. In this partial order, files open for reading and writing are a subtype of files open only for a single kind of access, and all three are a subtype of files opened for undetermined access. We have a similar partial order for files opened but unchecked, and a closed file can be considered a file open in any state but not yet checked.

We give the expected types to functions that manipulate files. For example, let τ be the type of files. Then the type signature for the function `fclose` to close a file is

$$\texttt{fclose} : (S, \mathit{ref}\ (\rho)) \longrightarrow^{\rho} (\mathit{Assign}(S, \rho\colon \mathsf{closed}\ \tau), \mathit{int}) \quad \text{where } S(\rho) \leq \mathsf{open}\ \tau$$

Here, as below, we omit uninteresting qualifiers. The function `fclose` takes a pointer to a file that must be open, and when the function returns the file is closed. Similarly, the type signature for `fopen` is

$$\texttt{fopen} : (S, \cdots \times \mathit{mode}) \longrightarrow^{\rho} (\mathit{Assign}(\mathit{Alloc}(S, \rho), \rho\colon \mathit{mode}\ \tau), \mathit{ref}\ (\rho)) \quad \text{where } S(\rho) \leq \mathsf{closed}\ \tau$$

We require that initially the file not be open, and upon returning we return a pointer to a new file whose qualifier is specified by *mode*. In practice the mode is usually a constant string, and therefore we can determine the correct qualifier, *read′*, *write′*, or *readwrite′*, by a simple syntactic comparison. If we cannot determine the mode qualifier syntactically, we issue a warning and use the *open′* qualifier.

We assign similar types to functions that read and write files. CQUAL contains code to handle comparisons of files against `NULL`, the special value that indicates an unsuccessfully opened file. For example, consider the following C code:

```
if ((file = fopen(filename, "r")) != NULL) {
    ...  fgetc(file); ...
    fclose(file);
} else {
    printf("Failed to open %s", filename);
}
```

At the call to `fopen`, we syntactically recognize the string `"r"` and determine that the file is being opened for reading. Let ρ be the abstract location for the file. In the state S immediately following the call to `fopen`, we assign ρ the type *read′* τ, where τ is the type of files. Next we see a comparison again `NULL`. Intuitively, this is a kind of type case. For this example we analyze the true branch with initial store $Assign(S, \rho : \mathit{read}\ \tau)$ and the false branch with initial store $Assign(S, \rho : \mathit{closed}\ \tau)$. We use conditional constraints to relate the *read′* qualifier in S to *read* on the true branch.

Note that unlike Das [23], we do not model arbitrary predicates to keep track of the state of files. For the examples we looked at, modeling file pointer comparisons as described above was usually sufficient—for our examples, typically a file pointer would first be tested against `NULL` and then used. In contrast, in Das's examples instead of checking a file pointer directly for `NULL` before accessing a file, often a separate, global boolean flag would be tested.

The class of file operation usage errors we can detect with this technique includes files used without having been opened and checked against `NULL`, files accessed in a mode incompatible with how they were opened, and files accessed after being closed. ANSI C specifies that files open for both reading and writing switch into read-only (write-only) mode after the first read (write) until a call to certain functions or until end-of-file is encountered

[6]; we did not model this rule in our experiments.

Finally, because our system is monomorphic, as in Section 6.3 if we simply assigned the above types to the C file operations we would not be able to check many programs. Hence we again replace library function definitions with macro expansions, effectively inlining the function calls. In practice this is achieved by taking a suitably modified `stdio.h` file and dropping it in the directory containing the application.

6.4.1 Experiments

We applied CQUAL to check file usage in two application programs, `man-1.5h1` and `sendmail-8.11.6`. We were primarily interested in the performance of our tool on a more complex application, as we did not expect to find any latent file operation usage bugs in such mature programs. However, we did find one minor bug in `sendmail-8.11.6`, in which an opened log file is never closed in some circumstances. On a 550 MHz dual processor Pentium III Xeon with 2GB of memory, the analysis of `sendmail-8.11.6`, with 175,193 preprocessed source lines, took 22.853 seconds and 273MB; `man-1.5h1`, with 16,411 preprocessed source lines, took 1.729 seconds and 36MB. These results suggest that our algorithm also behaves efficiently when checking C stream library usage.

Chapter 7

Conclusion

In this dissertation, we have presented type qualifiers, a lightweight, specification-based technique for improving software quality. To use our system, the programmer supplies three components: a set of qualifiers, a partial order among the qualifiers, and a source program with a few key type qualifier annotations. The partial order on type qualifiers is extended to a subtyping relation among qualified types. Constraint-based type qualifier inference takes as input the source program, determines the remaining qualifiers, and checks for consistency. Any inconsistent qualifiers indicate potential bugs in the program.

The basic type qualifier system is flow-insensitive, meaning that a variable's qualifiers are fixed throughout execution, just as standard types are. An important feature of our system is that type qualifiers may also be flow-sensitive, i.e., the type qualifiers associated with a particular memory location may change from one point to the next. We presented a lazy, constraint-based flow-sensitive type qualifier inference algorithm that runs in two phases. In the first phase, we perform flow-insensitive alias analysis and effect inference. In the second phase, we use the resulting set of abstract locations and effects to infer flow-sensitive linearities, which we use to distinguish strong and weak updates, and flow-sensitive type qualifiers. A linear location supports strong updates, and a non-linear locations supports only weak updates. We also use effects to gain a measure of polymorphism by propagating the state of any locations not used by a function "around" rather than "through" the function call. By combining this technique with a rule for effect hiding, we can precisely track states of purely local variables even in the presence of recursive functions.

We have introduced a new language construct $\texttt{restrict}\, x = e_1\, \texttt{in}\, e_2$ that may

be of independent interest. The `restrict` construct allows programmers to express their intentions about aliasing. In this construct, the name x, bound in e_2, is initialized to e_1, which must be a pointer. Suppose x and e_1 point to object o. Then within e_2, only the name x and copies of x may be used to access o. Dually, outside the scope of e_2 the name x and values copied from x may *not* be used to access o (they may not escape). This information often enables a flow-sensitive analysis to track the state of o precisely within e_2. In our system, programmers add `restrict` to their programs whenever flow-sensitive type qualifier inference fails because a non-linear location cannot be strongly updated. Internally this works because only x may be used within e_2 and only e_1 may be used outside of e_2; thus x can be bound to a different abstract location than e_1, and x may be linear even if e_1 is non-linear.

We have described a tool CQUAL that implements both flow-insensitive and flow-sensitive type qualifier inference for C. An important component of our tool is a user interface for presenting analysis results. In this interface, the source code is colored according to the inferred qualifiers. Additionally, the user can click on any qualifier variable to see a set of constraints showing how the solution for that qualifier variable was inferred. Clicking on the constraints jumps to the source line where the constraint was generated, and this allows a programmer to trace through the source code to find the cause of an error.

Finally, we presented a number of experiments using CQUAL that suggest that type qualifiers are useful in practice. We presented four separate studies. In the first, we performed *const* inference for C programs, and we discovered that inference can add many more *const*s to existing programs, even ones whose authors make a significant effort to use *const*. In the second, we used CQUAL to find a number of format-string bugs in popular programs; several of these bugs were unknown to us at the time. In the third, we found new deadlocks in the Linux kernel using CQUAL's flow-sensitive type qualifier inference, including one that spanned multiple files. In the last experiment, we used CQUAL to check the use of file operations in two C programs.

In conclusion, we believe we have shown that type qualifiers are lightweight and easy to use because they are natural extensions of type systems and because of constraint visualization; that type qualifiers are practical, because of efficient inference algorithms that scale to large programs; and that type qualifiers are useful for a number of realistic applications.

Appendix A

Soundness of Flow-Insensitive Type Qualifiers

In this appendix we present a complete proof of soundness for the flow-insensitive type qualifier checking system of Figure 3.5. In the main exposition of this dissertation, we present the semantics of type qualifiers using a big-step semantics (Figure 3.9). We can prove soundness of our type qualifier checking system under these semantics using the same technique we use in Appendix B to prove soundness of `restrict`. However, rather than present a nearly similar proof twice, for variety in this appendix we instead prove that our type qualifier checking system is sound with respect to a small-step operational semantics that is equivalent to our big-step semantics for terminating programs. Our soundness proof uses techniques from Wright and Felleisen [122] and Eifrig et al [31].

A.1 Small-Step Semantics

In a big-step semantics, a reduction step runs a computation to completion. In a small-step semantics, we instead perform only a single "unit" of computation at a time, and each reduction may produce an intermediate state rather than a final result. In our small-step semantics, values are of the form (v, Q), where v is either a location, an integer, or a syntactic function. We begin by defining a reduction context R, which is an expression

$$
\begin{array}{rcll}
\langle S, R[\texttt{annot}(n, Q)]\rangle & \rightarrow & \langle S, R[(n, Q)]\rangle & \\
\langle S, R[\texttt{annot}(\lambda x.e, Q)]\rangle & \rightarrow & \langle S, R[(\lambda x.e, Q)]\rangle & \\
\langle S, R[\texttt{annot}(\texttt{ref}\ (v, Q), Q')]\rangle & \rightarrow & \langle S[l \mapsto (v, Q)], R[(l, Q')]\rangle & l \notin dom(S) \\
\langle S, R[(\lambda x.e, Q')\ (v, Q)]\rangle & \rightarrow & \langle S, R[e[x \mapsto (v, Q)]]\rangle & \\
\langle S, R[\texttt{let}\ x = (v, Q)\ \texttt{in}\ e]\rangle & \rightarrow & \langle S, R[e[x \mapsto (v, Q)]]\rangle & \\
\langle S, R[\texttt{*}(l, Q)]\rangle & \rightarrow & \langle S, R[S(l)]\rangle & l \in dom(S) \\
\langle S, R[(l, Q')\ \texttt{:=}\ (v, Q)]\rangle & \rightarrow & \langle S[l \mapsto (v, Q)], R[(v, Q)]\rangle & l \in dom(S) \\
\langle S, R[\texttt{check}((v, Q'), Q)]\rangle & \rightarrow & \langle S, R[(v, Q')]\rangle & Q' \leq Q
\end{array}
$$

Figure A.1: Small-Step Operational Semantics with Qualifiers

containing a hole []:

$$
\begin{array}{rcl}
R & ::= & [] \mid R\ e \mid (v, Q)\ R \mid \texttt{let}\ x = R\ \texttt{in}\ e \mid \texttt{annot}(\texttt{ref}\ R, Q) \mid \texttt{*}R \\
& \mid & R\ \texttt{:=}\ e \mid (v, Q)\ \texttt{:=}\ R \mid \texttt{check}(R, Q)
\end{array}
$$

We write $R[e]$ to mean reduction context R with R's hole replaced by e. A reduction context fixes the left-to-right ordering of evaluation and tells us, for a given expression, where the "next" reduction must occur. For example, by the above grammar we see that we may only evaluate the right-hand side of an assignment if the left-hand side has already been evaluated to a value.

Figure A.1 presents our small-step operational semantics. We define a configuration $\langle S, e\rangle$ to be a pair where e is the expression that is being evaluated and S is the current *store*, a mapping from locations to values just as in the big-step semantics. Our small-step semantics defines a reduction relation $\langle S, e\rangle \rightarrow \langle S', e'\rangle$ that shows how the configuration changes with a single step of execution. We write $\rightarrow^*$ for the reflexive, transitive closure of $\rightarrow$. Notice that, just as in our big-step semantics, we discard the outermost qualifier on a value except when we encounter a qualifier check. Note that in these semantics, as in the rest of this section, we ignore the standard type annotation on function definitions $\lambda x.e$; they are unnecessary for this proof, and they simply add clutter.

Definition A.1 *A configuration $\langle S, e\rangle$ is* stuck *if no reductions in Figure A.1 are applicable.*

In the next section, we prove that well-typed programs do not get stuck.

