Checking and Inferring Local Non-Aliasing[*]

Alex Aiken
University of California, Berkeley
aiken@cs.berkeley.edu

Jeffrey S. Foster
University of Maryland, College Park
jfoster@cs.umd.edu

John Kodumal
University of California, Berkeley
jkodumal@cs.berkeley.edu

Tachio Terauchi
University of California, Berkeley
tachio@cs.berkeley.edu

ABSTRACT

In prior work [15] we studied a language construct `restrict` that allows programmers to specify that certain pointers are not aliased to other pointers used within a lexical scope. Among other applications, programming with these constructs helps program analysis tools locally recover strong updates, which can improve the tracking of state in flow-sensitive analyses. In this paper we continue the study of `restrict` and introduce the construct `confine`. We present a type and effect system for checking the correctness of these annotations, and we develop efficient constraint-based algorithms implementing these type checking systems. To make it easier to use `restrict` and `confine` in practice, we show how to automatically infer such annotations without programmer assistance. In experiments on locking in 589 Linux device drivers, `confine` inference can automatically recover strong updates to eliminate 95% of the type errors resulting from weak updates.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; D.3.3 [**Programming Languages**]: Language Constructs and Features; D.2.4 [**Software Engineering**]: Software/Program Verification; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs

General Terms

Algorithms, Design, Reliability, Experimentation, Languages, Theory, Verification

Keywords

Restrict, confine, types, type qualifiers, alias analysis, effect inference, flow-sensitivity, constraints, locking, Linux kernel

[*]This research was supported in part by NSF CCR-0085949 and DARPA Contract No. F33615-00-C-1693.

1. INTRODUCTION

Almost all program analyses for languages with pointers must perform some form of *alias analysis*: when a program indirectly loads or stores through a pointer p, the analysis must determine to which location(s) p points. Alias analysis is a key ingredient in many program checking systems and compiler optimizations. The research literature abounds with proposed alias analysis techniques [1, 7, 10, 18, 20, 21, 24, 26, 30] (to name only a few), some of which scale to very large programs. Almost all of these techniques are fully automatic. That is, such an analysis takes a bare program and infers all possible aliasing.

This paper is about aliasing in programs, but the purpose is different from previous work on automatic alias analysis. Our motivation comes from experience developing and using CQUAL, a system for extending C with user-defined type qualifiers [15]. Consider the partial program shown in Figure 1. We use this example to tell a story, the moral of which is that we needed a new form of alias analysis to make CQUAL and similar analyses work in practice; that new form of alias analysis, previously sketched briefly [15] and developed further here, is the topic of this paper.

One application of CQUAL is to verify properties of locking. CQUAL uses two non-standard, flow-sensitive (see below) type qualifiers `locked` and `unlocked` to refine the type `lock`. If all goes well in the example in Figure 1, CQUAL infers that `*l` (the lock that `l` points to) has type `unlocked lock` at point 1 (i.e., the lock is not held), the type `locked lock` at point 2 (i.e., the lock is held), and the type `unlocked lock` at point 3.[1] In this way, CQUAL checks code for deadlocks caused by reacquiring a lock that is already held or releasing a lock that has not been acquired.

CQUAL models state by mapping every program variable v (or other concrete memory location) to an abstract location ρ. If two program quantities may alias each other (according to a particular alias analysis), they are mapped to the same abstract location. In the example, because our alias analysis cannot distinguish different elements of an array, all elements of the array reside at the same abstract location ρ. Similarly, `l` points to location ρ, meaning that `*l` is stored at location ρ, and thus both `*l` and all array elements may alias.

To analyze locking CQUAL performs a *flow-sensitive* analysis, which means that CQUAL must be able to assign `*l` different types at different points in the program. Assume that all locks in the array begin in the state `unlocked lock`. The call to `spin_lock(l)` changes the state of `*l` to a `locked lock`. However, it is not `*l`'s state that is changed, but the state of `*l`'s abstract location

[1]To accomplish this, CQUAL also needs to know how the functions `spin_lock` and `spin_unlock` change the state of locks. This information is given as type signatures [15].

```
void foo(int i) {
    do_with_lock(&locks[i]);
}

void do_with_lock(lock *l) { /* 1 */
    spin_lock(l);            /* 2 */
    work();
    spin_unlock(l);          /* 3 */
}
```

Figure 1: Example program

ρ. But ρ stands for other locks, too—namely the other locks in the array, which are still in the `unlocked` state. Thus after the call to `spin_lock(l)` the static information about ρ degrades to knowing only that any locks it represents may be either in the `locked` or the `unlocked` state, and the verification of any locking properties on any of these locks becomes impossible.

The difficulty is that the single abstract location ρ stands for multiple concrete locks, and the call to `spin_lock(l)` only changes the state of a single lock. Thus the information about ρ after the call to `spin_lock(l)` is the union of the old state (for the locks that did not change) and the new state (for the one lock that did change). In flow-sensitive analysis, this is known as a *weak update*. What we need for accurate analysis, though, is a *strong update*: we want to change the state of `*l` from `unlocked` to `locked` and not affect the status of any other lock. The need to perform strong updates is not specific to locks. This problem arises in any static analysis where there are both collections such as arrays or lists and we want to track state changes of values.

If we knew that `do_with_lock` could only access `*l` through its formal parameter `l`, and not through some alias it holds through, e.g., a global variable, then locally within `do_with_lock` we could ignore the aliases of `*l` external to `do_with_lock` and perform strong updates on `*l`'s location [15].

The recent C99 standard for the C programming language [2] provides a way to say almost exactly this. Change the definition of `do_with_lock` to

```
void do_with_lock(lock *restrict l)
```

At a high level, the `restrict` keyword means that no alias of `*l` defined outside of `do_with_lock` is used during the function's execution. Although there may be many aliases of `l` in the program, locally we know `l` is the only way for `do_with_lock` to access `*l`. This notion of locally unaliased pointers is missing from conventional flow-insensitive may-alias analysis, where pointers are either aliased or not and the only scope of interest is the entire program. Notice that while context-sensitive [24] or parameterized [21] alias analysis may help our `do_with_lock` example, we can also use restrict to indicate local non-aliasing within nested scopes smaller than function scopes. Indeed, we make use of this feature in our experiments (Section 7).

Another key feature of `restrict` is that it provides a form of program documentation: it allows the programmer to specify a particular kind of non-aliasing. Combined with a checking system such as we propose, we believe that `restrict` is not only beneficial for tools like CQUAL, but also for the programmer when writing their program.

In C, `restrict` is trusted and unchecked by the compiler—it amounts to a license for compilers to perform aggressive optimizations that would be unsound in the presence of aliases. We believe `restrict` is even more useful in program checking tools, and not just for C programs, but for programs written in any language with references. While there are important exceptions, such as functions that copy data or pointers, we believe many pointers in practice can be marked `restrict`ed.

The thesis of this paper is that `restrict`ed references are common in real programs, and that exploiting this (usually implicit) structure is important to software engineering tools such as CQUAL that need to reason about references. More specifically, the contributions of this paper are:

- We develop a formal semantics of `restrict` (Section 3.2) and also present an informal description and examples (Section 2).

- We give a *type and effect system* [22] for checking that a `restrict`-annotated program is correct with respect to our semantics (Section 3).

