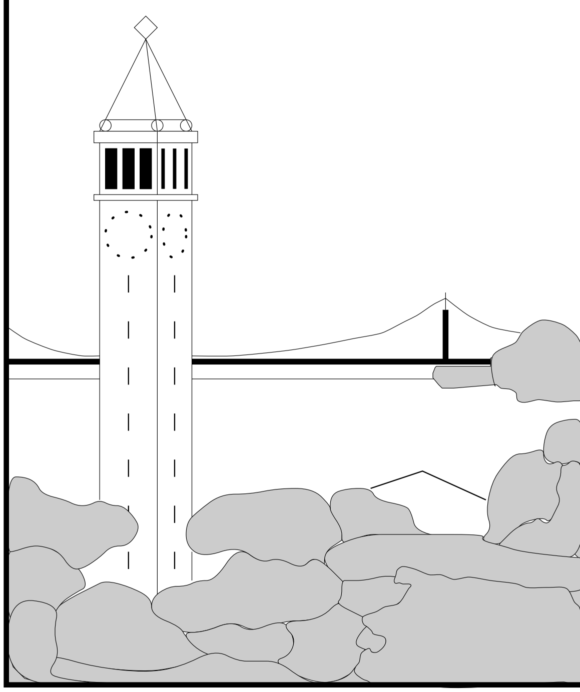
Flow-Sensitive Type Qualifiers

Jeffrey S. Foster *Tachio Terauchi* *Alex Aiken*



Report No. UCB/CSD-01-1162

November 2001

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Flow-Sensitive Type Qualifiers*

Jeffrey S. Foster Tachio Terauchi Alex Aiken
EECS Department
University of California, Berkeley
Berkeley, CA 94720-1776
{jfoster,tachio,aiken}@cs.berkeley.edu

November 2001

Abstract

We present a system for extending standard type systems with flow-sensitive type qualifiers. Users annotate their programs with type qualifiers, and inference checks that the annotations are correct. In our system only the type qualifiers are modeled flow-sensitively—the underlying standard types are unchanged, which allows us to obtain an efficient constraint-based inference algorithm that integrates flow-insensitive alias analysis, effect inference, and ideas from linear type systems to support strong updates. We demonstrate the usefulness of flow-sensitive type qualifiers by finding a number of new locking bugs in the Linux kernel.

1 Introduction

Standard type systems are *flow-insensitive*, meaning a value's type is the same everywhere. However, many important properties are *flow-sensitive*. Checking such properties requires associating different facts with a value at different program points.

This paper shows how to extend standard type systems with user-specified flow-sensitive *type qualifiers*, which are atomic properties that refine standard types. In our system users annotate programs with type qualifiers, and inference checks that the annotations are correct. The critical feature of our approach is that flow-sensitivity is restricted to the type qualifiers that decorate types—the underlying standard types are unchanged—which allows us to obtain an efficient type inference algorithm. Type qualifiers capture a natural class of flow-sensitive properties, while efficient inference of the type qualifiers allows us to apply an implementation to large code bases with few user annotations.

For an example of type qualifiers, consider the type `File` used for I/O operations on files. 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 flow-sensitive type qualifiers. We introduce qualifiers `open`, `read`, `write`, `readwrite`, and `closed`. The type `open File` describes a file that has been opened in an unknown mode, the type `read File` (respectively `write File`) is a file that is open for reading (respectively writing), the type `readwrite File` is a file open for both reading and writing, and the type `closed File` is a closed file. These qualifiers capture inherently flow-sensitive properties. For example, the `close()` function takes an `open File` as an argument and changes the file's state to `closed File`.

These qualifiers have a natural subtyping relation, shown in Figure 1. The qualifier `closed` is incomparable to other qualifiers because a file may not be both closed and open. Qualifiers that introduce subtyping are very common, and our framework supports subtyping directly; in addition to a set of qualifiers, users can define a partial order on the qualifiers.

*This research was supported in part by NSF CCR-9457812, NASA Contract No. NAG2-1210, NSF CCR-0085949, and DARPA Contract No. F33615-00-C-1693.

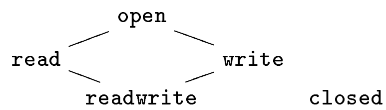


Figure 1: Subtyping relation among `File` qualifiers

Our results build on recent advances in flow-sensitive type systems [CWM99, SWM00, DF01] as well as our own previous work on flow-insensitive type qualifiers [FFA99]. The main contribution of our work is a practical, flow-sensitive type inference algorithm, in contrast to the type checking systems of [CWM99, SWM00, DF01].

Our flow-sensitive type inference algorithm is made practical by solving constraints lazily. As in any flow-sensitive analysis, explicitly forming a model of the store at every program point is prohibitively expensive for large code bases. By generating a linear-size constraint system from the original program and solving only the portion of the constraints needed to check qualifier annotations, our algorithm is able to scale to large examples.

Finally, our system is designed to be sound; we aim to prove the absence of bugs, not just to be heuristically good at finding bugs. For example, we believe that our system could be integrated into Java in a sound manner. We have shown soundness for `restrict` (Section 4), a key new construct in our system (see technical report [FA01]). Since the remainder of our system can be viewed as a simplification of [SWM00], we believe it is straightforward to prove soundness for our full type system using their techniques.

In Section 5 we report on experience with two applications, analyzing locking behavior in the Linux kernel and analyzing C stream library usage in application code. Our system found a number of new locking bugs, including some that extend across multiple functions or even, in one case, across multiple files.

1.1 System Architecture

Our flow-sensitive qualifier inference algorithm has several interlocking components. We first give an overview of the major pieces and how they fit together.

We expect programmers to interact with our type system, both when adding qualifier annotations and when reviewing the results of inference. Thus, we seek a system that supports efficient inference and is straightforward for a programmer to understand and use. Our type inference system integrates alias analysis, effect inference, and ideas from linear type systems.

- We use a flow-insensitive alias analysis to construct a model of the store. The alias analysis infers an *abstract location* for the result of each program expression; expressions that evaluate to the same abstract location may be aliased.

- We use effect inference [LG88] to calculate the set of abstract locations an expression e might use during e's evaluation. These effects are used in analyzing function calls and `restrict` (see below). Effect inference is done simultaneously with alias analysis.

- We model the state at a program point as an abstract *store*, which is a mapping from abstract locations to types. We can use the abstract locations from the flow-insensitive alias analysis because we allow only the type qualifiers, and not the underlying standard types, to change during execution. We represent abstract stores using a constraint formalism. Store constructors model allocations, updates, and function calls, and store constraints $C_1 \leq C_2$ model a branch from the program point represented by store C_1 to the program point represented by store C_2.

- We compute a linearity [SWM00] for each abstract location at each program point. Informally, an abstract location is *linear* if the type system can prove that it corresponds to a single concrete location in every execution; otherwise, it is non-linear. We perform *strong updates* [CWZ90] on locations that are linear and *weak updates* on locations that are non-linear. A strong update can change the qualifier on a location's type arbitrarily. Weak updates cannot change qualifiers. Computing linearities is important because most interesting flow-sensitive properties require strong updates.