The connection between the small-step semantics of Figure A.1 and the big-step semantics of Figure 3.9 should be clear. We formally state the correspondence with the

following lemma.

Lemma A.2 *If* $S \vdash_q e \to (v, Q); S'$*, then* $\langle S, e\rangle \to^* \langle S', (v, Q)\rangle$*. If* $S \vdash_q e \to \textbf{\textit{err}}$*, then* $\langle S, e\rangle \to^* \langle S', e'\rangle$ *where* $\langle S', e'\rangle$ *is stuck.*

A.2 Soundness

Recall that values in our semantics are of the form (v, Q). We use the following rule to assign types to such values:

$$\frac{\Gamma \vdash_q v : \sigma}{\Gamma \vdash_q (v, Q) : Q\ \sigma}\ (\text{Value}_q)$$

Recall that our type system in Figure 3.5 assigns integers and functions types σ without top-level qualifiers; hence the above rule. In our proof, variables are used for two purposes. Semantic locations l are represented as free variables, and their unqualified types σ are stored in Γ. Functions in our semantics are represented not as closures but as syntactic functions, which is a standard technique [31, 122]. Thus evaluated functions are type checked using (Lam_q), in which case type environments Γ bind program variables (i.e., function parameters) to qualified types τ. Therefore Γ may map program variables to both qualified and unqualified types—qualified types for program variables, unqualified types for locations.

As evaluation progresses, we extend a type environment with the types of new locations.

Definition A.3 (Extension) *We say that* Γ' *is an* extension *of* Γ*, written* $\Gamma \Rightarrow \Gamma'$*, if* $\Gamma'|_{dom(\Gamma)} = \Gamma$.

Here $\Gamma'|_{dom(\Gamma)}$ is the restriction of Γ' to the domain of Γ. We define a compatibility relation that says when a type environment gives a valid type to a store.

Definition A.4 (Compatibility) *We say that* Γ *is compatible with store* S*, written* $\Gamma \sim S$*, if* $dom(\Gamma) = dom(S)$ *and for all* $l \in dom(S)$ *there exists a* τ *such that* $\Gamma(l) = ref\ (\tau)$ *and* $\Gamma \vdash S(l) : \tau$.

In other words, $\Gamma \sim S$ means that type environment Γ assigns each location l in S a type compatible with the value stored at l. Notice that, as mentioned above, $\Gamma(l)$ has no top-level qualifier.

We need several lemmas to show our main result. We use the first lemma without comment during our proof.

Lemma A.5 *A proof* $\Gamma \vdash_q e : \tau$ *can be rewritten to use at most one application of* (Sub_q) *in sequence.*

Proof: By transitivity of $\leq$. □

Lemma A.6 *If* $\Gamma \vdash_q e : \tau$ *and* $\Gamma \Rightarrow \Gamma'$, *then* $\Gamma' \vdash_q e : \tau$.

Lemma A.7 (Substitution) *If* $\Gamma[x \mapsto \tau] \vdash_q e : \tau'$ *and* $\Gamma \vdash_q (v, Q) : \tau$, *then* $\Gamma \vdash_q e[x \mapsto (v, Q)] : \tau'$.

Note that for the substitution lemma to hold we implicitly assume that we rename any bound variables in e to avoid capturing any free variables (i.e., locations) in v, as is standard.

Theorem A.8 (Subject Reduction) *If* $\Gamma \vdash_q e : \tau$ *and* $\Gamma \sim S$, *then either* e *is a value* (v, Q), *or there exist* S', e', *and* Γ' *such that*

1. $\langle S, e\rangle \rightarrow \langle S', e'\rangle$,
2. $\Gamma' \vdash_q e' : \tau$,
3. $\Gamma' \sim S'$, *and*
4. $\Gamma \Rightarrow \Gamma'$.

Proof: By induction on the structure of e.

Case $n, \lambda x.e, \texttt{ref}\, e$

There are no typing rules that assign qualified types to unannotated integers, functions, or references, so this cannot occur.

Case x

This case cannot happen. By $\Gamma \sim S$, we have $\Gamma(x) = \mathit{ref}\ (\tau')$, and $\mathit{ref}\ (\tau')$ is not a qualified type τ.

Case $\texttt{annot}(n, Q)$

By assumption we have $\Gamma \sim S$, and our typing proof must be of the form

$$\frac{\Gamma \vdash_q \texttt{annot}(n, Q) : Q\ \ \mathit{int} \qquad Q \leq Q'}{\Gamma \vdash_q \texttt{annot}(n, Q) : Q'\ \ \mathit{int}}$$

Then by examination of the reduction rules in Figure A.1, we can apply the reduction $\langle S, \texttt{annot}(n, Q)\rangle \to \langle S, (n, Q)\rangle$. Let $S' = S$ and $\Gamma' = \Gamma$. Then we can show $\Gamma' \vdash_q (n, Q) : \mathit{int}\, Q$, and by ($\text{Sub}_q$), since $Q \leq Q'$, we can show $\Gamma' \vdash_q (n, Q) : \mathit{int}\, Q'$.

Case $\texttt{annot}(\lambda x.e, Q)$

Similar to the case for annotated integers.

Case $e_1\ e_2$

By induction we have a typing proof

$$\dfrac{\dfrac{\Gamma \vdash_q e_1 : Q''\ (\tau'' \longrightarrow \tau') \qquad \Gamma \vdash_q e_2 : \tau''}{\Gamma \vdash_q e_1\ e_2 : \tau'} \qquad \tau' \leq \tau}{\Gamma \vdash_q e_1\ e_2 : \tau} \tag{A.1}$$

There are three sub-cases. If e_1 is not a value, then by induction there exist $\Gamma_{e'}$, $S_{e'}$, and e_1' such that $\langle S, e_1\rangle \to \langle S', e_1'\rangle$, $\Gamma \Rightarrow \Gamma_{e'}$, $\Gamma_{e'} \sim S'$, and

$$\Gamma_{e'} \vdash_q e_1' : Q\ (\tau'' \longrightarrow \tau') \tag{A.2}$$

Thus $\langle S, e_1\ e_2\rangle \to \langle S', e_1'\ e_2\rangle$. Picking $\Gamma' = \Gamma_{e'}$, we already know Γ' is an extension of Γ and is compatible with S'. By Lemma A.6 we can show $\Gamma' \vdash_q e_2 : \tau''$. Then since $\tau' \leq \tau$, combining this with (A.2) we have $\Gamma' \vdash_q e_1'\ e_2 : \tau$, which proves our conclusion.

If e_2 is not a value, then by similar reasoning we can show our conclusion. So the last case we need to check is when both e_1 and e_2 are values. Examination of the type rules in Figure 3.5 shows that the only rules that can assign a function type to e_1 are (Lam_q) (followed by (Value_q)) and (Var_q). But since locations are all assigned *ref* types by Γ, we know that only (Lam_q) could have been applied. Thus our typing proof must be of the form

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma[x \mapsto \tau_x] \vdash_q e : \tau_e}{\Gamma \vdash_q \lambda x.e : \tau_x \longrightarrow \tau_e}}{\Gamma \vdash_q (\lambda x.e, Q') : Q'\ (\tau_x \longrightarrow \tau_e)}}{Q' \leq Q'' \qquad \tau'' \leq \tau_x \qquad \tau_e \leq \tau'}}{\Gamma \vdash_q (\lambda x.e, Q') : Q''\ (\tau'' \longrightarrow \tau')} \qquad \Gamma \vdash_q (v, Q) : \tau''}{\Gamma \vdash_q (\lambda x.e, Q')\ (v, Q) : \tau'} \qquad \tau' \leq \tau}{\Gamma \vdash_q (\lambda x.e, Q')\ (v, Q) : \tau} \tag{A.3}$$

Therefore $e_1\ e_2$ must be of the form $(\lambda x.e, Q')\ (v, Q)$, and hence we can apply the reduction $\langle S, (\lambda x.e, Q')\ (v, Q)\rangle \to \langle S, e[x \mapsto (v, Q)]\rangle$.

Then since $\Gamma \vdash_q (v, Q) : \tau''$ and $\tau'' \leq \tau_x$, by (Sub_q) we have $\Gamma \vdash_q (v, Q) : \tau_x$. Then by the topmost hypothesis of (A.3) and by Lemma A.7 we have $\Gamma \vdash_q e[x \mapsto (v, Q)] : \tau_e$.

And since $\tau_e \leq \tau' \leq \tau$, by ($\text{Sub}_q$) we have $\Gamma \vdash_q e[x \mapsto (v, Q)] : \tau$. Then letting $\Gamma' = \Gamma$, we clearly have $\Gamma \Rightarrow \Gamma'$, $\Gamma \sim S$ (by assumption), and $\Gamma' \vdash_q e[x \mapsto (v, Q)] : \tau$, which proves our conclusion.

Case $\texttt{let}\, x = e_1\, \texttt{in}\, e_2$

In a monomorphic type system $\texttt{let}\, x = e_1\, \texttt{in}\, e_2$ is equivalent to $(\lambda x.e_2)\ e_1$, so we can simply apply the proof steps for application and abstraction.

Case $\texttt{annot}(\texttt{ref}\, e, Q')$

If e is not a value, then by reasoning similar to the first sub-case for $e_1\ e_2$ we can prove our conclusion. Otherwise, suppose e is a value (v, Q). Then examination of the reduction rules in Figure A.1 shows that we can apply the reduction $\langle S, \texttt{annot}(\texttt{ref}\, (v, Q), Q')\rangle \rightarrow \langle S', (l, Q')\rangle$ where $S' = S[l \mapsto (v, Q)]$ and $l \notin dom(S)$. Our typing judgment must be of the form

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash_q (v, Q) : \tau}{\Gamma \vdash_q \texttt{ref}\, (v, Q) : ref\ (\tau)}}{\Gamma \vdash_q \texttt{annot}(\texttt{ref}\, (v, Q), Q') : Q'\ \ ref\ (\tau)} \quad Q' \leq Q''}{\Gamma \vdash_q \texttt{annot}(\texttt{ref}\, (v, Q), Q') : Q''\ \ ref\ (\tau)} \tag{A.4}$$

Let $\Gamma' = \Gamma[l \mapsto ref\ (\tau)]$. Then clearly $\Gamma' \vdash_q (l, Q') : Q''\ \ ref\ (\tau)$ since $Q' \leq Q''$. Since $l \notin dom(S)$ and $\Gamma \sim S$ by assumption, we know $l \notin dom(\Gamma)$ and therefore $\Gamma \Rightarrow \Gamma'$ and $dom(\Gamma') = dom(S')$ Finally, to see $\Gamma' \sim S'$, pick any location $l' \in dom(S')$. If $l' \neq l$, then by construction of Γ' and S' and since $\Gamma \sim S$, clearly there exists a τ' such that $\Gamma'(l') = ref\ (\tau')$ and $\Gamma' \vdash_q S'(l') : \tau'$ since $\Gamma \Rightarrow \Gamma'$. If $l' = l$, then $\Gamma'(l) = ref\ (\tau)$, and from (A.4) and $\Gamma \Rightarrow \Gamma'$ we know $\Gamma' \vdash_q S'(l) : \tau$, since $S'(l) = (v, Q)$. Thus our conclusion holds.

Case $*e$

If e is not a value, then by reasoning similar to the first sub-case for $e_1\ e_2$ we can prove our conclusion. Otherwise, suppose e is a value. Then we must assign e a *ref* type, and examination of the typing rules in Figure 3.5 reveals that only (Var_q) (followed by (Value_q)) can assign such a type to a value. Thus our typing proof must be of the form

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash_q l : ref\ (\tau')}{\Gamma \vdash_q (l, Q) : Q\ \ ref\ (\tau')} \quad Q \leq Q'}{\Gamma \vdash_q (l, Q) : Q'\ \ ref\ (\tau')}}{\Gamma \vdash_q *(l, Q) : \tau'} \quad \tau' \leq \tau}{\Gamma \vdash_q *(l, Q) : \tau} \tag{A.5}$$

Since $l \in dom(\Gamma)$ by (A.5) and $\Gamma \sim S$, we know $l \in dom(S)$. Therefore we can perform a

reduction $\langle S, *(l, Q)\rangle \rightarrow \langle S, S(l)\rangle$. Let $\Gamma' = \Gamma$. Then clearly $\Gamma \Rightarrow \Gamma'$, and by assumption $\Gamma' \sim S$. By the latter and (A.5), we know $\Gamma' \vdash_q S(l) : \tau'$. Then since $\tau' \leq \tau$, by (Sub_q) we have $\Gamma' \vdash_q S(l) : \tau$. Thus our conclusion holds.