- We give a $\mathcal{O}(kn)$ constraint-based algorithm for verifying restrict annotations, where n is the size of the typed program and k is the number of `restrict` annotations in the program. The type system for restrict is described briefly in prior work [15], but this is the first description of the type checking algorithm (Section 4).

- In using CQUAL we have found it necessary to add many more `restrict` annotations to programs than we would like to do by hand. This motivates the idea of *restrict inference*: not just checking user-supplied `restrict` annotations, but automatically inferring `restrict`s in a program with no `restrict` annotations. We give an $\mathcal{O}(n^2)$ algorithm for `restrict` inference (Section 5).

- Furthermore, in many applications we wish to `restrict` not just a variable, but an expression. This extension of `restrict` introduces two new problems. First, to treat an arbitrary expression as a name, it must be referentially transparent, which introduces additional constraints beyond what is required for `restrict`. We call this stronger condition *confining* an expression and likewise name the associated construct `confine`. Second, for `confine` inference we have the additional problem of inferring in what scope an expression can be confined (Section 6).

- We present the results of experiments with `confine` inference, in which we use CQUAL to analyze the locking behavior of 589 Linux device drivers. In this experiment, `confine` is very effective at identifying the program points where strong updates can aid the analysis (Section 7).

2. RESTRICT

This section gives an informal semantics of `restrict` and several examples. Section 3.2 sketches a precise, formal semantics of `restrict`. Let `p` be a pointer declared as

```
int *restrict p;
```

meaning that `p` is a restricted pointer to an integer, and suppose `p` points to object `X`. The simplest use of `restrict` is to bind a new name `p` for all accesses of `X` in a local scope. Here an access within a scope is either a direct access or an access that occurs during the execution of a function called within that scope. The following example demonstrates valid and invalid pointer dereferences within the scope of a `restrict`:

```
{ int *restrict p = q;
  *p;  // valid
  *q;  // invalid
  *a;  // invalid if a and q may alias
}
```

Here `p` is initialized to `q`, and we attempt to dereference `p`, `q`, and `a` within the scope of `p`. Since `p` is annotated with `restrict`, we may dereference `p` but we may not dereference `q` or `a` (if `a` aliases `q`). In other words, within the scope of the `restrict`, the name `p` (and copies derived from `p`; see below) must be the sole access to the location `p` points to.

As another example, the following code shows that `restrict`-qualified pointers may be re-bound in an inner scope:

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

In our version of `restrict` (which differs from C99 [2] on a few points; see below), it is legal to create and use some aliases of restricted pointers. Consider the following example:

```
int *x;
{ int *restrict p = ...;
  int *r = p;
  *r;    // valid: use of local copy
  x = p; // invalid: copy escapes
}
```

Here we are allowed to make copies of the restricted pointer `p`, which we can also dereference inside of the `restrict`.

Intuitively, annotating the definition of `p` with `restrict` splits the aliases of `p` into two groups:

- Aliases of `p` created outside scope of the `restrict` may be accessed outside the `restrict` but not inside.
- `p` and aliases of `p` created inside the scope of the `restrict` may be accessed within the scope of the `restrict` but not outside.

In most uses of `restrict`, only the restricted pointer itself is used to reference storage it points to inside the `restrict` construct. In this common case, `restrict` serves to create a local pointer that is known to be the sole access to its storage in a particular scope. However, there is no difficulty in supporting copies of `restrict`ed pointers as in the example above. The requirement that aliases created inside the `restrict` not be used outside of the `restrict` means that we must check that no aliases of the restricted pointer *escape* the scope of the `restrict`. Thus the assignment to `x` is illegal in the example above and would be flagged as an error by our system. References can also escape by being stored into the heap or global variables; our system disallows such operations on restricted pointers. While we have found that preventing restricted pointers from escaping sufficient so far, one can imagine applications where allowing restricted pointers to leave their original scope of definition might be useful. We plan to consider such an extension as future work.

As mentioned in the introduction, our version of `restrict` is inspired by the ANSI C keyword of the same name [2]. The major difference between our version of `restrict` and ANSI C's is that in ANSI C `restrict` is not checked—the programmer is assumed to have added the `restrict` qualifier correctly. Another difference is that in ANSI C, a `restrict` annotation on a pointer `p` is ignored if the object pointed to by `p` is not written within the scope of the `restrict`. For a full discussion, see [13, 14].

3. LANGUAGE AND TYPE CHECKING

We present our type system for `restrict` using a small imperative language:

e	$::=$	x	Variable
	$\mid$	n	Integer
	$\mid$	$\texttt{new}\ e$	Allocate memory initialized to e
	$\mid$	$\texttt{*}\ e$	Dereference pointer e
	$\mid$	$e_1 \texttt{:=} e_2$	Assign e_2 to e_1
	$\mid$	$\texttt{let}\ x \texttt{=} e_1\ \texttt{in}\ e_2$	
			Bind e_1 to name x in e_2
	$\mid$	$\texttt{restrict}\ x \texttt{=} e_1\ \texttt{in}\ e_2$	
			Restrict e_1 to name x in e_2

For simplicity, we have omitted function definitions and calls from the language. The treatment of functions is standard and introduces no new issues; we omit it for brevity. Statement sequencing $e_1; e_2$ is also not present in the language, but is easily added with no complications. A discussion of the language extended with functions, as well as a detailed proof of soundness, can be found elsewhere [13, 14].

Besides variables, integers, pointer allocation `new` e, dereference `*` e, and assignment, our language has two mechanisms for introducing local variables. The first is $\texttt{let}\ x \texttt{=} e_1\ \texttt{in}\ e_2$, which simply initializes a new pointer variable x to e_1 for use in e_2. There is also a new scoping construct

$$\texttt{restrict}\ x \texttt{=} e_1\ \texttt{in}\ e_2$$

with the following meaning: like `let`, the pointer x is initialized to e_1 and bound within the body e_2. However, unlike `let`, within e_2 the only access to the location x points to is through x or values derived from x.

To enforce the semantics of `restrict`, our type system needs two extensions of standard types. First, we need a way to keep track of program names that may be aliased to one another. We use the standard solution, which is to associate abstract memory locations ρ with pointer types. Names that have the same abstract location in their types may be aliased to the same concrete memory location. The grammar for types is:

$$\tau \quad ::= \quad \mathit{int} \mid \mathit{ref}^{\,\rho}(\tau)$$

Pointer types are $\mathit{ref}^{\,\rho}(\tau)$, meaning a pointer to a value of type τ where ρ is the abstract location pointed to.