- The system described so far has a serious practical weakness: Type inference may fail because a location on which a strong update is needed may be inferred to be non-linear. We address this with a new annotation `restrict`. The expression `restrict` x `=` e `in` e' introduces a new name x bound to the value of e. The name x is given a fresh abstract location, and among all aliases of e, only x and values derived from x may be used within e'. Thus the location of x may be linear, and hence may be strongly updated, even if the location of e is non-linear. We use effects to enforce the correctness of `restrict` expressions—soundness requires that the location of e does not appear in the effect of e'.
- We use effects to increase the precision of the analysis. If an expression e does not reference location ρ, which we can determine by examining the effect of e, then it cannot change the value stored at ρ, and the analysis of ρ can simply flow from the store preceding e to the one immediately after e without passing through e. If e is an application of a function called in many different contexts, then this idea makes e fully polymorphic in all the locations that e does not reference.

2 Related Work

We discuss three threads of related work: type systems, dataflow analysis, and tools for finding bugs in software.

Type Systems. Our type system is inspired by region and alias type checking systems designed for low-level programs [CWM99, SWM00, WM00]. Two recent language proposals, Vault [DF01] and Cyclone [GMW$^+$01], adapt similar ideas for checking high-level programs. Both of these languages are based on type checking and require programmers to annotate their programs with types. In contrast, we propose a simpler and less expressive monomorphic type system that is designed for efficient type inference. Our system incorporates effect inference [LG88, Wri92] to gain a measure of polymorphism.

The type state system of NIL [SY86] is one of the earliest to incorporate flow-sensitive type checking. Xu et al [XRM01] use a flow-sensitive analysis to check type safety of machine code. Type systems developed for Java byte code [SA98, O'C99] also incorporate flow-sensitivity to check for initialization before use and to allow reuse of the same local variable with different types.

Igarashi and Kobayashi [IK02] propose a general framework for resource usage analysis, which associates a trace with each object specifying valid accesses to the object, and checks that the program satisfies the trace specifications. They provide an inference algorithm, although it is unclear how efficient it is in practice since it invokes as a sub-step an unspecified algorithm to check that a trace set is valid.

Flanagan and Freund [FF00] use a type checking system to verify Java locking behavior. In Java locks are acquired and released according to a lexical discipline. To model locking in the Linux kernel (as in Section 5) we must allow non-lexically scoped lock acquires and releases.

The subset of our system consisting of alias analysis and effect inference can be seen as a monomorphic variant of region inference [TT94]. The improvements to region inference reported in [AFL95] are a much more expensive and precise method for computing linearities.

Dataflow Analysis. Although our type-based approach is related to dataflow analysis [ASU88], it differs from classical dataflow analysis in several ways. First, we generate constraints over stores and 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.

The strong/weak update distinction was first described by Chase et al [CWZ90]. Several techniques that allow strong updates have been proposed for dataflow-based analysis of programs with pointers, among them [EGH94, AL95, WL95]. Jagannathan et al [JTWW98] 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.

Bug-Finding Tools. 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 [Wei01].

Meta-level compilation [ECCH00] is a system for finding bugs in programs. The programmer specifies a flow-sensitive property as an finite state automaton. A program is analyzed by traversing 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. In [ECCH00] an intraprocedural analysis of lock usage in the Linux kernel uncovered many local locking bugs. Our type-based system found *inter*procedural locking bugs that extended across multiple functions or even, in one case, across multiple files (Section 5).[1] Newer work on meta-level compilation [ECH+01] includes some interprocedural dataflow, but it is unclear how their interprocedural dataflow analysis handles aliasing.

LCLint [Eva96] is a dataflow-based tool for checking properties of programs. To use LCLint, the programmer adds extra annotations to their program, just like our type qualifier system. LCLint performs flow-sensitive intraprocedural analysis, using the programmer's annotations at function calls.

3 Type System

We describe our type system using a call-by-value lambda calculus extended with pointers and type qualifier annotations. The source language is

$$e ::= x \mid n \mid \lambda x.e \mid e_1\ e_2 \mid \texttt{ref}\, e \mid \texttt{!}e \mid e_1 \texttt{:=} e_2 \mid \texttt{assert}(e,Q) \mid \texttt{check}(e,Q)$$

Here x is a variable, n is an integer, $\lambda x.e$ is a function with argument x and body e, the expression $e_1\ e_2$ is the application of function e_1 to argument e_2, the expression $\texttt{ref}\, e$ allocates memory and initializes it to e, the expression $\texttt{!}e$ dereferences pointer e, and the expression $e_1 \texttt{:=} e_2$ assigns the value of e_2 to the location e_1 points to.

We introduce qualifiers into the source language by adding two new forms [FFA99]. The expression $\texttt{assert}(e,Q)$ asserts that e's top-level qualifier is Q, and the expression $\texttt{check}(e,Q)$ type checks only if e's top-level qualifier is at most Q.

Our type inference algorithm is divided into two steps. First we perform an initial flow-insensitive alias analysis and effect inference. Second we generate and solve store and qualifier constraints and compute linearities.

3.1 Alias Analysis and Effect Inference

We present the flow-insensitive alias analysis and effect inference as a translation system rewriting source expressions to expressions decorated with locations, types, and effects. The target language is

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

The target language extends the source language syntax in two ways. Every allocation site $\texttt{ref}^\rho\, e$ is annotated with the abstract location ρ that is allocated, and each function $\lambda^L x{:}t.e$ is annotated with both the type t of its parameter and the effect L of calling the function. Effects are unions and intersections of *effect variables* ψ, which represent an unknown set of effects, and *effect constants* ρ, which stands for a read, write, or allocation of location ρ.

Foreshadowing flow-sensitive analysis, pointer types are written $\mathit{ref}(\rho)$, and we maintain a separate global abstract store C_I mapping locations ρ to types; $C_I(\rho) = \tau$ if location ρ contains data of type τ. If type inference requires $\rho = \rho'$, we also require $C_I(\rho) = C_I(\rho')$. Function types $t \longrightarrow^L t'$ contain the effect L of calling the function.

[1] The bugs were found in a newer version of the Linux kernel than examined by [ECCH00], so a direct comparison is not possible, though these bugs cannot be found by purely intraprocedural analysis.

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

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

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

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

$$\frac{\begin{array}{cc} \Gamma \vdash e_1 \Rightarrow e_1' : t_1; L_1 & \Gamma \vdash e_2 \Rightarrow e_2' : t_2; L_2 \\ t_1 = ref(\rho) \quad C_I(\rho) = t_2 & \rho \text{ fresh} \end{array}}{\Gamma \vdash e_1 \texttt{:=} e_2 \Rightarrow e_1' \texttt{:=} e_2' : t_2; L_1 \cup L_2 \cup \{\rho\}} \text{ (Assign)}$$

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

$$\frac{\begin{array}{cc} \Gamma \vdash e_1 \Rightarrow e_1' : t_1; L_1 & \Gamma \vdash e_2 \Rightarrow e_2' : t_2; L_2 \\ t_1 = t_2 \longrightarrow^{\psi} \beta & \psi, \beta \text{ fresh} \end{array}}{\Gamma \vdash e_1\ e_2 \Rightarrow e_1'\ e_2' : \beta; L_1 \cup L_2 \cup \psi} \text{ (App)}$$

$$\frac{\Gamma \vdash e \Rightarrow e' : t; L}{\Gamma \vdash \texttt{assert}(e, Q) \Rightarrow \texttt{assert}(e', Q) : t; L} \text{ (Assert)}$$

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

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

Figure 2: Type, alias, and effect inference

Figure 2 gives rules for performing alias analysis and effect inference while translating source programs into our target language. This translation system proves judgments $\Gamma \vdash e \Rightarrow e' : t; L$, meaning that in type environment Γ, expression e translates to expression e', which has type t, and the evaluation of e may have effect L.