Case $e_1 := e_2$

If either e_1 or e_2 is not a value, then by reasoning similar to the first sub-case for $e_1\ e_2$ we can prove our conclusion. Otherwise, suppose both are values. Then we must assign e a *ref* type, and examination of the typing rules in Figure 3.5 reveals that only (Var_q) (followed by (Value_q)) can assign such a type to a value. Thus our typing proof must be of the form

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash_q l : \mathit{ref}\ (\tau')}{\Gamma \vdash_q (l, Q') : Q'\ \mathit{ref}\ (\tau')} \quad Q' \leq Q''}{\Gamma \vdash_q (l, Q') : Q''\ \mathit{ref}\ (\tau')} \quad \Gamma \vdash_q (v, Q) : \tau'}{\Gamma \vdash_q (l, Q') := (v, Q) : \tau'} \quad \tau' \leq \tau}{\Gamma \vdash_q (l, Q') := (v, Q) : \tau} \tag{A.6}$$

Since $l \in \mathit{dom}(\Gamma)$ by (A.6) and since $\Gamma \sim S$, we know $l \in \mathit{dom}(S)$. Therefore we can perform a reduction $\langle S, (l, Q') := (v, Q)\rangle \rightarrow \langle S', (v, Q)\rangle$ where $S' = S[l \mapsto (v, Q)]$. Let $\Gamma' = \Gamma$. Then clearly $\Gamma \Rightarrow \Gamma'$, and from (A.6) we see $\Gamma' \vdash_q (v, Q) : \tau$ since $\tau' \leq \tau$. To see $\Gamma' \sim S'$, pick any $l' \in \mathit{dom}(S')$. If $l' \neq l$ then by $\Gamma \sim S$ and $\Gamma \Rightarrow \Gamma'$ there exists a $\tau_{l'}$ such that $\Gamma'(l') = \mathit{ref}\ (\tau_{l'})$ and $\Gamma' \vdash_q S'(l') : \tau_{l'}$. If $l' = l$, then $\Gamma'(l') = \mathit{ref}\ (\tau')$ by (A.6), and $S'(l') = (v, Q)$. But also by (A.6) and $\Gamma \Rightarrow \Gamma'$ we have $\Gamma' \vdash_q (v, Q) : \tau'$. Thus our conclusion holds.

Case $\texttt{check}(e, Q)$

If e is not a value, then by reasoning similar to the first sub-case for $e_1\ e_2$ our conclusion holds. Otherwise suppose e is a value (v, Q'). Then our typing proof must be of the form

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash_q v : \sigma}{\Gamma \vdash_q (v, Q') : Q'\ \sigma} \quad Q' \leq Q''}{\Gamma \vdash_q (v, Q') : Q''\ \sigma} \quad Q'' \leq Q}{\Gamma \vdash_q \texttt{check}((v, Q'), Q) : Q''\ \sigma} \quad Q''\ \sigma \leq \tau}{\Gamma \vdash_q \texttt{check}((v, Q'), Q) : \tau} \tag{A.7}$$

Then since $Q' \leq Q'' \leq Q$, we can apply a reduction $\langle S, \texttt{check}((v, Q'), Q)\rangle \rightarrow \langle S, (v, Q')\rangle$. Let $\Gamma' = \Gamma$. Then clearly $\Gamma \Rightarrow \Gamma'$, and $\Gamma' \sim S$ by assumption. Finally, from (A.7) we have $\Gamma' \vdash_q (v, Q') : \tau$, since $Q''\ \sigma \leq \tau$. Thus our conclusion holds. □

Theorem A.9 *If* $\emptyset \vdash_q e : \tau$*, then either* $\langle \emptyset, e\rangle$ *diverges or* $\langle \emptyset, e\rangle \rightarrow^* \langle S, (v, Q)\rangle$.

Proof: Suppose $\emptyset \vdash_q e : \tau$. We can trivially show $\emptyset \sim \emptyset$. Therefore by subject reduction we can reduce $\langle \emptyset, e \rangle$ either indefinitely or until it reduces to a value. □

Corollary A.10 *If* $\emptyset \vdash_q e : \tau$ *and* $\emptyset \vdash_q e \rightarrow r; S'$*, then* r *is not* ***err****.*

Proof: By Lemma A.2. □

Appendix B

Soundness of Restrict

In this appendix we give the complete proof of soundness for `restrict`; we sketched this proof in Section 4.3.2. For reference, Figure B.1 combines Figures 2.2 and 4.6 to give the complete semantics of our source language with `restrict`. As before, we implicitly assume we have error reductions when the rules in Figure B.1 cannot be applied. For the sake of completeness, Figure B.2 gives the new error rules; recall that $S \vdash e \to \mathbf{err}$ is shorthand for $S \vdash e \to \mathbf{err}; S'$ for arbitrary S'.

We give a proof of soundness of the rules of Figure 4.4 with respect to the semantics of Figure B.1. We allow subsumption of effects, as discussed briefly in Section 4.3.4. Our proof is deliberately designed to resemble the proof of Appendix A, with somewhat more complicated details due to the semantics of `restrict`. This appendix and Appendix A may be read independently.

In our proof we have no need for the translation of expressions. Thus we abbreviate the judgment $\Gamma \vdash e \Rightarrow e' : t; L$ by $\Gamma \vdash e : t; L$. We ignore qualifier annotations and assertions, since they do not affect the correctness of `restrict`. Locations l are represented in the proof as free variables, and thus their types are stored in Γ and they type check using (Var_a). We implicitly treat evaluated and unevaluated integers identically and use (Int_a) to type check both. Functions are represented not as closures but as syntactic functions, as in standard small-step semantics subject-reduction proofs [31, 122]. Thus evaluated functions are type checked using (Lam_a).

To show soundness we first show a subject-reduction result. We begin by introducing a notion of compatibility to capture when it is safe to evaluate an expression.

$$\frac{l \in dom(S)}{S \vdash l \to l; S} \text{ [Var]}$$

$$\frac{}{S \vdash n \to n; S} \text{ [Int]}$$

$$\frac{}{S \vdash \lambda x.e \to \lambda x.e; S} \text{ [Lam]}$$

$$\frac{S \vdash e_1 \to \lambda x.e; S' \quad S' \vdash e_2 \to v; S'' \quad S'' \vdash e[x \mapsto v] \to v'; S'''}{S \vdash e_1\ e_2 \to v'; S'''} \text{ [App]}$$

$$\frac{S \vdash e_1 \to v_1; S' \quad S' \vdash e_2[x \mapsto v_1] \to v_2; S''}{S \vdash \texttt{let}\, x = e_1\, \texttt{in}\, e_2 \to v_2; S''} \text{ [Let]}$$

$$\frac{S \vdash e \to v; S' \quad l \notin dom(S')}{S \vdash \texttt{ref}\, e \to l; S'[l \mapsto v]} \text{ [Ref]}$$

$$\frac{S \vdash e \to l; S' \quad l \in dom(S')}{S \vdash *e \to S'(l); S'} \text{ [Deref]}$$

$$\frac{S \vdash e_1 \to l; S' \quad S' \vdash e_2 \to v; S'' \quad l \in dom(S'') \quad S''(l) \neq \mathbf{err}}{S \vdash e_1 := e_2 \to v; S''[l \mapsto v]} \text{ [Assign]}$$

$$\frac{\begin{array}{c} S \vdash e_1 \to l; S' \quad S'[l \mapsto \mathbf{err}, l' \mapsto S'(l)] \vdash e_2[x \mapsto l'] \to v, S'' \\ l \in dom(S') \qquad\qquad l' \notin dom(S') \end{array}}{S \vdash \texttt{restrict}\, x = e_1\, \texttt{in}\, e_2 \to v; S''[l \mapsto S''(l'), l' \mapsto \mathbf{err}]} \text{ [Restrict]}$$

Figure B.1: Complete Big-Step Operational Semantics for Restrict

$$\frac{S \vdash e_1 \to l; S' \quad S' \vdash e_2 \to v; S'' \quad l \in dom(S'') \quad S''(l) = \mathbf{err}}{S \vdash e_1 := e_2 \to \mathbf{err}} \text{ [Assign]}$$

$$\frac{S \vdash e_1 \to r; S' \quad r \text{ is not of the form } l}{S \vdash \texttt{restrict}\, x = e_1\, \texttt{in}\, e_2 \to \mathbf{err}} \text{ [Restrict]}$$

$$\frac{S \vdash e_1 \to l; S' \quad l \notin dom(S')}{S \vdash \texttt{restrict}\, x = e_1\, \texttt{in}\, e_2 \to \mathbf{err}} \text{ [Restrict]}$$

$$\frac{\begin{array}{c} S \vdash e_1 \to l; S' \quad S'[l \mapsto \mathbf{err}, l' \mapsto S'(l)] \vdash e_2[x \mapsto l'] \to \mathbf{err} \\ l \in dom(S') \qquad\qquad l' \notin dom(S') \end{array}}{S \vdash \texttt{restrict}\, x = e_1\, \texttt{in}\, e_2 \to \mathbf{err}} \text{ [Restrict]}$$

Figure B.2: Error Rules for Restrict

Definition B.1 (Compatibility) *We say* Γ *and* L *are* compatible *with store* S*, written* $(\Gamma, L) \sim S$*, if*

1. $dom(\Gamma) = dom(S)$ *and*

2. *for all* $l \in dom(S)$*, there exists* ρ *such that* $\Gamma(l) = ref\ (\rho)$ *and*

$$\begin{cases} \Gamma \vdash S(l) : C_I(\rho); \emptyset & \text{if } S(l) \neq \textbf{\textit{err}} \\ \rho \notin L & \text{if } S(l) = \textbf{\textit{err}} \end{cases}$$

Intuitively, $(\Gamma, L) \sim S$ means an expression e that type checks in environment Γ and has effect L can execute safely in store S. Notice that the definition of compatibility requires $dom(\Gamma) = dom(S)$, i.e., that expressions typed in environment Γ contain locations but not other free variables. This property is maintained during evaluation because in [App] we implement function calls with substitution.

Lemma B.2 *If* $(\Gamma, L \cup L') \sim S$ *then* $(\Gamma, L') \sim S$*.*

As evaluation progresses in our proof we extend Γ with new locations allocated by `ref` expressions.

Definition B.3 (Extension) *We say that* (Γ', S') *is an* extension *of* (Γ, S) *if*

1. $dom(\Gamma) = dom(S)$ *and* $dom(\Gamma') = dom(S')$ *and*

2. $\Gamma'|_{dom(\Gamma)} = \Gamma$

Here $\Gamma'|_{dom(\Gamma)}$ is the restriction of Γ' to the domain of Γ. It is a property of our semantics and type system that these extensions are safe, in the following sense:

Definition B.4 (Safe Extension) *We say that* (Γ', S') *is a* safe extension *of* (Γ, S)*, written* $(\Gamma, S) \Rightarrow (\Gamma', S')$*, if*

1. (Γ', S') *is an extension of* (Γ, S)*,*

2. *for all* $l \in dom(S') - dom(S)$*, if* $S'(l) = \textbf{\textit{err}}$ *and* $\Gamma'(l) = ref\ (\rho)$*, then* $\rho \notin loceff(\Gamma)$*, and*

3. *for all* $l \in dom(S)$*, if* $S'(l) = \textbf{\textit{err}}$ *then* $S(l) = \textbf{\textit{err}}$*.*

Intuitively, $(\Gamma, S) \Rightarrow (\Gamma', S')$ means the **err**-bound locations in S' are either also **err**-bound in S, or if they are fresh (do not appear in Γ).