Second, we need to enforce the rule that a location ρ_0 may not be accessed within the body e of a `restrict`. To accomplish this, we calculate the set of abstract locations L_e that e may read or write and check $\rho_0 \notin L_e$. The set L_e is called the *effect* of e [16]. The grammar for effects is:

$$L \quad ::= \quad \emptyset \mid \{\rho\} \mid L_1 \cup L_2 \mid L_1 \cap L_2$$

We need one auxiliary function for our type checking system. We write $\mathit{locs}(\tau)$ for the set of locations occurring in the type τ, defined as

$$\begin{array}{rcl} \mathit{locs}(\mathit{int}) & = & \emptyset \\ \mathit{locs}(\mathit{ref}^{\,\rho}(\tau)) & = & \{\rho\} \cup \mathit{locs}(\tau) \end{array}$$

Our type system proves judgments of the form

$$\Gamma \vdash e : \tau; L$$

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

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

$$\frac{\Gamma \vdash e : \tau; L}{\Gamma \vdash \texttt{new}\, e : \mathit{ref}^{\rho}(\tau); L \cup \{\rho\}} \text{(Ref)}$$

$$\frac{\Gamma \vdash e : \mathit{ref}^{\rho}(\tau); L}{\Gamma \vdash \texttt{*}\, e : \tau; L \cup \{\rho\}} \text{(Deref)}$$

$$\frac{\Gamma \vdash e_1 : \mathit{ref}^{\rho}(\tau); L_1 \qquad \Gamma \vdash e_2 : \tau; L_2}{\Gamma \vdash e_1 \texttt{:=} e_2 : \tau; L_1 \cup L_2 \cup \{\rho\}} \text{(Assign)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \mathit{ref}^{\rho}(\tau_1); L_1 \\ \Gamma[x \mapsto \mathit{ref}^{\rho}(\tau_1)] \vdash e_2 : \tau_2; L_2\end{array}}{\Gamma \vdash \texttt{let}\, x \texttt{=} e_1 \,\texttt{in}\, e_2 : \tau_2; L_1 \cup L_2} \text{(Let)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \mathit{ref}^{\rho}(\tau_1); L_1 \\ \Gamma[x \mapsto \mathit{ref}^{\rho'}(\tau_1)] \vdash e_2 : \tau_2; L_2 \\ \rho \notin L_2 \qquad \rho' \notin \mathit{locs}(\Gamma, \tau_1, \tau_2)\end{array}}{\Gamma \vdash \texttt{restrict}\, x \texttt{=} e_1 \,\texttt{in}\, e_2 : \tau_2; L_1 \cup L_2 \cup \{\rho\}} \text{(Restrict)}$$

Figure 2: Type Checking Rules

meaning that expression e has type τ in *type environment* Γ (a mapping of variables to types), and the evaluation of e may read or write the locations in L. We define *locs*(Γ) as $\bigcup_{x:\tau \in \Gamma} \mathit{locs}(\tau)$.

Figure 2 gives the type checking rules for our language. We briefly discuss the rules, which except for `restrict` are all typical of effect systems.

- (Var) looks up the type of a variable in the type environment Γ. Looking up a pointer variable does not dereference that variable and thus has no effect.
- (Int) says an integer constant has type *int*. There is no effect.
- (Ref) constructs a pointer type; there is an effect on the allocated location.
- (Deref) deconstructs a pointer type. Since operationally a dereference reads a location, we add ρ, the abstract location pointed to by e, to the effect set.
- (Assign) updates a location. As with (Deref), we add ρ to the effect set, since the assignment updates e_1. Notice we require that the type of e_2 and the type pointed to by e_1 match. Since those types may themselves contain abstract locations, this rule encodes a unification-based may-alias analysis Steensgaard [26].
- (Let) does two things. First, `let` introduces a local variable `x`. The type of `x` is required to be a pointer. This restriction just makes `let` parallel with `restrict` in our small language (`restrict` only makes sense for pointers). Second, `let` evaluates both e_1 and e_2; note that the effect of the `let` is the union of the effects of these two expressions.

The key rule in this system is (Restrict). The rule is written to highlight the similarities and differences with the rule for `let`, which introduces normal unrestricted pointers. There are four differences:

- Recall the semantics of `restrict` x `=`e_1 `in` e_2 states that x is a pointer to a copy of the location pointed to by e_1 (Section 2). This naturally suggests giving x a type with a fresh abstract location ρ' during the evaluation of e_2. With this binding we can distinguish accesses through x or copies of x, which have an effect on location ρ', from accesses through other aliases of e_1, which have an effect on location ρ.
- The constraint $\rho \notin L_2$ prevents other aliases of e_1 from being accessed within e_2.
- The constraint $\rho' \notin \mathit{locs}(\Gamma, \tau, \tau_2)$, prevents the new location for x from escaping the scope of e_2. Consider:

```
let x = new 0 in
let p = ... in
  (restrict q = x in
     p := q;
   /* 1 */
   restrict r = x in
     **p)
```

  Suppose `x` has type $\mathit{ref}^{\rho_x}(\mathit{int})$. By (Restrict), the types of `q` and `x` can contain different abstract locations. Let `q`'s type be $\mathit{ref}^{\rho_q}(\mathit{int})$, where $\rho_x \neq \rho_q$. Now if the clause $\rho' \notin \mathit{locs}(\Gamma, \tau, \tau_2)$ were not included in (Restrict), the assignment `p := q` would type check. At program point 1, we would have two different names for the same location—ρ_q and ρ_x—even though neither is `restrict`ed. Thus the dereference `**p` would type check even though the program is incorrect. We forbid ρ' from escaping in (Restrict) to prevent this problem.

- Finally, notice that the conclusion of (Restrict) contains the effect $\{\rho\}$, i.e., restricting a location is itself an effect. This forbids sneaky programs such as:

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

  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.

While the type and effect system presented here is built upon a unification-based alias analysis, restrict (and restrict checking) can also be combined with more precise alias analyses. We have not yet explored this possibility.

3.1 Removing Effects

This section details a kind of polymorphism that we have found to be important for effectively checking `restrict` annotations in programs. Consider a generic sentence in our logic $\Gamma \vdash e : \tau; L$. In practice, surprisingly often it happens that the effect L contains locations that are not mentioned either in the type τ or the environment Γ. The cause of this seemingly odd behavior is easy to see: e may have subexpressions that allocate temporary storage and have effects on that storage. No rule in Figure 2 removes locations from the effect of an expression, so effects simply grow as we move from the leaves to the root of the abstract syntax tree. This behavior is not benign. In recursive functions, these extra locations appear to be in both the effect of recursive calls and the effect of the body of the function, resulting in more locations being equated than should be and frequently causing `restrict` checking to fail.

We need a rule that removes effects:

$$\frac{\Gamma \vdash e : \tau; L}{\Gamma \vdash e : \tau; L \cap \mathit{locs}(\Gamma, \tau)} \ \text{(Down)}$$

(Down) states that effects on locations that are no longer in use—neither part of the result computed by an expression, nor accessible through the environment—can be removed from the effect set [4, 16, 22]. Note that the rule (Down) is the one non-syntactic rule in our system. We can construct a purely syntax-directed version of our system by observing that two applications of (Down) in a row always can be combined into one. Thus, we can assume there is one application of (Down) for each expression in the program. In fact, it is unprofitable to apply (Down) anywhere except before the rule for functions (which, again, we have not shown). Combining these observations yields a syntax-directed system.

3.2 Semantics and Soundness

In this section we give a very brief sketch of the semantics and soundness of `restrict`. Our big-step operational semantics is formulated to prove judgments of the form $S \vdash e \rightarrow v; S'$, meaning that evaluating e starting in initial store S (a map from locations l to values, which may themselves be locations) yields a value v and a (possibly updated) final store S'. Here a value v is either a location l or an integer n (or a function binding, if that were in our source language). We model `restrict` in our semantics using the following rule:

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

This rule uses copying to enforce `restrict`'s semantics. To evaluate `restrict` x `=`e_1 `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. The remainder of our semantics (not shown) is strict in **err**, and any computation that goes wrong reduces to **err** (rather than becoming stuck). Thus, 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 within e_2, but only to dereference it. After 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) that ρ' not escape. (An alternative formulation, which we leave to future work, is to rename occurrences of l' to l after e_2 finishes.)