The set of locations appearing in a type, $locs(t)$, is

$$\begin{array}{rcl} locs(int) & = & \emptyset \\ locs(ref(\rho)) & = & \{\rho\} \cup locs(C_I(\rho)) \\ locs(t_1 \longrightarrow^{L} t_2) & = & locs(t_1) \cup locs(t_2) \cup L \end{array}$$

We assume that $locs(\alpha)$ is empty until α is equated with a constructed type. We define $locs(\Gamma)$ to be $\bigcup_{x \mapsto t \in \Gamma} locs(t)$.

We briefly discuss the rules in Figure 2:

- (Var) and (Int) are standard. In lambda calculus, a variable is an r-value, not an l-value, and accessing a variable has no effect.
- (Ref) allocates a fresh abstract location ρ. We add the effect ρ of the allocation to the effect and record in C_I the type to which the location ρ points.
- (Deref) evaluates e, which yields a pointer to a location ρ. We look up the type of location ρ in C_I and add ρ to the effect set.

```
fun f w =                          fun^{ρz} f w: ref(ρz) =
   let x = ref 0                      let x = ref^{ρx} 0
       y = ref(assert(1, q_a))            y = ref^{ρy}(assert(1, q_a))
       z = ref(assert(2, q_b))            z = ref^{ρz}(assert(2, q_b))
   in                                 in
       /* Write to x's cell */
       x := 3                             x := 3
       w := 4                             w := 4
       y := assert(5, q_c)                y := assert(5, q_c)
       if (···)                           if (···)
         f z                                f z
       check(!y, q_c)                     check(!y, q_c)
```

(a) Source program (b) Target program

$$C_I(\rho_x) = C_I(\rho_y) = C_I(\rho_z) = int$$

Figure 3: Example alias and effect analysis

- (Assign) writes a location. Note that the type of e_2 and the type that e_1 points to are equated. Because types contain locations, this forces potentially aliased locations to be modeled by one abstract location.
- (Lam) defines a function. We annotate the function with the effect ψ of the function body and the type α of the parameter. Function types always have an effect variable ψ on the arrow, which makes effect inference easier. Notice that creating a function has no effect.
- (App) applies a function to an argument. The effect of applying e_1 to e_2 includes the effect ψ of calling the function e_1 represents. Notice that e_1's argument type is constrained to be equal to the type of e_2. As before, this forces possibly-aliased locations to have the same abstract location.
- (Assert) and (Check) are translated unchanged into the target language. Qualifiers are flow-sensitive, so we do not model them during this first, flow-insensitive step of the algorithm.
- (Down) hides effects on purely local state. If evaluating e produces an effect on some location ρ neither in Γ nor in t, then ρ cannot be accessed in subsequent computation. By intersecting the effects L with effects that may be visible $locs(\Gamma, t)$, we increase the precision of effect inference, which in turn increases the precision of flow-sensitive type qualifier inference. Although (Down) is not a syntactic rule, it only needs to be applied once per function body [FA01].

Figure 3 shows an example program and its translation. We use some syntactic sugar; all of these constructs can be encoded in our language (e.g., by assuming a primitive **Y** combinator of the appropriate type). In this example the constant qualifiers q_a, q_b, and q_c are in the discrete partial order (the qualifiers are incomparable). Just before f returns, we wish to check that y has the qualifier q_c. This check succeeds only if we can model the update to y as a strong update.

In Figure 3, we assign x, y, and z distinct locations ρ_x, ρ_y, and ρ_z, respectively. Because f is called with argument z, our alias analysis requires that the types of z and w match, and thus w is given the type $ref(\rho_z)$. Finally, notice that since x and y are purely local to the body of f, using the rule (Down) our analysis hides all effects on ρ_x and ρ_y. The effect of f contains ρ_z because f writes to its parameter w, which has type $ref(\rho_z)$.

Let n be the size of the input program. Applying the rules in Figure 2 generates a constraint system of size $O(n)$, using a suitable representation of $locs(\Gamma, t)$ (see [FA01]). Resolving the type equality constraints in the usual way with unification takes $O(n\alpha(n))$ time, where $\alpha(\cdot)$ is the inverse Ackerman's function. The remaining constraints are *effect constraints* of the form $L \subseteq \psi$. We solve these constraints on-demand—in the next step of the algorithm we will ask queries of the form $\rho \in L$. We can answer all such queries for a single location ρ in $O(n)$ time.

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

$$\frac{Q \leq Q'}{Q\ \mathit{ref}(\rho) \leq Q'\ \mathit{ref}(\rho)}\ (\mathrm{Ref}_{\leq})$$

$$\frac{\begin{array}{ccc} Q \leq Q' & \tau_2 \leq \tau_1 & \tau_1' \leq \tau_2' \\ C_2 \leq C_1 & & C_1' \leq C_2' \end{array}}{Q\ (C_1, \tau_1) \longrightarrow^L (C_1', \tau_1') \leq Q'\ (C_2, \tau_2) \longrightarrow^L (C_2', \tau_2')}\ (\mathrm{Fun}_{\leq})$$

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

Figure 4: Store compatibility rules

3.2 Stores and Qualified Types

Next we perform flow-sensitive analysis to check the qualifier-related annotations. In this second step of the algorithm we take as input a program decorated with types, locations, and effects by the inference algorithm of Figure 2. Throughout this step we treat the abstract locations ρ and effects L from the first step as constants. We analyze the input program using the extended types shown below:

$$\begin{array}{rcl} \tau & ::= & Q\ \sigma \\ Q & ::= & \kappa \mid B \\ \sigma & ::= & \alpha \mid \mathit{int} \mid \mathit{ref}(\rho) \mid (C, \tau) \longrightarrow^L (C', \tau') \\ C & ::= & \varepsilon \mid \mathit{Alloc}(C, \rho) \mid \mathit{Assign}(C, \rho : \tau) \mid \mathit{Merge}(C, C', L) \mid \mathit{Filter}(C, L) \\ \eta & ::= & 0 \mid 1 \mid \omega \end{array}$$

Here *qualified types* τ are standard types with qualifiers inserted at every level. Qualifiers Q are either *qualifier variables* κ, which stand for currently unknown qualifiers, or *constant qualifiers* B, specified by the user. We assume a supplied partial order $\leq$ among type qualifiers.

Flow-sensitive analysis associates a *store* C with each program point. This is in contrast to the flow-insensitive step, which uses one global store C_I to give types to locations. Function types are extended to $(C, \tau) \longrightarrow^L (C', \tau')$, where C describes the store the function is invoked in and C' describes the store when the function returns.