Lemma B.5 *If* $dom(\Gamma) = dom(S)$, *then* $(\Gamma, S) \Rightarrow (\Gamma, S)$.

Lemma B.6 *If* $(\Gamma, S) \Rightarrow (\Gamma', S')$ *and* $(\Gamma', S') \Rightarrow (\Gamma'', S'')$, *then* $(\Gamma, S) \Rightarrow (\Gamma'', S'')$.

Lemma B.7 (ρ-Renaming) *If* $\Gamma \vdash e : t; L$ *and* $\rho' \notin (\mathit{loceff}(\Gamma) \cup \mathit{loceff}(t) \cup L)$, *then* $R\Gamma \vdash e : Rt; RL$ *for substitution* $R = [\rho \mapsto \rho']$.

Proof: The assumption $\rho' \notin (\mathit{loceff}(\Gamma) \cup \mathit{loceff}(t) \cup L)$ means that ρ' is completely fresh: it is does not appear anywhere in the proof of $\Gamma \vdash e : t; L$. □

Lemma B.8 *If* $\Gamma'|_{dom(\Gamma)} = \Gamma$ *and* $\Gamma \vdash e : t; L$, *then* $\Gamma' \vdash e : t; L'$ *where* $L' \subseteq L$

Proof: By induction on the structure of the proof $\Gamma \vdash e : t; L$. For all cases except (Restrict$_a$) and (Down$_a$) this is trivial, since adding new bindings to an environment does not affect typability. For (Restrict$_a$) and (Down$_a$), given an assumption that a location $\rho \notin \mathit{loceff}(\Gamma)$, we need to show $\rho \notin \mathit{loceff}(\Gamma')$, which clearly does not hold for arbitrary Γ'. But we observe that if we construct a typing proof with assumptions Γ and $\rho \notin \mathit{loceff}(\Gamma)$, then the name ρ was arbitrary. Thus we can rename location ρ to a fresh location $\rho' \notin \mathit{loceff}(\Gamma')$ and repeat our proof. We also use this observation in the subject reduction proof (Theorem B.10).

Suppose the last rule applied is (Down$_a$):

$$\frac{\Gamma \vdash e : t; L \qquad \rho \notin \mathit{loceff}(\Gamma) \cup \mathit{loceff}(t)}{\Gamma \vdash e : t; L - \rho} \tag{B.1}$$

Pick a completely fresh ρ', that is, a $\rho' \notin \mathit{loceff}(\Gamma') \cup \mathit{loceff}(t) \cup L$, and let $C_I(\rho') = C_I(\rho)$ and $R = [\rho \mapsto \rho']$. Then since $\Gamma'|_{dom(\Gamma)} = \Gamma$, by (B.1) and Lemma B.7 we have $R\Gamma \vdash e : Rt; RL$ or $\Gamma \vdash e : t; RL$. Then by induction we have $\Gamma' \vdash e : t; L''$ where $L'' \subseteq RL$. Now by construction of ρ', we can apply (Down$_a$) to get $\Gamma' \vdash e : t; L'' - \rho'$ and $L'' - \rho' \subseteq L - \rho$, proving our conclusion.

Suppose that the last rule applied is (Restrict$_a$):

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \mathit{ref}\ (\rho); L_1 \qquad\qquad C_I(\rho') = C_I(\rho) \\ \Gamma[x \mapsto \mathit{ref}\ (\rho')] \vdash e_2 : t_2; L_2 \\ \rho \notin L_2 \qquad \rho' \notin \mathit{loceff}(\Gamma) \cup \mathit{loceff}(C_I(\rho)) \cup \mathit{loceff}(t_2)\end{array}}{\Gamma \vdash \mathtt{restrict}\, x = e_1 \,\mathtt{in}\, e_2 : t_2; L_1 \cup L_2 \cup \rho} \tag{B.2}$$

By induction, we have $\Gamma' \vdash e_1 : \mathit{ref}\ (\rho); L_1'$ where $L_1' \subseteq L_1$. Pick a completely fresh ρ'', that is, a $\rho'' \notin \mathit{loceff}(\Gamma') \cup \mathit{loceff}(C_I(\rho')) \cup \mathit{loceff}(t_2) \cup L_2 \cup \rho' \cup \rho$, and let $R = [\rho' \mapsto \rho'']$ and $C_I(\rho'') = C_I(\rho')(= C_I(\rho))$. Then by (B.2) and Lemma B.7 we have $R(\Gamma[x \mapsto \mathit{ref}\ (\rho')]) \vdash e_2 : Rt_2; RL_2$ or $\Gamma[x \mapsto \mathit{ref}\ (\rho'')] \vdash e_2 : t_2; RL_2$. By induction we have $\Gamma'[x \mapsto \mathit{ref}\ (\rho'')] \vdash e_2 : t_2; L_2'$ where $L_2' \subseteq RL_2$. Also, we know $\rho \notin L_2$, and since by construction $\rho'' \neq \rho$ we know that $\rho \notin L_2'$. Further, by construction $\rho'' \notin \mathit{loceff}(\Gamma') \cup \mathit{loceff}(C_I(\rho)) \cup \mathit{loceff}(t_2)$. Thus we can apply (Restrict$_a$) to our transformed hypotheses to show that $\Gamma' \vdash \texttt{restrict}\, x = e_1\, \texttt{in}\ e_2 : t_2; L_1' \cup L_2' \cup \rho$ Now there's a small hitch, because the transformed effect of `restrict` may contain ρ''. But since $\rho'' \notin \mathit{loceff}(\Gamma') \cup \mathit{loceff}(t_2)$, we can apply (Down$_a$) to yield $\Gamma' \vdash \texttt{restrict}\, x = e_1\, \texttt{in}\ e_2 : t_2; (L_1' \cup L_2' \cup \rho) - \rho''$ It is easy to see that $(L_1' \cup L_2' \cup \rho) - \rho'' \subseteq L_1 \cup L_2 \cup \rho$. By induction we know $L_1' \subseteq L_1$ and $L_2' \subseteq RL_2$. Since $\rho'' \neq \rho$ the latter implies $L_2' - \rho'' \subseteq L_2$, proving our conclusion. This removal of ρ'' from the effect set is the reason that the conclusion of our lemma is that $L' \subseteq L$ and not necessarily $L' = L$. □

Lemma B.9 (Substitution) *If* $\Gamma \vdash v : t; \emptyset$ *and* $\Gamma[x \mapsto t] \vdash e : t'; L$, *then* $\Gamma \vdash e[x \mapsto v] : t'; L$.

Note that for the substitution lemma to hold we implicitly assume that we rename any bound variables in e to avoid capturing any free variables (i.e., locations) in v, as is standard.

With these definitions we can prove our subject reduction theorem. We use r to stand for a semantic reduction result, either a value v or **err**.

Theorem B.10 (Subject Reduction) *If* $\Gamma \vdash e : t; L$ *and* $S \vdash e \to r; S'$, *where* $(\Gamma, L \cup L') \sim S$ *for some* L', *then there exists* Γ' *such that*

1. $\Gamma' \vdash r : t; \emptyset$ *(which implies* $r \neq \textbf{\textit{err}}$*),*
2. $(\Gamma', L') \sim S'$, *and*
3. $(\Gamma, S) \Rightarrow (\Gamma', S')$

Proof: By induction on the structure of the proof $S \vdash e \to r; S'$. Recall that for each rule of Figure B.1 there are corresponding reductions to **err** for cases for invalid programs. Thus for each case below, we first reason based on the shape of e to decide which possible rule we used and then show that r is not **err**.

Before beginning the case analysis, we first perform some generic reasoning to eliminate uses of (Sub$_a$) and (Down$_a$) at the end of the proof of $\Gamma \vdash e : t; L$.

Case ($\mathbf{Sub}_a$)

Observe that we may have applied (Sub_a) as the last step of our typing proof:

$$\frac{\Gamma \vdash e : t; L \qquad t \leq t'}{\Gamma \vdash e : t'; L} \tag{B.3}$$

By assumption we also know

$$S \vdash e \to r; S' \tag{B.4}$$

$$(\Gamma, L \cup L') \sim S \tag{B.5}$$

Then by applying our reasoning for (Sub_a) and (Down_a) as many times as necessary, and by (B.3), (B.4), (B.5), and the case analysis below, there exists a Γ' such that

$$\Gamma' \vdash r : t; \emptyset \tag{B.6}$$

$$(\Gamma', L') \sim S'$$

$$(\Gamma, S) \Rightarrow (\Gamma', S')$$

Then by combining (B.6) and (B.3) we have

$$\Gamma' \vdash r : t'; \emptyset$$

and thus our conclusion holds.

Case ($\mathbf{Down}_a$)

Observe that we may have applied (Down_a) as the last step of our typing proof:

$$\frac{\Gamma \vdash e : t; L \qquad \rho \notin \mathit{loceff}(\Gamma) \cup \mathit{loceff}(t)}{\Gamma \vdash e : t; L - \rho} \tag{B.7}$$

By assumption we also know

$$S \vdash e \to r; S' \tag{B.8}$$

$$(\Gamma, (L - \rho) \cup L') \sim S \tag{B.9}$$

Pick a fresh ρ', that is,

$$\rho' \notin \mathit{loceff}(\Gamma) \cup \mathit{loceff}(t) \cup L \cup L' \cup \rho \tag{B.10}$$

Let $R = [\rho \mapsto \rho']$. Then from (B.7) and Lemma B.7 we have

$$R\Gamma \vdash e : Rt; RL$$

Then by (B.10) we derive

$$\Gamma \vdash e : t; RL \tag{B.11}$$

In other words, because ρ did not escape the scope of e, its name was arbitrary, and we can repeat the typing proof of e with any choice of name.

Now we want to show

$$(\Gamma, RL \cup L') \sim S \tag{B.12}$$

Clearly from (B.9) we have $dom(\Gamma) = dom(S)$, and for all $m \in dom(S)$ there is a ρ_m such that $\Gamma(m) = ref\ (\rho_m)$. If $S(m) \neq \mathbf{err}$ then we are done, since also from (B.9) we have $\Gamma \vdash S(m) : C_I(\rho_m); \emptyset$. So suppose that $S(m) = \mathbf{err}$. Then from (B.9) we know $\rho_m \notin (L - \rho) \cup L'$. But since $\rho_m \in loceff(\Gamma)$ and $\rho' \notin loceff(\Gamma)$ from (B.10) we know that $\rho' \neq \rho_m$. Thus

$$\rho_m \notin (L - \rho) \cup \rho' \cup L'$$

and therefore $\rho_m \notin RL \cup L'$. Thus (B.12) holds.

Then by applying our reasoning for (Sub_a) and (Down_a) as many times as necessary, and by (B.11), (B.8), (B.12), and the case analysis below, there exists a Γ' such that

$$\Gamma' \vdash r : t; \emptyset$$

$$(\Gamma', L') \sim S'$$

$$(\Gamma, S) \Rightarrow (\Gamma', S')$$

Thus our conclusion holds.

In the remaining cases, we assume that the last rule applied in the typing proof was neither (Sub_a) nor (Down_a).

Case x

By assumption we have

$$\frac{x \in dom(\Gamma)}{\Gamma \vdash x : \Gamma(x); \emptyset} \tag{B.13}$$

$$S \vdash x \to r; S' \tag{B.14}$$

$$(\Gamma, \emptyset \cup L') \sim S \tag{B.15}$$

By (B.15), $dom(\Gamma) = dom(S)$, so by (B.13) we know $x \in dom(S)$. Therefore we must have used the reduction

$$\frac{x \in dom(S)}{S \vdash x \to x; S} \tag{B.16}$$

Thus $r = x$ and $S' = S$. Let $\Gamma' = \Gamma$. Then our conclusion trivially holds:

$$\Gamma' \vdash x : \Gamma(x); \emptyset$$

$$(\Gamma', L') \sim S'$$

$$(\Gamma, S) \Rightarrow (\Gamma', S')$$

The last conclusion holds by Lemma B.5.