We can show soundness in the usual way via a subject reduction theorem that shows that the type of an expression is preserved by evaluation. Then since **err** has no type, a program that starts off well-typed can never reduce to **err**:

THEOREM 1 (SOUNDNESS). *If* $\emptyset \vdash e : t; L$ *and* $\emptyset \vdash e \rightarrow r; S'$*, then* r *is not* ***err****.*

Here r is either a value or **err** (all terminating programs reduce to one or the other). In other words, in a program that type checks, no use of `restrict` is found to be invalid at run time.

$$\frac{}{\Gamma, \varepsilon_\Gamma \vdash x : \Gamma(x); \emptyset} \ \text{(Var)}$$

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

$$\frac{\begin{array}{c} \Gamma, \varepsilon_\Gamma \vdash e : \tau; L \qquad \rho \text{ fresh} \\ \varepsilon_\tau \cup \{\rho\} \subseteq \varepsilon_{\mathit{ref}^{\,\rho}(\tau)} \end{array}}{\Gamma, \varepsilon_\Gamma \vdash \texttt{new}\, e : \mathit{ref}^{\,\rho}(\tau); L \cup \{\rho\}} \ \text{(Ref)}$$

$$\frac{\Gamma, \varepsilon_\Gamma \vdash e : \mathit{ref}^{\,\rho}(\tau); L}{\Gamma, \varepsilon_\Gamma \vdash \texttt{*}\, e : \tau; L \cup \{\rho\}} \ \text{(Deref)}$$

$$\frac{\begin{array}{c} \Gamma, \varepsilon_\Gamma \vdash e_1 : \mathit{ref}^{\,\rho}(\tau_1); L_1 \qquad \Gamma, \varepsilon_\Gamma \vdash e_2 : \tau_2; L_2 \\ \tau_1 = \tau_2 \end{array}}{\Gamma, \varepsilon_\Gamma \vdash e_1 \texttt{:=} e_2 : \tau_1; L_1 \cup L_2 \cup \{\rho\}} \ \text{(Assign)}$$

$$\frac{\Gamma, \varepsilon_\Gamma \vdash e : \tau; L}{\Gamma, \varepsilon_\Gamma \vdash e : \tau; L \cap (\varepsilon_\Gamma \cup \varepsilon_\tau)} \ \text{(Down)}$$

$$\frac{\begin{array}{c} \Gamma, \varepsilon_\Gamma \vdash e_1 : \mathit{ref}^{\,\rho}(\tau_1); L_1 \\ \Gamma', \varepsilon_{\Gamma'} \vdash e_2 : \tau_2; L_2 \qquad \Gamma' = \Gamma[x \mapsto \mathit{ref}^{\,\rho}(\tau_1)] \\ \varepsilon_\Gamma \cup \varepsilon_{\mathit{ref}^{\,\rho}(\tau_1)} \subseteq \varepsilon_{\Gamma'} \end{array}}{\Gamma, \varepsilon_\Gamma \vdash \texttt{let}\, x \,\texttt{=} e_1 \,\texttt{in}\, e_2 : \tau_2; L_1 \cup L_2} \ \text{(Let)}$$

$$\frac{\begin{array}{c} \Gamma, \varepsilon_\Gamma \vdash e_1 : \mathit{ref}^{\,\rho}(\tau_1); L_1 \\ \Gamma', \varepsilon_{\Gamma'} \vdash e_2 : \tau_2; L_2 \qquad \Gamma' = \Gamma[x \mapsto \mathit{ref}^{\,\rho'}(\tau_1)] \\ \varepsilon_{\tau_1} \cup \{\rho'\} \subseteq \varepsilon_{\mathit{ref}^{\,\rho'}(\tau_1)} \qquad \varepsilon_\Gamma \cup \varepsilon_{\mathit{ref}^{\,\rho'}(\tau_1)} \subseteq \varepsilon_{\Gamma'} \\ \rho \notin L_2 \qquad \rho' \notin \varepsilon_\Gamma \cup \varepsilon_{\tau_1} \cup \varepsilon_{\tau_2} \end{array}}{\Gamma, \varepsilon_\Gamma \vdash \texttt{restrict}\, x \,\texttt{=} e_1 \,\texttt{in}\, e_2 : \tau_2; L_1 \cup L_2 \cup \{\rho\}} \ \text{(Restrict)}$$

Figure 3: Type Inference Rules for Checking Restrict

4. ALIAS AND EFFECT INFERENCE

We now give an algorithm for checking restrict annotations according to the type rules in Figure 2 together with the rule (Down). The algorithm we give is sound and complete; if there is a proof that a `restrict`-annotated program is correct according to Figure 2, the algorithm finds it.

We assume that the programmer has written their program using `restrict` and that the program type checks according to the standard type rules of the language. Figure 3 gives inference rules that show how to compute the remaining missing elements, namely the locations and effects needed at each point. As is standard, for inference we transform the conditions in the type checking rules into a system of constraints that can be solved if and only if there is some proof according to the rules in Figure 2. We first discuss the constraints and some details of the inference algorithm, and then we describe the individual type rules.

Our rules generate three kinds of constraints C: equality constraints between types, inclusion constraints between effects, and disinclusion constraints between locations and effects:

$$\begin{array}{rcl} C & ::= & \tau_1 = \tau_2 \mid L \subseteq \varepsilon \mid \rho \notin L \\ \tau & ::= & \mathit{int} \mid \mathit{ref}^{\,\rho}(\tau) \\ L & ::= & \emptyset \mid \{\rho\} \mid \varepsilon \mid L_1 \cup L_2 \mid L_1 \cap L_2 \end{array}$$

Here ε is an *effect variable*, which stands for an unknown set of locations. Notice that inclusion constraints between effects are of the special form $L \subseteq \varepsilon$, which makes these constraints particularly easy to solve.

An important algorithmic consideration is how we compute the sets of locations $\mathit{locs}(\tau)$ and $\mathit{locs}(\Gamma)$ required by the type checking rules (Restrict) and (Down). We want to avoid repeatedly traversing type structures τ and type environments Γ at each point in the program—a program with $\mathcal{O}(n)$ expressions may have (monomorphic) types of size $\mathcal{O}(n)$ and environments Γ with $\mathcal{O}(n)$ variables, and thus this part of the algorithm alone would likely be at least quadratic.

Our solution is to memoize the computation of *locs*$(\cdot)$. Recall that effects are sets of locations. We associate an effect variable ε_τ with each type τ, and we maintain this association with an implicit global mapping. As we construct new types, e.g., in (Ref), we generate constraints to represent the locations in the new types:

$$\varepsilon_\tau \cup \{\rho\} \subseteq \varepsilon_{\mathit{ref}^{\,\rho}(\tau)}$$

Then in the type inference rules, instead of *locs*(τ), we use ε_τ, e.g., in (Down).