Each location in each store has an associated *linearity* η. There are three linearities: 0 for unallocated locations, 1 for linear locations (these admit strong updates), and ω for non-linear locations (which admit only weak updates). The three linearities form a lattice $0 < 1 < \omega$. Addition on linearities is as expected: $0 + x = x$, $1 + 1 = \omega$, and $\omega + x = \omega$.

Formally a store is a vector assigning a type and a linearity to every abstract location computed by the alias analysis:

$$\{\rho_1^{\eta_1}\!:\!\tau_1, \ldots, \rho_n^{\eta_n}\!:\!\tau_n\}$$

We call such a vector a *ground store*. If G is a ground store, we write $G(\rho)$ for ρ's type in G, and we write $G_{lin}(\rho)$ for ρ's linearity in G.

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 relate stores at consecutive program points either with *store constructors* (see below), which build new stores from old stores, or with *store constraints* $C_1 \leq C_2$, which are generated at branches from the program point represented by store C_1 to the program point represented by store C_2.

A *solution* to a system of store constraints is a mapping from store variables to ground stores. A solution S *satisfies* a system of store constraints if for each constraint $C_1 \leq C_2$ we have $S(C_1) \leq S(C_2)$ according to the rules in Figure 4.

In Figure 4, constraints between stores yield constraints between linearities and types, which in turn yield constraints between qualifiers and between stores. In our constraint resolution algorithm, we exploit the fact

$$
\begin{array}{rcl}
S(Alloc(C,\rho'))(\rho) & = & S(C)(\rho) \\
S(Merge(C,C',L))(\rho) & = & \begin{cases} S(C)(\rho) & \rho \in L \\ S(C')(\rho) & \text{otherwise} \end{cases} \\
S(Filter(C,L))(\rho) & = & S(C)(\rho) \qquad \rho \in L \\
S(Assign(C,\rho':\tau))(\rho) & = & \begin{cases} \tau & \rho = \rho' \\ S(C)(\rho) & \text{otherwise} \end{cases}
\end{array}
$$

(a) Types

$$
\begin{array}{rcl}
S(Alloc(C,\rho'))_{lin}(\rho) & = & \begin{cases} 1 + S(C)_{lin}(\rho) & \rho = \rho' \\ S(C)_{lin}(\rho) & \text{otherwise} \end{cases} \\
S(Merge(C,C',L))_{lin}(\rho) & = & \begin{cases} S(C)_{lin}(\rho) & \rho \in L \\ S(C')_{lin}(\rho) & \text{otherwise} \end{cases} \\
S(Filter(C,L))_{lin}(\rho) & = & \begin{cases} S(C)_{lin}(\rho) & \rho \in L \\ 0 & \text{otherwise} \end{cases} \\
S(Assign(C,\rho':\tau))_{lin}(\rho) & = & S(C)_{lin}(\rho)
\end{array}
$$

(b) Linearities

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

(c) Weak updates

Figure 5: Extending a solution to constructed stores

that we are only interested in qualifier relationships to solve as little of the expensive store constraints as possible.

In (Ref$_\leq$) we require that the locations on the left- and right-hand sides of the $\leq$ are the same. Alias analysis enforces this property, which corresponds to the standard requirement that subtyping becomes equality below a pointer constructor. We emphasize that in this step we treat abstract locations ρ as constants, and we will never attempt (or need) to unify two distinct locations to satisfy (Ref$_\leq$).

In (Fun$_\leq$) we require that the effects of the constrained function types match exactly.

Figure 5 formalizes the four kinds of store constructors by showing how a solution S mapping store variables to ground stores is extended to constructed stores.

The store $Alloc(C,\rho)$ is the same as store C, 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.

The store $Merge(C,C',L)$ combines stores C and C' according to effect L. If $\rho \in L$, then $Merge(C,C',L)$ assigns ρ the type it has in C, otherwise $Merge(C,C',L)$ assigns ρ the type it has in C'. The linearity definition is similar.

The store $Filter(C,L)$ assigns the same types and linearities as C for all locations ρ such that $\rho \in L$. The types of all other locations are undefined, and the linearities of all other locations are 0.

Finally, the store $Assign(C,\rho:\tau)$ is the same as store C, except location ρ is given type τ. If ρ is non-linear in C, then we require that τ be equal to the type of ρ in C; this corresponds to a weak update.

3.3 Flow-Sensitive Constraint Generation

Figure 6 gives the type inference rules for our system. In this system judgments have the form $\Gamma, C \vdash e : \tau, C'$, meaning that in type environment Γ and with initial store C, evaluating e yields a result of type τ and a new store C'. We write $C(\rho)$ for the type associated with ρ in store C; we discuss the computation of $C(\rho)$ in

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

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

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

$$\frac{\Gamma, C \vdash e : Q\ ref(\rho), C'}{\Gamma, C \vdash \texttt{!}e : C'(\rho), C'} \text{ (Deref)}$$

$$\frac{\Gamma, C \vdash e_1 : Q\ ref(\rho), C' \qquad \Gamma, C' \vdash e_2 : \tau, C''}{\Gamma, C \vdash e_1 \texttt{ := } e_2 : \tau, Assign(C'', \rho : \tau)} \text{ (Assign)}$$

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

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

$$\frac{\Gamma, C \vdash e : Q'\ \sigma, C'}{\Gamma, C \vdash \texttt{assert}(e, Q) : Q\ \sigma, C'} \text{ (Assert)}$$

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

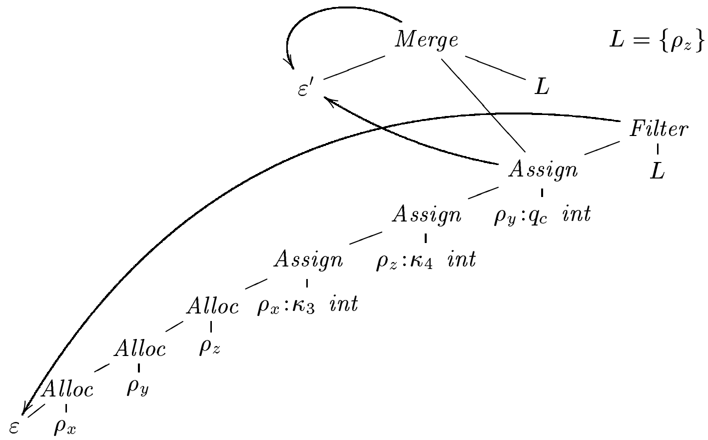
Figure 6: Constraint generation rules

Section 3.4. We use the function $sp(t)$ to decorate a standard type t with fresh qualifier and store variables:

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

We briefly discuss the rules in Figure 6

- (Var) and (Int) are standard. For (Int), we pick a fresh qualifier variable κ to annotate n's type.
- (Ref) adds a new location ρ to the store C', yielding the store $Alloc(C', \rho)$. The type τ of e is constrained to be compatible with ρ's type in C'.
- (Deref) looks up the type of e's location ρ in the current store C'. Any qualifier may appear on e's type; qualifiers are checked only by (Check), see below.
- (Assign) produces a new store representing the assignment of type τ to location ρ.
- (Lam) type checks function body e in fresh initial store ε and with parameter x bound to a type with fresh qualifier variables.
- (App) constrains $\tau_2 \leq \tau$ to ensure that e_2's type is compatible with e_1's argument type. The constraint $Filter(C'', L) \leq \varepsilon$ ensures that the current state of the locations that e_1 uses, which are captured by its effect set L, is compatible with the state e_1 expects. The final store $Merge(\varepsilon', C'', L)$ joins the store C'' before the function call with the result store ε' of the function. Intuitively, this rule gives us some low-cost polymorphism, in which functions do not act as join points for locations they do not use.