Case n

Trivial (see proof for x).

Case $\lambda x.e$

Trivial (see proof for x).

Case $e_1\ e_2$

By assumption we have

$$\frac{\Gamma \vdash e_1 : t \longrightarrow^L t'; L_1 \qquad \Gamma \vdash e_2 : t; L_2}{\Gamma \vdash e_1\ e_2 : t'; L_1 \cup L_2 \cup L} \tag{B.17}$$

$$S \vdash e_1\ e_2 \to r; S' \tag{B.18}$$

$$(\Gamma, L_1 \cup L_2 \cup L \cup L') \sim S \tag{B.19}$$

By (B.18) and inspection of the semantic rules, we must have applied a reduction for e_1:

$$S \vdash e_1 \to r_{e_1}; S'_{e_1} \tag{B.20}$$

By (B.17), (B.20), (B.19), and induction, there exists a Γ'_{e_1} satisfying

$$\Gamma'_{e_1} \vdash r_{e_1} : t \longrightarrow^L t'; \emptyset \tag{B.21}$$

$$(\Gamma'_{e_1}, L_2 \cup L \cup L') \sim S'_{e_1} \tag{B.22}$$

$$(\Gamma, S) \Rightarrow (\Gamma'_{e_1}, S'_{e_1}) \tag{B.23}$$

By (B.21) we know that r_{e_1} is a value and is not **err**. By inspection of the type rules we see that the only type rules that can assign a value the type $t \longrightarrow^L t'$ are (Var_a) and (Lam_a). But by (B.22) we know that Γ'_{e_1} assigns only reference types. Thus the proof (B.21) must in fact be

$$\frac{\dfrac{\Gamma[x \mapsto t_s] \vdash e : t'_s; L_s}{\Gamma \vdash \lambda x.e : t_s \longrightarrow^{L_s} t'_s; \emptyset} \qquad t \le t_s \qquad t'_s \le t' \qquad L_s \subseteq L}{\Gamma \vdash \lambda x.e : t \longrightarrow^L t'; \emptyset} \tag{B.24}$$

where $r_{e_1} = \lambda x.e$ (we can assume that (Sub_a) was only used once by transitivity of $\leq$).

Further, by inspection of the semantic rules, in (B.18) we must also have applied a reduction for e_2:

$$S'_{e_1} \vdash e_2 \to r_{e_2}; S'_{e_2} \tag{B.25}$$

By (B.17), (B.23), and Lemma B.8, we have

$$\Gamma'_{e_1} \vdash e_2 : t; L'_2 \tag{B.26}$$

where $L'_2 \subseteq L_2$. Then by (B.26), (B.25), (B.22), and induction, there exists a Γ'_{e_2} such that

$$\Gamma'_{e_2} \vdash r_{e_2} : t; \emptyset \tag{B.27}$$

$$(\Gamma'_{e_2}, L \cup L') \sim S'_{e_2} \tag{B.28}$$

$$(\Gamma'_{e_1}, S'_{e_1}) \Rightarrow (\Gamma'_{e_2}, S'_{e_2}) \tag{B.29}$$

From (B.27) we know that r_{e_2} is not **err**. Thus by inspection of the semantic rules, in (B.18) we must also have applied a reduction for $e[x \mapsto r_{e_2}]$:

$$S'_{e_2} \vdash e[x \mapsto r_{e_2}] \to r_e; S'_e \tag{B.30}$$

Combining (B.23) and (B.29) we see

$$(\Gamma, S) \Rightarrow (\Gamma'_{e_2}, S'_{e_2}) \tag{B.31}$$

Now by (B.24) and (B.31) with Lemma B.8, we see that

$$\Gamma'_{e_2}[x \mapsto t_s] \vdash e : t'_s; L'_s \tag{B.32}$$

where $L'_s \subseteq L_s \subseteq L$. By (B.27) and $t \leq t_s$ from (B.24) we see that

$$\Gamma'_{e_2} \vdash r_{e_2} : t_s; \emptyset \tag{B.33}$$

Then by (B.32), (B.33), and Lemma B.9 we have

$$\Gamma'_{e_2} \vdash e[x \mapsto r_{e_2}] : t'_s; L'_s \tag{B.34}$$

Since $L'_s \subseteq L_s \subseteq L$, from (B.28) and Lemma B.2 we have

$$(\Gamma'_{e_2}, L'_s \cup L') \sim S'_{e_2} \tag{B.35}$$

Now by (B.34), (B.30), (B.35), and induction, there exists a Γ'_e such that

$$\Gamma'_e \vdash r_e : t'_s; \emptyset \tag{B.36}$$

$$(\Gamma'_e, L') \sim S'_e \tag{B.37}$$

$$(\Gamma'_{e_2}, S'_{e_2}) \Rightarrow (\Gamma'_e, S'_e) \tag{B.38}$$

where $r = r_e$ and $S' = S'_e$ (see (B.18)). Let $\Gamma' = \Gamma'_e$. Then clearly since $t'_s \leq t'$ by (B.24) we have

$$\Gamma' \vdash r : t'; \emptyset$$

from (B.36). Then we also have

$$(\Gamma', L') \sim S'$$

from (B.37). Finally, we get

$$(\Gamma, S) \Rightarrow (\Gamma', S')$$

by combining (B.31) with (B.38). Thus our conclusion holds.

Case $\texttt{let}\, x = e_1\, \texttt{in}\, e_2$

In a monomorphic type system $\texttt{let}\, x = e_1\, \texttt{in}\, e_2$ is equivalent to $(\lambda x.e_2)\ e_1$, so we can simply apply those the proof steps for application and abstraction.

Case $\texttt{ref}\, e$

By assumption we have

$$\frac{\Gamma \vdash e : t; L \qquad S_I(\rho) = t}{\Gamma \vdash \texttt{ref}\, e : ref\ (\rho); L \cup al(\rho)} \tag{B.39}$$

$$S \vdash \texttt{ref}\, e \rightarrow r; S' \tag{B.40}$$

$$(\Gamma, L \cup al(\rho) \cup L') \sim S \tag{B.41}$$

By (B.40) and inspection of the semantic rules, we must have applied a reduction for e:

$$S \vdash e \rightarrow r_e; S'_e \tag{B.42}$$

By (B.39), (B.42), (B.41), and induction, there exists a Γ'_e satisfying

$$\Gamma'_e \vdash r_e : t; \emptyset \tag{B.43}$$

$$(\Gamma'_e, al(\rho) \cup L') \sim S'_e \tag{B.44}$$

$$(\Gamma, S) \Rightarrow (\Gamma'_e, S'_e) \tag{B.45}$$

By (B.43), r_e is not **err**. Thus the reduction (B.40) must in fact be

$$\frac{S \vdash e \to r_e; S'_e \qquad l \notin dom(S'_e)}{S \vdash \texttt{ref}\, e \to l; S'_e[l \mapsto r_e]} \tag{B.46}$$

with $S' = S'_e[l \mapsto r_e]$ and $r = l$ (see (B.40)). Let

$$\Gamma' = \Gamma'_e[l \mapsto ref\ (\rho)]$$

Clearly

$$\Gamma' \vdash l : ref\ (\rho); \emptyset$$

Further, since by (B.46) we have $l \notin dom(S'_e)$, we also have $l \notin dom(\Gamma'_e)$ by (B.44), and therefore $l \notin dom(\Gamma)$ by (B.45). Thus (Γ', S') is an extension of (Γ, S). Further, since $S'(l) = r_e \neq \mathbf{err}$ we have

$$(\Gamma'_e, S'_e) \Rightarrow (\Gamma', S') \tag{B.47}$$

Combining (B.47) and (B.45) by Lemma B.6, we have

$$(\Gamma, S) \Rightarrow (\Gamma', S')$$

Finally, by (B.44) and Lemma B.2 we also have

$$(\Gamma', L') \sim S'$$

since $l \notin dom(S'_e)$ and $r_e \neq \mathbf{err}$, and by (B.39) we know $S_I(\rho) = t$. Thus our conclusion holds.

Note that we did not need to use the fact that the effect $al(\rho)$ is safe after evaluating e. Intuitively this is because allocation writes to a known location—the one that was allocated—before that location can be conflated with any other location by our abstract location approximation. This implies that allocation does not need to be treated as an effect in order to ensure the correctness of `restrict`, though we use allocation effects to improve the precision of the flow-sensitive type qualifier system.

Case $*e$

By assumption we have

$$\frac{\Gamma \vdash e : ref\ (\rho); L}{\Gamma \vdash *e : S_I(\rho); L \cup rd(\rho)} \tag{B.48}$$

$$S \vdash *e \to r; S' \tag{B.49}$$

$$(\Gamma, L \cup rd(\rho) \cup L') \sim S \tag{B.50}$$

By (B.49) and inspection of the semantic rules, we must have a reduction for e:

$$S \vdash e \to r_e; S'_e \tag{B.51}$$

By (B.48), (B.51), (B.50), and induction, there exists a Γ'_e satisfying

$$\Gamma'_e \vdash r_e : ref\ (\rho); \emptyset \tag{B.52}$$

$$(\Gamma'_e, rd(\rho) \cup L') \sim S'_e \tag{B.53}$$

$$(\Gamma, S) \Rightarrow (\Gamma'_e, S'_e) \tag{B.54}$$

By (B.52) we know r_e is a value and is not **err**. By inspection of the type rules we see that the only type rule that can assign a value the type $ref\ (\rho)$ is (Var_a). (As an aside, notice that any uses of (Sub_a) in the proof (B.52) must be trivial, since there is no subtyping under a *ref* type constructor.) Therefore we see that $r_e \in dom(\Gamma'_e)$, hence r_e is in fact a location and $r_e \in dom(S'_e)$ by (B.53). Therefore the reduction (B.49) must in fact be

$$\frac{S \vdash e \to r_e; S'_e \qquad r_e \in dom(S'_e)}{S \vdash *e \to S'_e(r_e); S'_e} \tag{B.55}$$

with $S' = S'_e$ and $r = S'_e(r_e)$ (see (B.49)). Let $\Gamma' = \Gamma'_e$. Then clearly

$$(\Gamma', L') \sim S'$$

by (B.53) and Lemma B.2, and

$$(\Gamma, S) \Rightarrow (\Gamma', S')$$

by (B.54). Further, by (B.52) we know $\Gamma'(r_e) = ref\ (\rho)$. Then since $rd(\rho) \in rd(\rho) \cup L'$, by (B.53) we know $S'(r_e) \neq \textbf{err}$ and

$$\Gamma' \vdash S'(r_e) : S_I(\rho); \emptyset$$

Thus our conclusion holds.