Similarly, we observe that the type environment is empty at the root of the proof tree and then is only incrementally modified for each subexpression. If we know the set of locations in an environment at an expression e, we can incrementally compute the set of locations in the environments at each of e's subexpressions. We use effect variables ε_Γ to contain the set of locations occurring in environment Γ. Where we extend environment Γ with a new binding $x \mapsto \tau$ in (Let) and (Restrict), we generate a constraint

$$\varepsilon_\Gamma \cup \varepsilon_\tau \subseteq \varepsilon_{\Gamma[x\mapsto\tau]}$$

Thus we succinctly capture *locs*$(\Gamma[x \mapsto \tau])$ without recomputing *locs*(Γ). Because the variables ε_Γ must be communicated between adjacent steps of the proof, they are included as part of the environment (to the left of the turnstile $\vdash$) in the rules of Figure 3.

We briefly discuss the type inference rules in Figure 3. We assume that type checking has already been carried out for the underlying standard types of the language, and that these types are given to us. That is, we do not infer the standard types.

- (Var), (Int), and (Deref) are identical to the type checking rules except for the addition of ε_Γ to the left of the turnstile.
- (Ref) and (Assign) are written with explicit fresh variables and equality constraints between types where needed.
- (Down) is similar to its type checking rule, except we use the variable ε_Γ in place of *locs*(Γ) and ε_τ in place of *locs*(τ). Notice the use of our implicit global mapping of τ to ε_τ.
- (Let) differs in one significant way from its type checking rule. The set of locations $\varepsilon_{\Gamma'}$ of Γ' is taken to be the union of the locations of Γ (which is ε_Γ) and the locations in *ref*$^{\,\rho}(\tau_1)$.
- (Restrict) differs from (Let) in the following ways.
  - The variable x is given a type with the new location ρ', and ρ' instead of ρ is included in the set of locations $\varepsilon_{\Gamma'}$ of environment Γ'.
  - A check ensures that ρ does not appear in the effect L_2 of e_2.
  - A check ensures that ρ' does not escape.
  - There is an extra effect on ρ in the effect of the whole expression.

Let n be the size of the initial program with its standard types. Applying the inference rules in Figure 3 takes $\mathcal{O}(n)$ time and generates a system of constraints C of size $\mathcal{O}(n)$. We split the resolution of the side constraints C into two phases, shown as left-to-right rewrite rules in Figure 4. First, we solve the type equality constraints $\tau_1 = \tau_2$ using the rules in Figure 4a. Because we assume checking of the standard types has already been done, the type equality rules can never discover an inconsistency. However, the type equalities must still be solved to discover all implied constraints between ρ and ε variables. This step requires $\mathcal{O}(n)$ time.

$$\begin{array}{rcl}
C \cup \{\mathit{int} = \mathit{int}\} & \Rightarrow & C \\
C \cup \{\mathit{ref}^{\,\rho_1}(\tau_1) = \mathit{ref}^{\,\rho_2}(\tau_2)\} & \Rightarrow & \\
 & & C \cup \{\rho_1 = \rho_2\} \cup \{\tau_1 = \tau_2\} \\
C \cup \{\rho_1 = \rho_2\} & \Rightarrow & C[\rho_1 \mapsto \rho_2] \\
C \cup \{\varepsilon_1 = \varepsilon_2\} & \Rightarrow & C[\varepsilon_1 \mapsto \varepsilon_2]
\end{array}$$

(a) Type Equality

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

(b) Constraint Normalization

Figure 4: Constraint Resolution

The resulting constraints are of the form $L \subseteq \varepsilon$ and $\rho \notin L$. We call such a system of constraints an *effect constraint system*. A *solution* to an effect constraint system C is a mapping σ from effect variables to sets of locations such that $\sigma(L) \subseteq \sigma(\varepsilon)$ and $\rho \notin \sigma(L)$ for each constraint $L \subseteq \varepsilon$ and $\rho \notin L$ in C, where we extend σ from effect variables to arbitrary effects in the natural way. An effect constraint system is *satisfiable* if it has a solution. Notice that abstract locations are not in the domain of σ—intuitively, after discovering all equalities between locations after applying the rules in Figure 4a, we can treat abstract locations as constants.

We define a partial order on solutions, $\sigma \leq \sigma'$ iff for every effect variable ε we have $\sigma(\varepsilon) \subseteq \sigma'(\varepsilon)$. The *least solution* to an effect constraint system is the solution σ such that $\sigma \leq \sigma'$ for any other solution σ'. If an effect constraint system C has any solution, then C has a least solution [13, 14].

To test satisfiability of an effect constraint system, we first apply the rules in Figure 4b to translate the constraints into the following normal form:

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

Notice that the rules in Figure 4b preserve least solutions but not arbitrary solutions. Also notice that in Figure 4b we do not consider the case $(L_1 \cap L_2) \cap L \subseteq \varepsilon$ or $L \cap (L_1 \cap L_2) \subseteq \varepsilon$. Such constraints are never generated once (Down) is merged into the rule for functions (not shown). Applying the rules in Figure 4b takes time $\mathcal{O}(n)$.

We view the inclusion constraints in a normal form effect con-

```
CHECK-SAT($\rho \notin \varepsilon$):
    Associate $Count(v)$ with each node $v$ in the graph
    Initialize $Count(v) = 0$ for all $v$
    Let $W = \{\rho\}$, the set of nodes left to visit
    While $W$ is not empty
        Remove some node $v$ from $W$
        If $v == \varepsilon$ return unsatisfiable
        For each edge $v \to \varepsilon'$
            If $Count(\varepsilon') == 0$ then
                $Count(\varepsilon') = 1$
                Add $\varepsilon'$ to $W$
        For each edge $v \to I$
            If $Count(I) == 0$ then
                $Count(I) = 1$
            Else if $Count(I) == 1$ then
                $Count(I) = 2$
                Add $I$ to $W$
    Return satisfiable
```

Figure 5: Checking satisfiability of $\rho \notin \varepsilon$

straint system as a directed graph:

Constraint	Edge(s)
$\{\rho\} \subseteq \varepsilon$	$\rho \to \varepsilon$
$\varepsilon_1 \subseteq \varepsilon_2$	$\varepsilon_1 \to \varepsilon_2$
$M_1 \cap M_2 \subseteq \varepsilon$	$M_1 \to I$ $M_2 \to I$ $I \to \varepsilon$ I fresh

The nodes of the directed graph are abstract locations ρ (with in-degree 0), effect variables ε (with arbitrary in-degree), and intersections I (with in-degree 2). We generate a fresh I node for each constraint $M_1 \cap M_2 \subseteq \varepsilon$.

Given a normal form effect constraint system, we test satisfiability by checking, for each constraint $\rho \notin \varepsilon$, whether $\rho \in \sigma(\varepsilon)$ in the least solution σ. Figure 5 shows the modified depth-first search we use to check this condition. The algorithm in Figure 5 takes time $\mathcal{O}(n)$ for each $\rho \notin \varepsilon$ constraint. Given an initial program with k occurrences of `restrict`, the system considered in Figure 5 has $\mathcal{O}(k)$ constraints of the form $\rho \notin \varepsilon$. Hence the time for this step is $\mathcal{O}(kn)$, which is also the total time for the algorithm.