$$\begin{array}{rcl}
\kappa_0\ \ int & \leq & \varepsilon(\rho_x) \\
q_a\ \ int & \leq & Alloc(\varepsilon, \rho_x)(\rho_y) \\
q_b\ \ int & \leq & Alloc(Alloc(\varepsilon, \rho_x), \rho_y)(\rho_z) \\
\varepsilon'(\rho_y) & \leq & q_c\ \ int
\end{array}$$

Figure 7: Store constraints for example

- (Assert) adds a qualifier annotation to the program, and (Check) checks that the inferred top-level qualifier Q' of e is compatible with the expected qualifier Q.

Figure 7 shows the stores and store constraints generated for our example program. We have simplified the graph for clarity. Here ε is f's initial store and ε' is f's final store. We use undirected edges for store constructors and a directed edge from C_1 to C_2 for the constraint $C_1 \leq C_2$.

We step through constraint generation. The store $Alloc(\varepsilon, \rho_x)$ models the allocation of ρ_x. Location ρ_x is initialized to 0, which is given the type κ_0 *int* for fresh qualifier variable κ_0. (Ref) 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 κ_3 *int* and κ_4 *int*, respectively, where κ_3 and κ_4 are fresh qualifier variables.

For the recursive call to f, we construct a *Filter* and add an inclusion 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 q_c.

3.4 Flow-Sensitive Constraint Resolution

The rules of Figure 6 generate three kinds of constraints: qualifier constraints $Q \leq Q'$, subtyping constraints $\tau \leq \tau'$, and store constraints $C \leq \varepsilon$ (the right-hand side of a store constraint is always a store variable). A set of m type and qualifier constraints can be solved in $O(m)$ time using well-known techniques [FFA99, RM96], so in this section we focus on computing a solution S to a set of store constraints.

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 set of constructed stores and store constraints, we perform a least fixpoint computation to determine $S(C)_{lin}(\rho)$. We initially assume that in every store, location ρ has linearity 0. Then we exhaustively apply the rules in Figure 5(b) and the rule $S(\varepsilon)_{lin}(\rho) = (\max_{\{C \mid C \leq \varepsilon\}} S(C)_{lin}(\rho))$ until we reach a fixpoint. This last rule is derived from Figure 4.

In our implementation, we compute $S(C)_{lin}(\rho)$ in a single pass over the store constraints using Tarjan's strongly-connected components algorithm 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.

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

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

thereby assigning fresh qualifiers to the type of every location at every program point. Replace uses of $C(\rho)$ in Figure 6 with $S(C)(\rho)$, using the logic in Figure 5(a).

Apply the following two closure rules until no more constraints are generated:

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

Given a program of size n, in the worst case this naive algorithm requires at least n^2 space and time to build $S(\cdot)$ 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 $S(C)(\rho)$ for a constructed store C, we must deconstruct C recursively until we reach a variable store or an assignment to ρ (see Figure 5(a)). Because we represent the effect constraints compactly (in linear space), deconstructing $\mathit{Filter}(C, L)$ or $\mathit{Merge}(C, C', L)$ may require a potentially linear time computation to check whether $\rho \in L$. We recover efficient lookups by replacing C with a fresh store variable ε and adding the constraint $C \leq \varepsilon$. Then rather than computing $S(C)(\rho)$ we compute $S(\varepsilon)(\rho)$, which requires only a map lookup. Of course, we must use space to store ρ in $S(\varepsilon)$. However, as shown below, we often can avoid this cost completely. We apply this transformation to each store $\mathit{Merge}(C, C', L)$ constructed during constraint inference.

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

We can extend this idea further. For each qualifier variable κ, inference maintains a set of possible qualifier constants that are valid solutions for κ. If that set contains every constant qualifier, 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:

$$\begin{array}{c} C \leq \varepsilon \quad \Longrightarrow \quad S(C)(\rho) \leq S(\varepsilon)(\rho) \\ \qquad\qquad \text{for all } \rho \in S(C) \text{ or } S(\varepsilon) \text{ s.t.} \\ \qquad\qquad\qquad S(C)(\rho) \text{ or } S(\varepsilon)(\rho) \text{ interesting} \end{array}$$

$$\begin{array}{c} S(C)_{lin}(\rho) = \omega \quad \Longrightarrow \quad \tau = S(C)(\rho) \\ \qquad \text{for all } \mathit{Assign}(C, \rho : \tau) \text{ s.t. } \tau \text{ or } S(C)(\rho) \text{ interesting} \end{array}$$

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

Figure 8 gives an algorithm for lazy location propagation. We associate a mark with each ρ in each $S(\varepsilon)$ and with ρ in $\mathit{Assign}(C, \rho : \tau)$. Initially this mark is not set, indicating that location ρ is bound to an uninteresting type.

If a qualifier variable κ appears in $S(\varepsilon)(\rho)$, we associate the pair (ρ, C) with κ, and similarly for *Assign* stores. If during constraint resolution the set of possible solutions κ changes, we call PROPAGATE(ρ, C) 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 $S(C)(\rho)$ and $S(C')(\rho)$ for every store C' reachable from C. This step may add ρ to C' if C' is a store variable, and the type constraints BACK-PROP() and FORWARD-PROP() generate may trigger subsequent calls to PROPAGATE().

Consider again our running example. Figure 9 shows how locations and qualifiers propagate through the store constraint graph. Dotted edges in this graph indicate inferred constraints (discussed below). For clarity we have omitted the *Alloc* edges and the base types.

The four type constraints in Figure 7 are shown as directed edges in Figure 9. For example, the constraint $\kappa_0\ \mathit{int} \leq \varepsilon(\rho_x)$ 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 $\kappa_3\ \mathit{int}$ to ρ_x also does not cause any propagation.

The constraint $q_a\ \ \mathit{int} \leq \mathit{Alloc}(\varepsilon, \rho_x)(\rho_y)$ reduces to $q_a\ \ \mathit{int} \leq \varepsilon(\rho_y)$, which reduces to $q_a \leq \kappa_y$. This constraint does trigger propagation. PROPAGATE(ρ_y, ε) first pushes ρ_y 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 $\mathit{Assign}(\cdot, \rho_y : q_c\ \ \mathit{int})$; forward propagation assumes that this is a strong update.

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

The constraint $q_b \leq \kappa_z$ 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 $\mathit{Assign}(\cdot, \rho_z : \kappa_4\ \ \mathit{int})$ and add the constraint $\kappa_4 \leq \kappa_z$. This in turn triggers propagation from $\mathit{Assign}(\cdot, \rho_z : \kappa_4\ \ \mathit{int})$. This propagation step reaches ε' and 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 an equality constraint $\kappa_z = \kappa_4$, indicated with a double-dotted line.