Case $e_1 := e_2$

By assumption we have

$$\frac{\Gamma \vdash e_1 : ref\ (\rho); L_1 \qquad \Gamma \vdash e_2 : S_I(\rho); L_2}{\Gamma \vdash e_1 := e_2 : S_I(\rho); L_1 \cup L_2 \cup wr(\rho)} \tag{B.56}$$

$$S \vdash e_1 := e_2 \to r; S' \tag{B.57}$$

$$(\Gamma, L_1 \cup L_2 \cup wr(\rho) \cup L') \sim S \tag{B.58}$$

By (B.57) and inspection of the semantic rules, we must have applied a reduction for e_1:

$$S \vdash e_1 \to r_{e_1}; S'_{e_1} \tag{B.59}$$

By (B.56), (B.59), (B.58), and induction, there exists a Γ'_{e_1} satisfying

$$\Gamma'_{e_1} \vdash r_{e_1} : \mathit{ref}\ (\rho); \emptyset \tag{B.60}$$

$$(\Gamma'_{e_1}, L_2 \cup wr(\rho) \cup L') \sim S'_{e_1} \tag{B.61}$$

$$(\Gamma, S) \Rightarrow (\Gamma'_{e_1}, S'_{e_1}) \tag{B.62}$$

By (B.60) we see that r_{e_1} is not **err**. Thus we must also have applied a reduction for e_2 in (B.57), by inspection of the semantic rules:

$$S'_{e_1} \vdash e_2 \to r_{e_2}; S'_{e_2} \tag{B.63}$$

By (B.56), (B.62), and Lemma B.8 we have

$$\Gamma'_{e_1} \vdash e_2 : S_I(\rho); L'_2 \tag{B.64}$$

where $L'_2 \subseteq L_2$. Since $L'_2 \subseteq L_2$, by (B.61) and Lemma B.2 we have

$$(\Gamma'_{e_1}, L'_2 \cup wr(\rho) \cup L') \sim S'_{e_1} \tag{B.65}$$

Then by (B.64), (B.63), (B.65), and induction, there exists a Γ'_{e_2} satisfying

$$\Gamma'_{e_2} \vdash r_{e_2} : S_I(\rho); \emptyset \tag{B.66}$$

$$(\Gamma'_{e_2}, wr(\rho) \cup L') \sim S'_{e_2} \tag{B.67}$$

$$(\Gamma'_{e_1}, S'_{e_1}) \Rightarrow (\Gamma'_{e_2}, S'_{e_2}) \tag{B.68}$$

By (B.60) we know that r_{e_1} is a value and is not **err**. By inspection of the type rules we see that the only type rule that can assign a value the type *ref* (ρ) is (Var_a). (As an aside, notice that any uses of (Sub_a) in the proof (B.60) must be trivial, since there is no subtyping under a *ref* type constructor.) Therefore we see that $r_{e_1} \in \mathit{dom}(\Gamma'_{e_1})$, hence r_{e_1} is in fact a location and $r_{e_1} \in \mathit{dom}(S'_{e_1})$ by (B.61).

Then by (B.68) we know that $r_{e_1} \in dom(S'_{e_2})$ and $\Gamma'_{e_2} \vdash r_{e_2} : ref\,(\rho); \emptyset$. Then since $wr(\rho) \in wr(\rho) \cup L'$, by (B.67) it must be that $S'_{e_2}(r_{e_1})$ is not **err**. Therefore the reduction (B.57) must in fact be

$$\frac{\begin{array}{cc} S \vdash e_1 \to r_{e_1}; S'_{e_1} & S'_{e_1} \vdash e_2 \to r_{e_2}; S'_{e_2} \\ r_{e_1} \in dom(S'_{e_2}) & S'_{e_2}(r_{e_1}) \neq \mathbf{err} \end{array}}{S \vdash e_1 := e_2 \to r_{e_2}; S'_{e_2}[r_{e_1} \mapsto r_{e_2}]} \tag{B.69}$$

where $S' = S'_{e_2}[r_{e_1} \mapsto r_{e_2}]$ and $r = r_{e_2}$ (see (B.57)). Let $\Gamma' = \Gamma'_{e_2}$. Clearly we have

$$\Gamma' \vdash r_{e_2} : S_I(\rho); \emptyset$$

by (B.66). Combining (B.68) and (B.62) we have

$$(\Gamma, S) \Rightarrow (\Gamma'_{e_2}, S'_{e_2}) \tag{B.70}$$

Clearly, then, (Γ', S') is an extension of (Γ, S). And since $S'(r_{e_1}) = r_{e_2} \neq \mathbf{err}$ by (B.66), we have

$$(\Gamma, S) \Rightarrow (\Gamma', S')$$

Finally, since $r_{e_2} \neq \mathbf{err}$, $r_{e_1} \in dom(S'_{e_2})$, and $\Gamma' \vdash r_{e_1} : ref\,(\rho)$, from (B.67) and (B.66) we can conclude

$$(\Gamma', L') \sim S'$$

Thus our conclusion holds.

Case `restrict` $x = e_1$ `in` e_2

By assumption, we know

$$\frac{\begin{array}{cc} \Gamma \vdash e_1 : ref\,(\rho); L_1 & S_I(\rho') = S_I(\rho) \\ \multicolumn{2}{c}{\Gamma[x \mapsto ref\,(\rho')] \vdash e_2 : t_2; L_2} \\ \rho \notin L_2 & \rho' \notin loceff(\Gamma) \cup loceff(S_I(\rho)) \cup loceff(t_2) \end{array}}{\Gamma \vdash \mathtt{restrict}\, x = e_1 \,\mathtt{in}\, e_2 : t_2; L_1 \cup L_2 \cup \rho} \tag{B.71}$$

$$S \vdash \mathtt{restrict}\, x = e_1 \,\mathtt{in}\, e_2 \to r; S' \tag{B.72}$$

$$(\Gamma, L_1 \cup L_2 \cup \rho \cup L') \sim S \tag{B.73}$$

By (B.72) and inspection of the semantic rules, we must have applied a reduction for e_1:

$$S \vdash e_1 \to r_{e_1}; S'_{e_1} \tag{B.74}$$

By (B.71), (B.74), (B.73), and induction, there exists a Γ'_{e_1} such that

$$\Gamma'_{e_1} \vdash r_{e_1} : ref\,(\rho); \emptyset \tag{B.75}$$

$$(\Gamma'_{e_1}, L_2 \cup \rho \cup L') \sim S'_{e_1} \tag{B.76}$$

$$(\Gamma, S) \Rightarrow (\Gamma'_{e_1}, S'_{e_1}) \tag{B.77}$$

By (B.75) we see that r_{e_1} is a value and is not **err**. By inspection of the type rules we see that the only type rule that can assign a value the type *ref* (ρ) is (Var_a). (As an aside, notice that uses of (Sub_a) in the proof (B.75) must be trivial, since there is no subtyping under a *ref* type constructor.) Therefore we see that $r_{e_1} \in dom(\Gamma'_{e_1})$, hence r_{e_1} is in fact a location and $r_{e_1} \in dom(S'_{e_1})$ by (B.76). Thus by inspection of the semantic rules, in (B.72) we must also have applied

$$S'_{e_1}[r_{e_1} \mapsto \mathbf{err}, l' \mapsto S'_{e_1}(r_{e_1})] \vdash e_2[x \mapsto l'] \to r_{e_2}; S'_{e_2} \tag{B.78}$$

with

$$l' \notin dom(S'_{e_1}) \tag{B.79}$$

Before we can apply induction to e_2, we need to do a little more work. We know that ρ' does not appear in Γ, but by coincidence it may appear in Γ'_{e_1}. So before we proceed, we need to rename ρ' to avoid meaningless collisions. Pick a fresh ρ'', that is, pick a ρ'' such that

$$\rho'' \notin \mathit{loceff}(\Gamma'_{e_1}) \cup \mathit{loceff}(S_I(\rho)) \cup \mathit{loceff}(t_2) \cup L_2 \cup L' \cup \rho \tag{B.80}$$

and set $S_I(\rho'') = S_I(\rho')$. Let $R = [\rho' \mapsto \rho'']$. Then by Lemma B.7 and (B.71), we have

$$R(\Gamma[x \mapsto \mathit{ref}\ (\rho')]) \vdash e_2 : Rt_2; RL_2$$

which by (B.80) is equivalent to

$$\Gamma[x \mapsto \mathit{ref}\ (\rho'')] \vdash e_2 : t_2; RL_2$$

Combining this with (B.77), by Lemma B.8 we have

$$\Gamma'_{e_1}[x \mapsto \mathit{ref}\ (\rho'')] \vdash e_2 : t_2; L'_2$$

where $L'_2 \subseteq RL_2$. Then by α-conversion, since $l' \notin dom(S'_{e_1})$ implies $l' \notin \Gamma'_{e_1}$ by (B.76), we can rename x to l' and derive

$$\Gamma'_{e_1}[l' \mapsto \mathit{ref}\ (\rho'')] \vdash e_2[x \mapsto l'] : t_2; L'_2 \tag{B.81}$$

Finally, before we can apply induction, we need to show compatibility between the type environment in (B.81) and the store in (B.78). But which effect set should we use for

compatibility? Clearly the set we choose cannot contain ρ, because the store in (B.78) contains a location corresponding to ρ that maps to **err**. Thus we use the following set L_{e_2}:

$$L_{e_2} = L'_2 \cup (L' - \rho)$$

Also let

$$\Gamma_{e_2} = \Gamma'_{e_1}[l' \mapsto \mathit{ref}\ (\rho'')]$$

$$S_{e_2} = S'_{e_1}[r_{e_1} \mapsto \mathbf{err}, l' \mapsto S'_{e_1}(r_{e_1})]$$

where Γ_{e_2} is from (B.81) and S_{e_2} is from (B.78). Notice that Γ_{e_2} is an extension of Γ'_{e_1}, since by (B.79) and (B.76) $l' \notin \Gamma'_{e_1}$. We want to show

$$(\Gamma_{e_2}, L_{e_2}) \sim S_{e_2} \tag{B.82}$$

To see (B.82), first observe that $r_{e_1} \in \mathit{dom}(S'_{e_1})$ by (B.75) and (B.76), and observe that $\mathit{dom}(\Gamma'_{e_1}) = \mathit{dom}(S'_{e_1})$ by (B.76), and therefore $\mathit{dom}(\Gamma_{e_2}) = \mathit{dom}(S_{e_2})$.

For the second component of compatibility, pick any $m \in \mathit{dom}(S_{e_2})$, and suppose $\Gamma_{e_2}(m) = \mathit{ref}\ (\rho_m)$, which holds trivially by (B.76) and construction of Γ_{e_2}. There are three cases:

1. Suppose $m = r_{e_1}$. Then $\rho_m = \rho$, and by construction of S_{e_2} we have $S_{e_2}(m) = \mathbf{err}$. But $\rho \notin L_2$ by (B.71), and since $\rho'' \neq \rho$ by (B.80) and $L'_2 \subseteq RL_2$, we have $\rho \notin L_{e_2}$.

2. Suppose $m = l'$. Then $\rho_m = \rho''$. By construction of S_{e_2} we have $S_{e_2}(m) = S'_{e_1}(r_{e_1})$. But by (B.75) and (B.76) we have $S'_{e_1}(r_{e_1}) \neq \mathbf{err}$ and $\Gamma'_{e_1} \vdash S'_{e_1}(r_{e_1}) : S_I(\rho)$. But then since Γ_{e_2} is an extension of Γ'_{e_1} and $S_I(\rho'') = S_I(\rho)$ by (B.71) and construction of ρ'' we also have $\Gamma_{e_2} \vdash S'_{e_1}(r_{e_1}) : S_I(\rho''); \emptyset$.

3. Suppose $m \neq r_{e_1}$ and $m \neq l'$. Then $S_{e_2}(m) = S'_{e_1}(m)$ and $\Gamma_{e_2}(m) = \Gamma'_{e_1}(m)$. By (B.80), it must be that $\rho_m \neq \rho''$. If $S'_{e_1}(m) \neq \mathbf{err}$, then by (B.76) we have $\Gamma'_{e_1} \vdash S'_{e_1}(m) : S_I(\rho_m); \emptyset$, and since Γ_{e_2} is an extension of Γ'_{e_1} we also have $\Gamma_{e_2} \vdash S'_{e_1}(m) : S_I(\rho_m); \emptyset$. Otherwise, suppose $S'_{e_1}(m) = \mathbf{err}$. Then by (B.76) we have $\rho_m \notin L_2 \cup \rho \cup L'$. Thus clearly $\rho_m \notin L' - \rho$. Since $\rho_m \notin L_2$ and $\rho'' \neq \rho_m$, we have $\rho_m \notin RL_2$ and thus $\rho_m \notin L'_2$. Therefore $\rho_m \notin L_{e_2}$.