5. RESTRICT INFERENCE

The type checking algorithm of the previous section checks user-supplied `restrict` annotations. In practice, however, many such annotations may be necessary to give the quality of aliasing information needed for other analyses, and adding these annotations by hand can be very time-consuming. In this section we give an algorithm for automatically adding `restrict` to a program. More precisely, we show how to automatically decide whether a binding construct should be `let` or `restrict`. A bit surprisingly, our type rules always admit a unique maximum set of `let` expressions that can be `restrict`ed. Our inference algorithm computes this optimal annotation of the program.

As we have observed, `let` and `restrict` differ only in a few ways. Our approach is to combine the inference rules (Let) and (Restrict) into a single rule (Let-or-Restrict), corresponding to a new construct `let-or-restrict` x =e_1 `in` e_2, with two properties. First, in any solution of the constraints, (Let-or-Restrict) satisfies the requirements of exactly one of the (Let) or (Restrict) rules. Second, (Let-or-Restrict) "prefers" the (Restrict) solution: if the constraints have any solution satisfying the requirements of (Restrict), then that will be the least solution.

Recall that `restrict` has four differences from `let`. We consider each of these in turn. First, a `restrict` uses two locations ρ and ρ' where `let` has only ρ. Thus, our inference rule should begin by assuming ρ and ρ' are distinct (the `restrict` case), and if it turns out that the expression cannot be a `restrict`, the locations should be unified ($\rho = \rho'$, the `let` case).

Second, there are two negative constraints

$$\begin{array}{rcl}\rho & \notin & L_2\\ \rho' & \notin & \varepsilon_\Gamma \cup \varepsilon_{\tau_1} \cup \varepsilon_{\tau_2}\end{array}$$

in `restrict`. If either of these is unsatisfiable, then the expression must be a `let`. We can combine this with the reasoning above to yield the following constraints:

$$\begin{array}{rcl}\rho \in L_2 & \Rightarrow & \rho = \rho'\\ \rho' \in (\varepsilon_\Gamma \cup \varepsilon_{\tau_1} \cup \varepsilon_{\tau_2}) & \Rightarrow & \rho = \rho'\end{array}$$

These constraints say that if either the old location ρ is used in the body of the construct, or the new location ρ' escapes, then the locations must be equal and the construct is a `let`. We have not seen such *conditional* constraints before in this paper. These constraints are easy to solve, though we omit the details here.

Finally, there is the extra effect on ρ in the result of `restrict`. If the expression is a `restrict` we must have the effect, and if it is a `let` we must not. Given the semantics of `restrict` we have used so far, we do not know how to model this choice efficiently. However, if we interpret `restrict` a little more liberally, an efficient solution is straightforward.

Consider the construct `let-or-restrict` x =e_1 `in` e_2, which will behave either as `restrict` or `let`. If e_2 has an effect on ρ', we are done: if it is a `restrict` there is an effect on ρ and if it is a `let` there is also an effect on ρ because $\rho = \rho'$. What if e_2 has no effect on ρ'? In that case, we do not need to require that `restrict` have an effect on ρ! Recall from the example in Section 3 that the extra effect is needed to prevent ρ from being restricted twice and both restricted pointers used simultaneously. If a restricted pointer is not used at all, there is no need to prevent it from being restricted a second time in the same scope.[2] These two cases (e_2 does or does not have an effect on ρ') can be combined in one additional conditional constraint

$$(\rho' \in L_2) \Rightarrow \{\rho\} \subseteq \varepsilon$$

where L_2 is the effect of e_2 and ε is included in the effect of the entire expression.

Putting everything together, we have the following inference rule:

$$\frac{\begin{array}{c}\Gamma, \varepsilon_\Gamma \vdash e_1 : \mathit{ref}^{\,\rho}(\tau_1); L_1\\ \Gamma', \varepsilon_{\Gamma'} \vdash e_2 : \tau_2; L_2 \qquad \Gamma' = \Gamma[x \mapsto \mathit{ref}^{\,\rho'}(\tau_1)]\\ \varepsilon_{\tau_1} \cup \{\rho'\} \subseteq \varepsilon_{\mathit{ref}^{\,\rho'}(\tau_1)} \qquad \varepsilon_\Gamma \cup \varepsilon_{\mathit{ref}^{\,\rho'}(\tau_1)} \subseteq \varepsilon_{\Gamma'}\\ \rho' \in (\varepsilon_\Gamma \cup \varepsilon_{\tau_1} \cup \varepsilon_{\tau_2}) \Rightarrow \rho = \rho'\\ \rho \in L_2 \Rightarrow \rho = \rho'\\ \rho' \in L_2 \Rightarrow \{\rho\} \subseteq \varepsilon\\ \rho, \rho', \varepsilon \text{ fresh}\end{array}}{\Gamma, \varepsilon_\Gamma \vdash \texttt{let-or-restrict}\ x = e_1\ \texttt{in}\ e_2 : \tau_2; L_1 \cup L_2 \cup \varepsilon}$$

[2]This is consistent with the semantics of `restrict` in C. We have not introduced this semantics before now because it is more complicated.

This new rule, which replaces (Let) and (Restrict), allows us to infer `restrict` annotations. It is easy to check that these constraints have a least solution, which guarantees the existence of an optimal annotation of a program with `restrict`.

A straightforward implementation of this inference rule gives a quadratic time algorithm. Given a typed program of size n, there are $\mathcal{O}(n)$ possible locations and $\mathcal{O}(n)$ constraints. Computing initial reachability in the constraint graph (without the conditional constraints) for all $\mathcal{O}(n)$ locations using the algorithm in Figure 5 takes $\mathcal{O}(n^2)$ time. We maintain a work-list of conditional constraints whose left-hand side has become true. For each conditional constraint on our work-list, we perform $\mathcal{O}(n)$ extra work to recompute reachability for the unified locations (or to recompute reachability for the location ρ in the constraint $\{\rho\} \subseteq \varepsilon$). Since there are $\mathcal{O}(n)$ possible total unifications, and each may trigger $\mathcal{O}(n)$ work, the overall complexity is $\mathcal{O}(n^2)$.

6. CONFINE

Recall that, as in the code in Figure 1, many realistic examples where `restrict`-like functionality is useful involve values held in containers. Unlike Figure 1, however, in practice programs often do not include an explicit variable to `restrict`. For example, consider

```
spin_lock(&locks[i]);
work();
spin_unlock(&locks[i]);
```

Assuming that `work()` does not modify `i` or access any other elements of the array `locks`, we can rewrite this code using `restrict` to allow a checker like CQUAL to analyze this code:

```
restrict x = &locks[i] in {
  spin_lock(x);
  work()                                (*)
  spin_unlock(x); }
```

While this is effective, implementing this transformation by hand is tedious, not only because we must introduce a new name and manually perform a substitution, but also because we must discover the scope of the `restrict` and check that the `restrict`ed expression refers to the same object throughout the body. For instance, in this case, we must notice that there are two occurrences of `&locks[i]` in the code that refer to the same object, and then we must put the scope of the new variable that names the lock around both uses. Note that in the system described in Section 5 we only infer `restrict` for variables, which come with an obvious scope.