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) rule in Figure 2 was 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 Restrict

As mentioned in the introduction, type inference may fail because a location on which a strong update is needed may be non-linear. In practice a major source of non-linear locations is data structures. For example, given a linked list `l`, our alias analysis cannot distinguish `l->lock` from `l->next->lock`, hence both are non-linear.

Our solution to this problem is to add a new form $\texttt{restrict}\, x \texttt{=} e_1\, \texttt{in}\, e_2$ to the language. Intuitively, this declares that of all aliases of e_1, only the particular value bound to x will be used within e_2. For example:

```
restrict x = y in
  x := ...; /* valid */
  y := ...; /* invalid */
```

The first assignment through `x` is valid, but the assignment through `y` is forbidden by `restrict`.

We check `restrict` using the following type rule, which is integrated into the first inference pass of Figure 2:

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

```
Propagate(ρ, ε) =
  case C of
    ε:
      add ρ : sp(C_I(ρ)) to S(ε) if not already in S(ε)
      if ρ is not marked in ε
        mark ρ in S(ε)
        Forward-prop(C, ρ, S(ε)(ρ))
        for each C' such that C' ≤ ε
          Back-prop(C', ρ, S(ε)(ρ))
    Assign(C', ρ : τ):
      if ρ is not marked in Assign(C', ρ : τ)
        mark ρ in Assign(C', ρ : τ)
        Forward-prop(C, ρ, τ)

Back-prop(C, ρ, τ) =
  case C of
    ε:
      add ρ : sp(C_I(ρ)) to S(ε) if not already in S(ε)
      S(ε)(ρ) ≤ τ
    Alloc(C', ρ'):
      Back-prop(C', ρ, τ)
    Merge(C', C'', L):
      if ρ ∈ L
        then Back-prop(C', ρ, τ)
        else Back-prop(C'', ρ, τ)
    Filter(C', L):
      if ρ ∈ L
        then Back-prop(C', ρ, τ)
    Assign(C', ρ' : τ'):
      if ρ = ρ'
        then τ' ≤ τ
        else Back-prop(C', ρ, τ)


Forward-prop(C, ρ, τ) =
  for each ε such that C ≤ ε
      add ρ : sp(C_I(ρ)) to S(ε) if not already in S(ε)
      τ ≤ S(ε)(ρ)
  for each C' such that C' is constructed from C
    case C' of
      Alloc(C, ρ'):
        Forward-prop(C', ρ, τ)
      Merge(C_1, C_2, L):
        if ρ ∈ L and C = C_1
          then Forward-prop(C', ρ, τ)
        if ρ ∉ L and C = C_2
          then Forward-prop(C', ρ, τ)
      Filter(C, L):
        if ρ ∈ L
          then Forward-prop(C', ρ, τ)
      Assign(C, ρ' : τ'):
        if ρ ≠ ρ'
          then Forward-prop(C', ρ, τ)
```

Figure 8: Lazy location constraint propagation

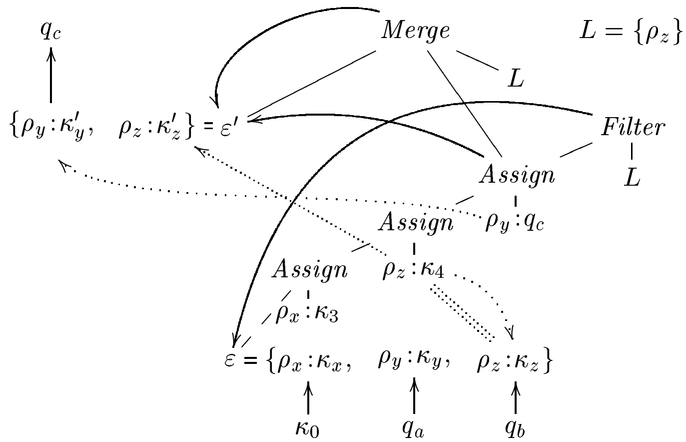


Figure 9: Constraint propagation

Here we bind x to a type with a fresh abstract location ρ' to distinguish dereferences of x from dereferences of other aliases of e_1. The constraint $\rho \notin L_2$ forbids location ρ from being dereferenced in e_2; notice dereferences of ρ' within e_2 are allowed. We require that ρ' not escape the scope of e_2 with $\rho' \notin \mathit{locs}(\Gamma, \tau, \tau_2)$, and we also add ρ to the effect set. We translate `restrict` into the target language by annotating it with the location ρ' that x is bound to. A full discussion of `restrict`, including a soundness proof, can be found in a technical report [FA01].

We use `restrict` to locally recover strong updates. The key observation is that the location ρ of e_1 and the location ρ' of x can be different. Thus even if the linearity of ρ is ω, the linearity of ρ' can be 1. Therefore within the body of e_2 we may be able to perform strong updates of ρ'. When the scope of `restrict` ends, we may need to do a weak update from ρ' to ρ.

For example, suppose that we wish to type check a state change of some lock deep within a data structure, and the location of the lock is non-linear. The following is not atypical of Linux kernel code:

```
spin_lock(&a->b[c].d->lock);    /* invalid; */
...                             /* non-linear loc */
spin_unlock(&a->b[c].d->lock);
```

Assuming the `...` above contains no accesses to aliases of the lock and does not alias the lock to a non-linear location, we can modify the code to type check as follows:

```
restrict lock = &a->b[c].d->lock in
  spin_lock(lock);              /* valid */
  ...
  spin_unlock(lock);
```

In our flow-sensitive step, we use the following inference rule for `restrict`:

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

In this rule, we infer a type for e_1, which is a pointer to some location ρ. Then we create a new store C'' in which the location ρ' of x is both allocated and initialized $C'(\rho)$. In C'', and with x added to the type environment, we evaluate e_2. Finally, the result store is the store C''' with a potentially weak update assigning the contents of ρ' to ρ.

$$
\begin{aligned}
S(\mathit{Merge}(C, C', L))(\rho) &= \begin{cases} S(C)(\rho) & \mathit{al}(\rho) \in L \vee \mathit{rw}(\rho) \in L \\ S(C')(\rho) & \text{otherwise} \end{cases} \\
S(\mathit{Filter}(C, L))(\rho) &= S(C)(\rho) \qquad \mathit{al}(\rho) \in L \vee \mathit{rw}(\rho) \in L
\end{aligned}
$$

$$
\begin{aligned}
S(\mathit{Merge}(C, C', L))_{\mathit{lin}}(\rho) &= \begin{cases} S(C)_{\mathit{lin}}(\rho) & \mathit{al}(\rho) \in L \\ S(C')_{\mathit{lin}}(\rho) & \text{otherwise} \end{cases} \\
S(\mathit{Filter}(C, L))_{\mathit{lin}}(\rho) &= \begin{cases} S(C)_{\mathit{lin}}(\rho) & \mathit{al}(\rho) \in L \vee \mathit{rw}(\rho) \in L \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 10: New definition of S with allocation and read-write effects

5 Experiments

We have used a prototype implementation of our analysis to check two program properties: locking in the 2.4.9 Linux kernel device drivers and uses of the C Stream Library. Our implementation is sound up to the unsafe features of C: type casts, variable-argument functions, ill-defined pointer arithmetic, and conversions from arbitrary integers to pointers. We currently make no attempt to track the effect of any of these features on aliasing, except for the special case of type casting of the result of `malloc`-like functions. In combination with a system for enforcing memory safety, such as CCured [NMW02], our implementation would be sound.