Thus (B.82) holds.

Then by (B.81), (B.78), (B.82), and induction, there exists a Γ'_{e_2} such that

$$\Gamma'_{e_2} \vdash r_{e_2} : t_2; \emptyset \tag{B.83}$$

$$(\Gamma'_{e_2}, L' - \rho) \sim S'_{e_2} \tag{B.84}$$

$$(\Gamma_{e_2}, S_{e_2}) \Rightarrow (\Gamma'_{e_2}, S'_{e_2}) \tag{B.85}$$

Now we're almost done. Combining (B.75) and (B.76) with (B.74), (B.78), and (B.79), we see that the reduction (B.72) must have been

$$\frac{\begin{array}{cc} S \vdash e_1 \to r_{e_1}; S'_{e_1} & S'_{e_1}[r_{e_1} \mapsto \mathbf{err}, l' \mapsto S'_{e_1}(r_{e_1})] \vdash e_2[x \mapsto l'] \to r_{e_2}; S'_{e_2} \\ r_{e_1} \in dom(S'_{e_1}) & l' \notin dom(S'_{e_1}) \end{array}}{S \vdash \texttt{restrict}\, x = e_1 \,\texttt{in}\, e_2 \to r_{e_2}; S'} \tag{B.86}$$

with $r = r_{e_2}$ and

$$S' = S'_{e_2}[r_{e_1} \mapsto S'_{e_2}(l'), l' \mapsto \mathbf{err}]$$

(see (B.72)). Let $\Gamma' = \Gamma'_{e_2}$. We show the conclusions of the inductive hypothesis one by one.

First, by (B.83) we have

$$\Gamma' \vdash r_{e_2} : t_2; \emptyset \tag{B.87}$$

Next we need to show

$$(\Gamma', L') \sim S' \tag{B.88}$$

We proceed as in the proof of (B.82). Clearly $dom(\Gamma') = dom(S'_{e_2})$ by (B.84). And by construction of S_{e_2} we have $r_{e_1}, l' \in dom(S_{e_2})$. Then by (B.85) we see $r_{e_1}, l' \in dom(S'_{e_2})$. Thus $dom(S') = dom(S'_{e_2}) = dom(\Gamma')$.

For the second component of compatibility, pick any $m \in dom(S')$, and suppose $\Gamma'(m) = ref\ (\rho_m)$, which holds trivially by (B.84). There are three cases:

1. Suppose $m = r_{e_1}$. Then $S'(m) = S'_{e_2}(l')$. Since $S_{e_2}(l') = S'_{e_1}(r_{e_1}) \neq \mathbf{err}$ by (B.75) and (B.76), then by (B.85) we see that $S'_{e_2}(l') \neq \mathbf{err}$. By the construction of Γ_{e_2} and (B.85) we see that $\Gamma'_{e_2}(l') = ref\ (\rho'')$. But then by (B.84) we have $\Gamma'_{e_2} \vdash S'_{e_2}(l') : S_I(\rho''); \emptyset$, and hence $\Gamma' \vdash S'(r_{e_1}) : S_I(\rho''); \emptyset$.

2. Suppose $m = l'$. Then $\rho_m = \rho''$ and $S'(l') = \mathbf{err}$. But then by (B.80) we know $\rho'' \notin L'$.

3. Suppose $m \neq r_{e_1}$ and $m \neq l'$. Then $S'(m) = S'_{e_2}(m)$. Suppose $S'_{e_2}(m) \neq \mathbf{err}$. Then from (B.84) we know $\Gamma' \vdash S'(m) : S_I(\rho_m); \emptyset$. Otherwise, suppose $S'(m) = S'_{e_2}(m) = \mathbf{err}$. There are two cases. If $m \in dom(S_{e_2})$, then by (B.85) $S_{e_2}(m) = \mathbf{err}$. By construction of S_{e_2}, we then have $S'_{e_1}(m) = \mathbf{err}$. Then by (B.76) we know $\rho_m \notin L'$. Otherwise, suppose $m \notin dom(S_{e_2})$. Then by (B.85) $\rho_m \notin \mathit{loceff}(\Gamma_{e_2})$. But since $\rho \in \mathit{loceff}(\Gamma_{e_2})$ (which we can conclude from (B.75), (B.77), and the construction of Γ_{e_2}), we know that $\rho_m \neq \rho$. By (B.84) we have $\rho_m \notin L' - \rho$, and since $\rho_m \neq \rho$ we see that $\rho_m \notin L'$.

Thus (B.88) holds. Finally, we need to show

$$(\Gamma, S) \Rightarrow (\Gamma', S') \tag{B.89}$$

Clearly by (B.88) we have $dom(\Gamma') = dom(S')$, and by assumption (B.73) we have $dom(\Gamma) = dom(S)$. Also by (B.77) and (B.85) and the construction of Γ_{e_2} we see that (Γ', S') is an extension of (Γ, S). So we just need to show that it's a safe extension.

Pick any $m \in dom(S')$. If $S'(m) \neq \mathbf{err}$ then we're done. Otherwise suppose $S'(m) = \mathbf{err}$ and $\Gamma'(m) = \mathit{ref}\ (\rho_m)$. Then there are three cases:

1. Suppose $m = r_{e_1}$. This is impossible, since $S'(r_{e_1}) = S'_{e_2}(l') \neq \mathbf{err}$ by the same reasoning used to show (B.88).

2. Suppose $m = l'$. Then $l' \notin dom(S)$ by (B.79) and (B.77). So we need to show $\rho_m \notin \mathit{loceff}(\Gamma)$. But $\rho_m = \rho''$ by construction of Γ_{e_2} and (B.85). And $\rho'' \notin \mathit{loceff}(\Gamma)$ by (B.80) and (B.77).

3. Suppose $m \neq r_{e_1}$ and $m \neq l'$. Then $S'(m) = S'_{e_2}(m)$. If $m \in dom(S'_{e_2}) - dom(S_{e_2})$ then by (B.85) we see that $\rho_m \notin \mathit{loceff}(\Gamma_{e_2})$. But then by construction of Γ_{e_2} and (B.77) we see $\rho_m \notin \mathit{loceff}(\Gamma)$.

   Otherwise if $m \in dom(S_{e_2})$ then by (B.85) we see that $S_{e_2}(m) = \mathbf{err}$. But $S_{e_2}(m) = S'_{e_1}(m)$. Then there are again two cases. If $m \in dom(S'_{e_1}) - dom(S)$, then by (B.77) we see that $\rho_m \notin \mathit{loceff}(\Gamma)$. Otherwise if $m \in dom(S)$ then by (B.77) we see that $S(m) = \mathbf{err}$.

Thus (B.89) holds. Combining (B.87), (B.88), and (B.89) we see that our conclusion holds.

□

Bibliography

[1] 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.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

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

[4] Rita Altucher and William Landi. An Extended Form of Must Alias Analysis for Dynamic Allocation. In *Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 74–84, San Francisco, California, January 1995.

[5] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, May 1994.

[6] ANSI. *Programming languages – C*, 1999. ISO/IEC 9899:1999.

[7] Christophe Bailleux. More security problems in bftpd-1.0.12. BugTraq Mailing List, 8 December 2000. `http://www.securityfocus.com/archive/1/149977`.

[8] Thomas Ball and Sriram K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *The 8th International SPIN Workshop on Model Checking of Software*, number 2057 in Lecture Notes in Computer Science, pages 103–122, May 2001.

[9] Thomas Ball and Sriram K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3, Portland, Oregon, January 2002.

[10] Henk Barendregt. Lambda Calculi with Types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 117–309. Oxford University Press, 1992.

[11] Erik Barendsen and Sjaak Smetsers. Uniqueness Typing for Functional Languages with Graph Rewriting Semantics. *Mathematical Structures in Computer Science*, 6(6):579–612, 1996.

[12] Matt Bishop and Michael Dilger. Checking for Race Conditions in File Accesses. *Computing Systems*, 2(2):131–152, 1996.

[13] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software—Practice and Experience*, 30(7):775–802, June 2000.

[14] Cristiano Calcagno. Stratified Operational Semantics for Safety and Correctness of The Region Calculus. In *Proceedings of the 28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 155–165, London, United Kingdom, January 2001.

[15] CERT Advisory CA-2001-19 "Code Red" Worm Exploiting Buffer Overflow In IIS Indexing Service DLL, 19 July 2001. `http://www.cert.org/advisories/CA-2001-19.html`.

[16] Satish Chandra and Thomas W. Reps. Physical Type Checking for C. In *Proceedings of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 66–75, Toulouse, France, September 1999.

[17] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of Pointers and Structures. In *Proceedings of the 1990 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 296–310, White Plains, New York, June 1990.

[18] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–269, Berlin, Germany, June 2002.

[19] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[20] Crispin Cowan, Matt Barringer, Steve Beattie, and Greg Kroah-Hartman. FormatGuard: Automatic Protection From printf Format String Vulnerabilities. In *Proceedings of the 10th Usenix Security Symposium*, Washington, D.C., August 2001.

[21] Karl Crary, David Walker, and Greg Morrisett. Typed Memory Management in a Calculus of Capabilities. In *Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 262–275, San Antonio, Texas, January 1999.

[22] Manuvir Das. Unification-based Pointer Analysis with Directional Assignments. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46, Vancouver B.C., Canada, June 2000.

[23] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-Sensitive Program Verification in Polynomial Time. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 57–68, Berlin, Germany, June 2002.

[24] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[25] Alan DeKok. PScan: A limited problem scanner for C source files. `http://www.striker.ottawa.on.ca/~aland/pscan`.

[26] Robert DeLine and Manuel Fähndrich. Enforcing High-Level Protocols in Low-Level Software. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 59–69, Snowbird, Utah, June 2001.

[27] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended Static Checking. Technical Report 159, Compaq Systems Research Center, December 1998.

[28] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Demand-driver Computation of Interprocedural Data Flow. In *Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 37–48, San Francisco, California, January 1995.

[29] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A Practical Framework for Demand-Driven Interprocedural Data Flow Analysis. *ACM Transactions on Programming Languages and Systems*, 19(6):992–1030, November 1997.

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

[31] 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.

[32] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–256, Orlando, Florida, June 1994.

[33] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *Fourth symposium on Operating System Design and Implementation*, San Diego, California, October 2000.

[34] 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.

[35] Manuel Fähndrich. *BANE: A Library for Scalable Constraint-Based Program Analysis*. PhD thesis, University of California, Berkeley, 1999.

[36] Manuel Fähndrich and Robert DeLine. Adoption and Focus: Practical Linear Types for Imperative Programming. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–24, Berlin, Germany, June 2002.

[37] Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable Context-Sensitive Flow Analysis using Instantiation Constraints. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 253–263, Vancouver B.C., Canada, June 2000.

[38] Cormac Flanagam and Martín Abadi. Object Types against Races. In Jos C. M. Baeten and Sjouke Mauw, editors, *CONCUR '99: Concurrency Theory, 10th International Conference*, volume 1664, pages 288–303, Eindhoven, The Netherlands, August 1999. Springer-Verlag.

[39] Cormac Flanagam and Martín Abadi. Types for Safe Locking. In Doaitse Swierstra, editor, *8th European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 91–108, Amsterdam, The Netherlands, March 1999. Springer-Verlag.

[40] Cormac Flanagan and Stephen N. Freund. Type-Based Race Detection for Java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 219–232, Vancouver B.C., Canada, June 2000.

[41] Cormac Flanagan and K. Rustan M. Leino. Houdini, an Annotation Assitant for ESC/Java. In J. N. Oliverira and Pamela Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods*, number 2021 in Lecture Notes in Computer Science, pages 500–517, Berlin, Germany, March 2001. Springer-Verlag.

[42] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245, Berlin, Germany, June 2002.

[43] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A Theory of Type Qualifiers. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 192–203, Atlanta, Georgia, May 1999.