Editing a program to add a few `restrict` annotations is not difficult; the problem is that in practice there are many `restrict` annotations to add, and many involve expressions. In our experience, the manual labor required to introduce local variables for all of those expressions is just too much. Our solution is to introduce a new construct, `confine`, that deals specifically with restricting the aliases of expressions. The syntax is

$$\texttt{confine}\, e_1\, \texttt{in}\, e_2$$

meaning that aliases of the location e_1 refers to are `restrict`ed in the scope e_2. (Note that our use of the word *confine* is not related to the term as used in object-oriented alias control systems [28].) The expression e_1 itself serves as the name for the `restrict`ed location. Assuming all program variables have been renamed to be distinct, we define `confine` syntactically by translation to `restrict`:

$$\texttt{confine}\, e_1\, \texttt{in}(e_2[e_1/x]) \;=\; \texttt{restrict}\, x\, \texttt{=} e_1\, \texttt{in}\, e_2$$

where x is a fresh variable that is substituted for occurrences of e_1 in e_2. For nested `confine`s, the translation must be done innermost-first.

For instance, the example above would be written:

```
confine (&locks[i]) in {
  spin_lock(&locks[i]);
  work();
  spin_unlock(&locks[i]);
}
```

and this is defined to be equivalent to (*) above. Notice that with `confine` we do not need to rewrite the body of the `restrict`ed scope—we need only wrap an appropriate `confine` around it.

Our goal is to perform `confine` inference—to automatically place `confine`s in the program. Intuitively, `confine` inference corresponds to performing a kind of common sub-expression elimination that handles aliasing and then applying restrict inference. There are two issues:

- *Referential transparency.* For the definition given above of `confine` in terms of `restrict` to make sense, an expression that is `confine`d must truly behave like a name within the scope of the `confine`. An expression behaves like a name in a scope only if it is referentially transparent within that scope[3]—if it in fact always evaluates to the same value.

- *Inferring scopes.* As mentioned above, we must determine the scope for a `confine`.

6.1 Referential Transparency

Consider an expression `confine` e_1 `in` e_2. To enforce referential transparency of e_1 within the scope of e_2, we must first ensure that e_1 terminates. Our solution is to simply forbid e_1 from containing a function application, and indeed for the experiment described in Section 7 we are interested only in e_1's that are composed of identifiers, field accesses, and pointer dereferences.

To enforce referential transparency, we must also be certain that neither e_1 nor e_2 modify any of the locations e_1 needs during its evaluation—locations used by e_1 must only be read within e_1 and e_2. To accomplish this we must extend our notion of effect. For a given location ρ, we now distinguish effects *read(ρ)* (reads of location ρ), *write(ρ)* (writes to location ρ), and *alloc(ρ)* (allocation of location ρ). Our rules now have two different kinds of sets: sets S of locations (as before) and sets L of *read*, *write*, and *alloc* effects on locations. For clarity, we separate these two kinds of sets and use variables π for sets of *read*, *write*, and *alloc* effects, and variables ε for sets of locations. The grammars for sets are now:

$$\begin{array}{rcl} L & ::= & \emptyset \mid \{\mathit{read}(\rho)\} \mid \{\mathit{write}(\rho)\} \mid \{\mathit{alloc}(\rho)\} \mid \pi \\ & & \mid L_1 \cup L_2 \mid L_1 \cap L_2 \\ S & ::= & \emptyset \mid \{\rho\} \mid \varepsilon \mid S_1 \cup S_2 \mid S_1 \cap S_2 \end{array}$$

All inference rules must be modified to correctly report read, write, and allocation effects on locations; we omit the details due to space constraints. We also need to introduce a new kind of *effectful variable* x_L, which is typechecked just like a regular variable x, except that evaluating x_L has effect L:

$$\frac{}{\Gamma, \varepsilon_\Gamma \vdash x_L : \Gamma(x); L} \; (\mathrm{Var}_L)$$

Intuitively, we could also model x_L as a thunk if the language used in this paper contained functions. We discuss the use of effectful variables x_L below.

[3]Our usage is slightly non-standard; in the standard use of the term "referential transparency," the scope referred to is the whole program.

To perform type inference for `confine` e_1 `in` e_2, we modify `let-or-restrict` for use with `confine`:

$$\frac{\begin{array}{c}\Gamma, \varepsilon_\Gamma \vdash e_1 : \mathit{ref}^{\,\rho}(\tau_1); L_1 \\ \Gamma', \varepsilon_{\Gamma'} \vdash e_2[x_{\pi'}/x] : \tau_2; L_2 \qquad \Gamma' = \Gamma[x \mapsto \mathit{ref}^{\,\rho'}(\tau_1)] \\ \varepsilon_{\tau_1} \cup \{\rho'\} \subseteq \varepsilon_{\mathit{ref}\,\rho'(\tau_1)} \qquad \varepsilon_\Gamma \cup \varepsilon_{\mathit{ref}\,\rho'(\tau_1)} \subseteq \varepsilon_{\Gamma'} \\ \rho' \in (\varepsilon_\Gamma \cup \varepsilon_{\tau_1} \cup \varepsilon_{\tau_2}) \Rightarrow (\rho = \rho' \wedge L_1 \subseteq \pi') \\ X(\rho) \in L_2 \Rightarrow (\rho = \rho' \wedge L_1 \subseteq \pi') \\ X(\rho') \in L_2 \Rightarrow \{X(\rho)\} \subseteq \pi \\ \forall \rho''.\mathit{write}(\rho'') \in L_1 \Rightarrow (\rho = \rho' \wedge L_1 \subseteq \pi') \\ \forall \rho''.\mathit{alloc}(\rho'') \in L_1 \Rightarrow (\rho = \rho' \wedge L_1 \subseteq \pi') \\ \forall \rho''.\mathit{read}(\rho'') \in L_1 \Rightarrow (\mathit{write}(\rho'') \in L_2 \Rightarrow (\rho = \rho' \wedge L_1 \subseteq \pi')) \\ \forall \rho''.\mathit{read}(\rho'') \in L_1 \Rightarrow (\mathit{alloc}(\rho'') \in L_2 \Rightarrow (\rho = \rho' \wedge L_1 \subseteq \pi')) \\ \rho, \rho', \pi, \pi' \text{ fresh}\end{array}}{\Gamma, \varepsilon_\Gamma \vdash \texttt{confine?}\ e_1\ \texttt{in}\ e_2[e_1/x] : \tau_2; L_1 \cup L_2 \cup \pi}$$

The name `confine?` is meant to be suggestive of an "optional `confine`." This rule chooses whether to insert `confine` based on the solution of the constraints (if $\rho \neq \rho'$ and $L_1 \not\subseteq \pi'$ there is a `confine`, otherwise there is not). The easiest way to understand this rule is to compare it with the inference rule for `let-or-restrict` (Section 5). The first line of the two rules is identical. In the second line, we bind x as before, and we replace x in e_2 with an effectful variable $x_{\pi'}$. Recall that in e_2, occurrences of e_1 have been replaced by x. If inserting `confine` succeeds, $\pi' = \emptyset$ in the least solution of the constraints, so $x_{\pi'}$ is equivalent to x. In other words, if inserting `confine` succeeds, then we replace occurrences of the common sub-expression e_1 by x. But if inserting `confine` fails (for any reason) then the constraint $L_1 \subseteq \pi'$ will be generated, and therefore in this case when type-checking e_2 we will give each occurrence of x both e_1's type and e_1's effect. Thus, if inserting `confine` fails, we do not eliminate common sub-expression e_1.