In our implementation, we do not allow strong updates on locations containing functions. This improves efficiency because we never need to recompute $S(C)_{\mathit{lin}}(\rho)$—weak updates will not add constraints between stores.

Additionally, observe that allocations affect linearities but not types, and reads and writes affect types but not linearities. Thus in our implementation we also improve the precision of the analysis by distinguishing read-write and allocation effects. Formally, instead of effects of the form ρ, we introduce effects $\mathit{rw}(\rho)$ for a read or write of location ρ, and $\mathit{al}(\rho)$ for an allocation of location ρ. We modify Figure 2 so that (Ref) yields effect $\mathit{al}(\rho)$ and (Deref) and (Assign) yield effect $\mathit{rw}(\rho)$.

Then we modify the definition of S for *Merge* and *Filter* as shown in Figure 10. The first, second, and last case are as before. In the third case, we use only the allocation effects of L when computing the linearity of a location in a *Merge* store. Intuitively this means that functions that do not allocate a location ρ do not act as join points for location ρ with respect to linearities. We could also improve the precision of the analysis further by distinguish read and write effects from each other.

5.1 Linux Kernel Locking

The Linux kernel includes two primitive locking functions, which are used extensively by device drivers:

```
void spin_lock(spinlock_t *lock);
void spin_unlock(spinlock_t *lock);
```

We use three qualifiers `locked`, `unlocked`, and $\top$ (unknown) to check locking behavior. The subtyping relation is `locked` $< \top$ and `unlocked` $< \top$. We assign `spin_lock` the type

$$
\begin{gathered}
(C, \mathit{ref}(\rho)) \longrightarrow^{\{\rho\}} (\mathit{Assign}(C, \rho : \texttt{locked spinlock\_t}), \texttt{void}) \\
\text{where} \\
C(\rho) \leq \texttt{unlocked spinlock\_t}
\end{gathered}
$$

We omit the function qualifier since it is irrelevant. The type of `spin_lock` requires that the lock passed as the argument be unlocked (see the where clause) and changes it to locked upon returning. The signature for `spin_unlock` is the same with `locked` and `unlocked` exchanged. Since our implementation currently lacks parametric polymorphism, we inline calls to `spin_lock` and `spin_unlock`.

Using these type signatures we can check for two kinds of errors: deadlocks from acquiring a lock already held by the same thread, and attempting to acquire or release a lock in an unknown ($\top$) state.

We analyzed 513 whole device driver modules (a whole module includes all the files that make up a single driver). A module must meet a well-specified kernel interface, which we model with a `main` function that non-deterministically calls all possible driver functions registered with the kernel.

We have not yet finished reviewing the analysis results for all modules. So far we have found 14 apparently new locking bugs, including one which spanned multiple files. Five of the apparent bugs involve deadlocks, in which a function tries to acquire a lock already held by a function above it in the call chain. For example, the emu10k1 module contains a deadlock: `void`):

```
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, ...); ...
}
void emu10k1_set_volume_gpr(struct emu10k1_card *card, ...) {
    struct patch_manager *mgr = &card->mgr;
    ... spin_lock_irqsave(&mgr->lock, flags); ...
}
```

Note detecting this error requires interprocedural analysis.

One of our goals is to understand how often, and why, our system fails to type check real programs. We have categorized every type error in an earlier experiment where we separately analyze each of 910 driver files and remove the $\top$ qualifier so that `locked` and `unlocked` are incomparable. In this experiment, of the 52 files that fail to type check, 11 files have locking bugs and the remaining 41 files have type errors. Half of these type errors are due to incorrect assumptions eliminated by moving to whole module analysis, and the remaining type errors fall into two main categories.

In most cases the problem is that our alias analysis is not strong enough to type check the program, often because our current implementation does not have parametric polymorphism for store locations. We plan to add this feature using the techniques of [FRD00, RF01]. In another common situation there are multiple aliases of a location, but only one alias is actually used in the code of interest; we can type check this pattern using `restrict`. Not surprisingly, larger programs have more problems with spurious aliasing, so we believe both polymorphism and `restrict` are most important for large programs.

A less 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 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 (since we did not use $\top$ in these single file experiments). Most 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.

Even after further improvements, we expect some dynamically correct programs will not type check. As future work, we propose the following solution. The qualifier $\top$ represents an unknown state. We can use the information in the constraints to automatically insert coercions to and from $\top$ where needed. During execution these coercions perform runtime tests to verify locks are in the correct state. Thus, our approach can introduce dynamic type checking in situations where we cannot prove safety statically.

We added `restrict` annotations to the emu10k1 module, which is the Linux kernel module that yielded the largest number of false positives because non-linear locations could not be strongly updated. Using `restrict`, we eliminated all of these false positives. This supports our belief that `restrict` is the right tool for dealing with (necessarily) conservative alias analysis. Many of these `restrict` annotations were needed because of the current lack of location polymorphism; we must leave an accurate assessment of how burdensome `restrict` is to future work.

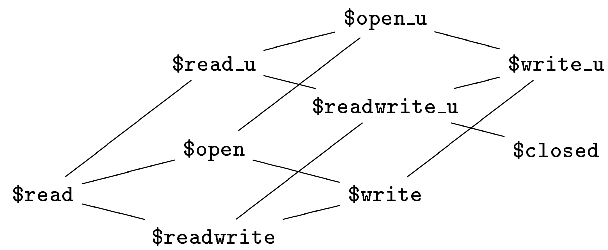


Figure 11: Subtyping relation among C stream library qualifiers

5.2 C Stream Library

As mentioned in the introduction, the C stream library interface contains certain sequencing constraints. For example, a file must be opened for reading before being read. A special property of the C stream library is that the result of `fopen` must be tested against `NULL` before being used, because `fopen` may or may not succeed.

We model C stream library file states using the qualifier partial order given in Figure 11. This partial order extends the partial order in Figure 1 with four additional qualifiers `$open_u`, `$read_u`, `$write_u`, and `$readwrite_u`. The qualifier `$X_u` stands for a file opened in state `X` that has not yet been checked against `NULL`.

The type signature for `fclose` is

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

and the type signature for `fopen` is

$$(C, \mathit{mode}) \longrightarrow^{\{\rho\}} (\mathit{Assign}(\mathit{Alloc}(C, \rho), \rho : \mathit{mode}\,\texttt{FILE}), \mathit{ref}(\rho))$$
$$\text{where}$$
$$C(\rho) \leq \texttt{\$closed}\,\texttt{FILE}$$

The mode is passed as a parameter to the `fopen` function. In practice the mode is usually a constant string, and therefore we can determine the correct mode qualifier, `$read_u`, `$write_u`, or `$readwrite_u`, by a simple syntactic comparison against possible mode strings. If we cannot determine the mode qualifier syntactically, we issue a warning and mark the file as `$open_u`.