[44] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. Polymorphic versus Monomorphic Flow-insensitive Points-to Analysis for C. In Jens Palsberg, editor, *Static Analysis, Seventh International Symposium*, volume 1824 of *Lecture Notes in Computer Science*, pages 175–198, Santa Barbara, CA, June/July 2000. Springer-Verlag.

[45] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-Sensitive Type Qualifiers. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, Berlin, Germany, June 2002.

[46] Przemyslaw Frasunek. format string vulnerability in mars_nwe 0.99pl19, 26 January 2001. `http://online.securityfocus.com/archive/1/158959`.

[47] Przemyslaw Frasunek. ports/24733: mars_nwe remote format string vulnerability, 30 January 2001. `http://groups.google.com/groups?q=mars_nwe+vulnerability&hl=en&lr=&ie=U%TF-8&oe=UTF-8&selm=9566si%24gpv%241%40FreeBSD.csie.NCTU.edu.tw&rnum=1`.

[48] Tim Freeman and Frank Pfenning. Refinement Types for ML. In *Proceedings of the 1991 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 268–277, Toronto, Ontario, Canada, June 1991.

[49] Bill Gates. Trustworthy computing. Microsoft internal memo. Available at `http://www.theregister.co.uk/content/4/23715.html`, 15 January 2002.

[50] David Gay and Alexander Aiken. Language Support for Regions. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 70–80, Snowbird, Utah, June 2001.

[51] David K. Gifford, Pierre Jouvelot, John M. Lucassen, and Mark A. Sheldon. FX-87 Reference Manual. Technical Report MIT/LCS/TR-407, MIT Laboratory for Computer Science, September 1987.

[52] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-Based Memory Management in Cyclone. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 282–293, Berlin, Germany, June 2002.

[53] Dan Grossman, Greg Morrisett, Yanling Wang, Trevor Jim, Michael Hicks, and James Cheney. Cyclone User's Manual. Technical Report 2001-1855, Department of Computer Science, Cornell University, November 2001.

[54] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A System and Language for Building System-Specific, Static Analyses. In *Proceedings of the 2002 ACM*

SIGPLAN Conference on Programming Language Design and Implementation, pages 69–82, Berlin, Germany, June 2002.

[55] Chris Hankin. *Lambda Calculi: A Guide for Computer Scientists*. Oxford University Press Inc., New York, 1994.

[56] Christopher Harrelson. Program Analysis Mode. `http://www.cs.berkeley.edu/~chrishtr/pam`.

[57] Nevin Heintze and Olivier Tardieu. Ultra-fast Aliasing Analysis using CLA: A Million Lines of C Code in a Second. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 254–263, Snowbird, Utah, June 2001.

[58] Maxime Henrion. muh IRC bouncer remote vulnerability. FreeBSD-SA-00:57, 13 October 2000. `http://www.securityfocus.com/advisories/2741`.

[59] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy Abstraction. In *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 58–70, Portland, Oregon, January 2002.

[60] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979.

[61] Susan Horwitz, Thomas Reps, and Mooly Sagiv. Demand Interprocedural Dataflow Analysis. In *Third Symposium on the Foundations of Software Engineering*, pages 104–115, Wasington, DC, October 1995.

[62] Jarno Huuskonen. Possibility for formatchar errors in syslog call, September 2000. `https://bugzilla.redhat.com/bugzilla/show_bug.cgi?id=17349`.

[63] Jarno Huuskonen. Some possible format string errors. Linux Security Audit Project Mailing List, 25 September 2000. `http://www2.merton.ox.ac.uk/~security/security-audit-200009/0118.html`.

[64] Jarno Huuskonen. syslog(prio, buf) in mars_nwe. Linux Security Audit Project Mailing List, 27 September 2000. `http://www2.merton.ox.ac.uk/~security/security-audit-200009/0136.html`.

[65] Atsushi Igarashi and Naoki Kobayashi. Resource Usage Analysis. In *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 331–342, Portland, Oregon, January 2002.

[66] Suresh Jagannathan, Peter Thiemann, Stephen Weeks, and Andrew Wright. Single and loving it: Must-alias analysis for higher-order languages. In *Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 329–341, San Diego, California, January 1998.

[67] John B. Kam and Jeffrey D. Ullman. Global Data Flow Analysis and Iterative Algorithms. *Journal of the ACM*, 23(1):158–171, January 1976.

[68] 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.

[69] William Landi and Barbara G. Ryder. Pointer-induced Aliasing: A Problem Classification. In *Proceedings of the 18th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 93–103, Orlando, Florida, January 1991.

[70] William Landi and Barbara G. Ryder. A Safe Approximate Algorithm for Interprocedural Pointer Aliasing. In *Proceedings of the 1992 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 235–248, San Francisco, California, June 1992.

[71] David Larochelle and David Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *Proceedings of the 10th Usenix Security Symposium*, Washington, D.C., August 2001.

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

[73] Ben Liblit and Alexander Aiken. Type Systems for Distributed Data Structures. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 199–213, Boston, Massachusetts, January 2000.

[74] John M. Lucassen. *Types and Effects: Towards the Integration of Functional and Imperative Programming*. PhD thesis, MIT Laboratory for Computer Science, August 1987. MIT/LCS/TR-408.

[75] 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.

[76] Mars Climate Orbiter Mishap Investigation Board. Phase I Report, 10 November 1999. `ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf`.

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

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

[79] Anders Møller and Michael I. Schwartzbach. The Pointer Assertion Logic Engine. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 221–231, Snowbird, Utah, June 2001.

[80] Christian Mossin. *Flow Analysis of Typed Higher-Order Programs*. PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, 1996.

[81] George Necula, Scott McPeak, and Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 128–139, Portland, Oregon, January 2002.

[82] Tim Newsham. Format String Attacks, September 2000. `http://online.securityfocus.com/guest/3342`.

[83] Robert O'Callahan. A Simple, Comprehensive Type System for Java Bytecode Subroutines. In *Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 70–78, San Antonio, Texas, January 1999.

[84] Robert O'Callahan. *Generalized Aliasing as a Basis for Program Analysis Tools*. PhD thesis, School of Computer Science, Carnegie Mellon University, November 2000.

[85] Robert O'Callahan and Daniel Jackson. Lackwit: A Program Understanding Tool Based on Type Inference. In *Proceedings of the 19th International Conference on Software Engineering*, pages 338–348, Boston, Massachusetts, May 1997.

[86] Kurt M. Olender and Leon J. Osterweil. Interprocedural Static Analysis of Sequencing Constraints. *ACM Transactions on Software Engineering and Methodology*, 1(1):21–52, January 1992.

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

[88] Jonathan D. Pincus. Personal communication, 2002.

[89] President's Information Technology Advisory Committee Report to the President, 24 February 1999. `http://www.ccic.gov/ac/report`.

[90] G. D. Plotkin. A Structural Approach to Operational Semantics. University of Aarhus, Denmark.

[91] Vaughan Pratt and Jerzy Tiuryn. Satisfiability of Inequalities in a Poset. *Fundamenta Informaticae*, 28(1-2):165–182, 1996.

[92] Jakob Rehof and Manuel Fähndrich. Type-Based Flow Analysis: From Polymorphic Subtyping to CFL-Reachability. In *Proceedings of the 28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 54–66, London, United Kingdom, January 2001.

[93] Jakob Rehof and Torben Æ. Mogensen. Tractable Constraints in Finite Semilattices. In Radhia Cousot and David A. Schmidt, editors, *Static Analysis, Third International Symposium*, volume 1145 of *Lecture Notes in Computer Science*, pages 285–300, Aachen, Germany, September 1996. Springer-Verlag.

[94] Tim J. Robbins. libformat–protection against format string attacks. `http://box3n.gumbynet.org/~fyre/software/libformat.html`.

[95] Alessandro Rubini and Jonathan Corbet. *Linux Device Drivers*. O'Reilly & Associates, 2nd edition edition, June 2001.

[96] Erik Ruf. Context-Insensitive Alias Analysis Reconsidered. In *Proceedings of the 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–22, La Jolla, California, June 1995.

[97] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise Interprocedural Dataflow Analysis with Applicatios to Constant Propagation. *Theoretical Computer Science*, 167(1&2):131–170, October 1996.

[98] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric Shape Analysis via 3-Valued Logic. In *Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 105–118, San Antonio, Texas, January 1999.

[99] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 27–37, St. Malo, France, October 1997.

[100] Pekka Savola. Very probable remote root vulnerability in cfengine. BugTraq Mailing List, 2 October 2000. `http://www.securityfocus.com/archive/1/136751`.

[101] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th Usenix Security Symposium*, Washington, D.C., August 2001.

[102] Olin Shivers. Control-Flow Analysis in Scheme. In *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 164–174, Atlanta, Georgia, June 1988.

[103] Christian Skalka and Scott Smith. Static Enforcement of Security with Types. In *Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming*, pages 34–45, Montreal, Canada, September 2000.

[104] Frederick Smith, David Walker, and Greg Morrisett. Alias Types. In Gert Smolka, editor, *9th European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–381, Berlin, Germany, 2000. Springer-Verlag.

[105] Geoffrey Smith and Dennis Volpano. Secure Information Flow in a Multi-Threaded Imperative Language. In *Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 355–364, San Diego, California, January 1998.

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

[107] Raymie Stata and Martín Abadi. A Type System for Java Bytecode Subroutines. In *Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 149–160, San Diego, California, January 1998.

[108] Bjarne Steensgaard. Points-to Analysis in Almost Linear Time. In *Proceedings of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, St. Petersburg Beach, Florida, January 1996.

[109] Robert E. Strom and Shaula Yemini. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, January 1986.

[110] Yan Mei Tang and Pierre Jouvelot. Effect Systems with Subtyping. In *Proceedings of theACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 45–53, La Jolla, California, USA, June 1995.

[111] 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.

[112] David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *FPCA '95 Conference on Functional Programming Languages and Computer Architecture*, pages 1–11, La Jolla, California, June 1995.

[113] John Viega, J.T. Bloch, Tadayoshi Kohno, and Gary McGraw. ITS4: A Static Vulnerability Scanner for C and C++ Code. In *16th Annual Computer Security Applications Conference*, December 2000. `http://www.acsac.org`.

[114] 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.

[115] David Walker and Greg Morrisett. Alias Types for Recursive Data Structures. In *International Workshop on Types in Compilation*, Montreal, Canada, September 2000.

[116] David Walker and Kevin Watkins. On Regions and Linear Types. In *Proceedings of the sixth ACM SIGPLAN International Conference on Functional Programming*, pages 181–192, Florence, Italy, September 2001.

[117] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly & Associates, 3rd edition edition, July 2000.

[118] Daniel Weise, 2001. Personal communication.

[119] Robert P. Wilson and Monica S. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. In *Proceedings of the 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, La Jolla, California, June 1995.

[120] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

[121] Andrew K. Wright. Typing References by Effect Inference. In Bernd Krieg-Brücker, editor, *4th European Symposium on Programming*, volume 582 of *Lecture Notes in Computer Science*, pages 473–491, Rennes, France, February 1992. Springer-Verlag.

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

[123] Zhichen Xu, Thomas Reps, and Barton P. Miller. Typestate Checking of Machine Code. In David Sands, editor, *10th European Symposium on Programming*, volume 2028 of *Lecture Notes in Computer Science*, pages 335–351, Genova, Italy, 2001. Springer-Verlag.

[124] 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.

[125] Suan Hsi Yong, Susan Horwitz, and Thomas Reps. Pointer Analysis for Programs with Structures and Casting. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 91–103, Atlanta, Georgia, May 1999.

[126] Sean Zhang, Barbara G. Ryder, and William A. Landi. Experiments with Combined Analysis for Pointer Aliasing. In *Proceedings of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 11–18, Montreal, Canada, June 1998.

[127] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. Using CQUAL for Static Analysis of Authorization Hook Placement. In *Proceedings of the 11th Usenix Security Symposium*, San Francisco, CA, August 2002.