Finally, functions that read and write files require appropriate qualifiers for their file arguments. For example, the `fgetc` function, which reads a character from a stream, has the signature

$$(C, \mathit{ref}(\rho)) \longrightarrow^{\{\rho\}} (C, \mathit{int})$$
$$\text{where}$$
$$C(\rho) \leq \texttt{\$read}\,\texttt{FILE}$$

Using these qualifiers we can type check the following C code fragment:

```
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 read. Thus the location ρ corresponding to the opened file is given the type `$read_u FILE`. We treat

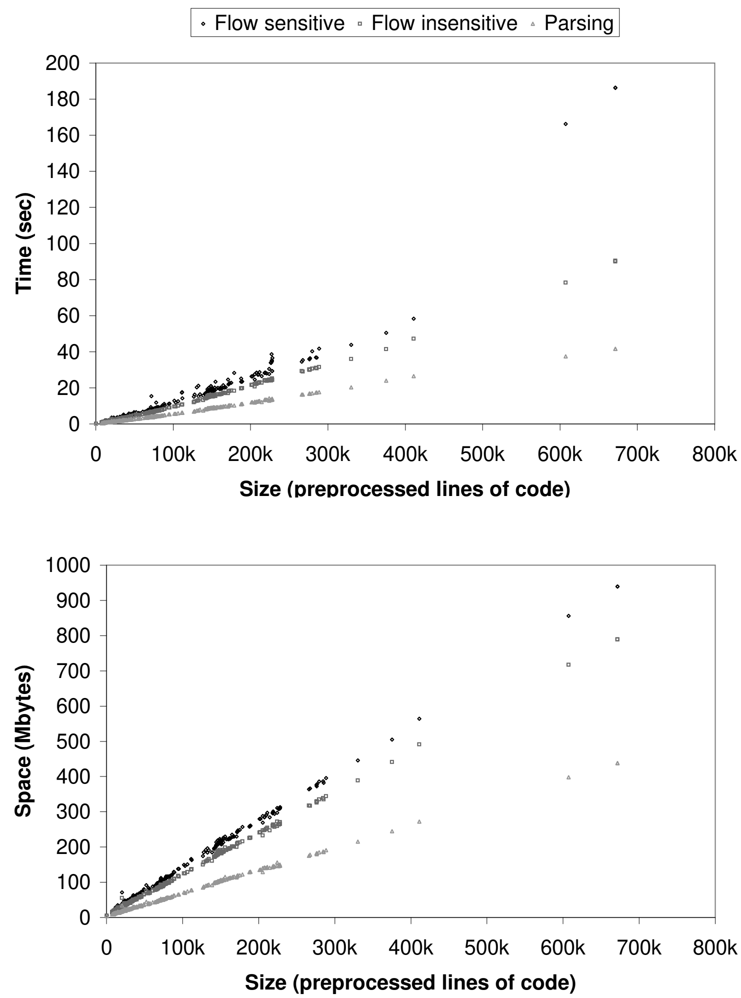


Figure 12: Resource usage for whole module analysis

the comparison between the pointer to location ρ and `NULL` as a kind of type case. We analyze the true branch starting in the store $Assign(C, \rho :$ `$read FILE`$)$, and we analyze the false branch starting in the store $Assign(C, \rho :$ `$closed FILE`$)$. We use conditional constraints to relate the `$read_u` qualifier in C to `$read` at the true branch.

The class of C stream library usage errors our tool can detect includes files used without having been opened and checked against `NULL`, files opened in an incompatible mode, and files accessed after being closed.

We tried our tool on 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 (see below), as we did not expect to find any latent stream library usage bugs in such mature programs. However, we did find one minor bug in `sendmail`, in which an opened log file is never closed in some circumstances.

5.3 Precision and Efficiency

The algorithm described in Section 3.4 is carefully designed to limit resource usage. Figure 12 shows time and space usage of whole module analysis versus preprocessed lines of code for 513 Linux kernel modules. All experiments were done on a dual processor 550 MHz Pentium III with 2GB of memory running RedHat 6.2.

We divide the resource usage into C parsing and type checking, flow-insensitive analysis, and flow-sensitive analysis. Flow-insensitive analysis consists of the alias and effect inference of Figure 2 together with flow-insensitive qualifier inference [FFA99]. Flow-sensitive analysis consists of the constraint generation and resolution described in Sections 3.3-3.4, including the linearity computation.

The graphs show 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 30% 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 2.3 of the flow-insensitive running times.

The analysis of `sendmail-8.11.6`, with 175,493 preprocessed source lines, took 168 seconds and 266MB; `man-1.5h1`, with 16,411 preprocessed source lines, took 1.99 seconds and 32MB. The time usage for sendmail suggests that C stream library analysis is more expensive than Linux kernel locking analysis. The higher running time is most likely because sendmail uses stream operations more often and more freely than a typical Linux kernel module uses spin locks. Because our algorithm is demand driven, more demand means more computation.

6 Conclusion

We have presented a system for extending standard type systems with flow-sensitive type qualifiers. We have given a lazy constraint resolution algorithm to infer type qualifier annotations and have shown that our analysis is effective in practice by finding a number of new locking bugs in the Linux kernel.

References

[AFL95] Alexander Aiken, Manuel Fähndrich, and Raph Levien. Better Static Memory Management: Improving Region-Based Analysis of Higher-Order Languages. In *Proceedings of the 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation*, La Jolla, California, June 1995, pages 174–185.

[AL95] 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.

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

[CWM99] 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*, San Antonio, Texas, January 1999, pages 262–275.

[CWZ90] 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.

[DF01] 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.

[ECCH00] 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.

[ECH+01] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Banff, Canada, October 2001.

[EGH94] 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.

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

[FA01] Jeffrey S. Foster and Alex Aiken. Checking Programmer-Specified Non-Aliasing. Technical Report UCB//CSD-01-1160, University of California, Berkeley, October 2001.

[FF00] Cormac Flanagan and Stephen N. Freund. Type-Based Race Detection for Java. *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Vancouver B.C., Canada, June 2000, pages 219–232.

[FFA99] 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.

[FRD00] 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*, Vancouver B.C., Canada, June 2000, pages 253–263.

[GMW+01] 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. Current version at `http://www.cs.cornell.edu/projects/cyclone`.

[IK02] Atsushi Igarashi and Naoki Kobayashi. Resource Usage Analysis. To appear in *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Portland, Oregon, January 2002.

[JTWW98] 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*, San Diego, California, January 1998, pages 329–341.

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

[NMW02] George Necula, Scott McPeak, and Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Code. To appear in *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Portland, Oregon, January 2002.

[O'C99] 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*, San Antonio, Texas, January 1999, pages 70–78.

[RF01] 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.

[RM96] 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.

[SA98] 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*, San Diego, California, January 1998, pages 149–160.

[SWM00] 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.

[SY86] 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.

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

[Wei01] Daniel Weise, 2001. Personal communication.

[WL95] 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*, La Jolla, California, June 1995, pages 1–12.

[WM00] David Walker and Greg Morrisett. Alias Types for Recursive Data Structures. In *International Workshop on Types in Compilation*, Montreal, Canada, September 2000.

[Wri92] 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.

[XRM01] 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.