The third and fourth line of our rule for `confine?` are as in our rule for `let-or-restrict`, with the addition of the constraint $L_1 \subseteq \pi'$. In the next two lines we have used a short-hand in the new rule: X is a wildcard constructor standing for any of *read*, *write*, or *alloc*. These constraints simply instantiate the requirements of the original rule for each of the three specific kinds of effects. The four lines beginning with universal quantifiers are the referential transparency constraints specific to `confine`. The first two lines of the premise require that the `confine`d expression e_1 have no side effects; the last two lines say that if a location ρ'' is read by e_1 (i.e., the effect $\mathit{read}(\rho'')$ is in L_1) then ρ'' cannot be written or allocated by e_2. In other words, the last two lines prevent e_1 from being confined if the meaning of e_1 may be changed by an assignment in e_2.

It is important to understand that it is not necessary to actually introduce new program variables and carry out inverse substitutions to implement `confine` inference. Our reduction of `confine` inference to `restrict` inference allows us to rely on soundness results for `restrict`, but an efficient implementation is possible without explicit program transformations. For space reasons we have not given a type checking rule for `confine`. A `confine` rule is easily derived from the rule for `confine?` by requiring that $\rho \neq \rho'$ and simplifying.

6.2 Inferring Scopes

Our `confine?` rule gives us a method for inferring `confine`. This facility gives us a very simple way to automatically infer the best (largest) scope in which to confine a given expression e_1. In Section 7, we use this technique to confine expressions corresponding to locks in the Linux kernel.

Briefly, the main idea is that to infer the scope of a `confine` of an expression e_1, we add `confine?` e_1 `in` e_2 to every possible scope e_2 and pick the largest scope where inference succeeds (where "succeeds" means that the solutions of the constraints indicate that a `confine` can be added). The possible scopes where e_1 can be confined are just those where the free variables of e_1 are in scope. For example, if we have

$$\texttt{let}\ x = e_3\ \texttt{in}(\texttt{let}\ y = e_2\ \texttt{in} \ldots e_1 \ldots)$$

then confine inference for e_1 adds `confine?` to the outer scopes:

$$\begin{array}{l}\texttt{confine?}\ e_1\ \texttt{in}(\texttt{let}\ x = e_3\ \texttt{in} \\ \qquad\quad \texttt{confine?}\ e_1\ \texttt{in}(\texttt{let}\ y = e_2\ \texttt{in} \ldots e_1 \ldots))\end{array}$$

assuming that e_1 did not mention either x or y. Confine inference is then carried out. After the constraints are solved, we select the outermost `confine?` that succeeds, if indeed any succeed. Note that this method checks all `confine?` expressions simultaneously. Given a typed program of size n and a fixed expression e_1, there are $\mathcal{O}(n)$ possible places to insert `confine?`. Finding syntactic occurrences of e_1 takes $\mathcal{O}(n^2)$ time. Solving the constraints for $\mathcal{O}(n)$ `confine?`s takes $\mathcal{O}(n^2)$ time. To see this, note that our type inference rules generate $\mathcal{O}(n)$ locations, and computing reachability for a location in the constraint graph (using an algorithm similar to that in Figure 5) takes $\mathcal{O}(n)$ time. Thus, computing an initial least solution of every location, before taking any conditional constraints into account, takes time $\mathcal{O}(n^2)$. Then for any conditional constraint whose left-hand side is true, we perform $\mathcal{O}(n)$ extra work to recompute reachability for the unified locations and to propagate the least solution of L_1 to π'. Since there are $\mathcal{O}(n)$ possible total unifications, and each may trigger $\mathcal{O}(n)$ work, the overall complexity is $\mathcal{O}(n^2)$.

In our implementation, we use an algorithm with a higher worst-case running time but better performance in practice. Rather than computing reachability for every location in the constraint graph (which takes $\mathcal{O}(n^2)$ time), we do a backwards search from effects in constraints generated for `confine?` to find which locations reach them. Since this tends to be a small portion of the constraint graph, this is usually more efficient.

7. EXPERIMENTS

As discussed in Section 1, among other applications, we are interested in `restrict` and `confine` because they enable strong updates in flow-sensitive analyses. To assess the usefulness of our `confine` construct in practice, we have implemented `confine` inference in CQUAL and tested it in conjunction with CQUAL's flow-sensitive analysis. This section presents the results of that experience.

The flow-sensitive analysis we use is the analysis of locking behavior reported previously [15]. The Linux kernel has two primitives `spin_lock(e)` and `spin_unlock(e)` for acquiring and releasing locks, respectively. By tracking the state (held or not held) of locks, we can detect when a lock is acquired or released twice in succession within a single thread. These particular programming errors are surprisingly common in Linux device drivers [11].

In previous experiments [15], we examined hundreds of device drivers and discovered numerous locking errors by inspecting the results of analyzing single files and a few entire device drivers. Potentially, more errors could be found by analyzing entire device driver modules (which typically consist of many files), but we discovered in analyzing many whole modules that the aliasing of locks became so pervasive that it was very difficult to separate true bugs from type errors reported due to spurious aliases, and it was simply impractical to remove the spurious aliases by adding all of the needed `restrict` annotations by hand.

We have repeated our locking experiments, using `confine` inference as described in Section 6 to try to `confine` any arguments passed to `change_type()`, a special state-changing statement built in to CQUAL [15]—in particular, we try to `confine` any arguments to `spin_lock` or `spin_unlock`. There is one wrinkle in performing `confine` inference in a language with blocks of statements

$$\{e_1; \ldots e_i; \ldots e_j; \ldots e_n\}$$

such as C. Sometimes it is necessary to `confine` only a portion of the block with respect to an expression e; e.g., we may need to introduce a new scope to write

$$\{e_1; \ldots; \texttt{confine}\, e\, \texttt{in}\{e_i; \ldots e_j;\}; \ldots e_n\}$$

The problem is to discover to what portion of a block we can add `confine`. Observe that

$$(\texttt{confine}\, e\, \texttt{in}\, e_1; \texttt{confine}\, e\, \texttt{in}\, e_2) = (\texttt{confine}\, e\, \texttt{in}\{e_1; e_2\})$$

That is, adjacent `confine`s of the same expression can be combined. This suggests the following general algorithm: in a block $e_1; \ldots e_n$, for each i, add `confine?` e `in` e_i and then greedily combine all adjacent `confine?`s that succeed. This algorithm discovers the largest possible `confine` sub-blocks within a block of statements.

To improve performance, our current implementation introduces new sub-blocks for `confine?` using a slightly different algorithm based on a syntactic heuristic. For each statement in the program (including statement blocks), we keep track of whether the statement contains `change_type`. When two statements in the same block contain `change_type`, and the arguments to `change_type` match syntactically, we introduce the smallest possible sub-block around the two statements and report that the new sub-block does not contain a `change_type`. Intuitively, this heuristic tries to put `confine?` around sets of statements that call `spin_lock` and `spin_unlock` with the same syntactic expression. The heuristic is weaker than the general strategy outlined above, but still works well in our experiment, as our results suggest. Also notice that although the introduction of `confine?` is a syntactic heuristic, our `confine` inference algorithm uses our type and effect system to decide whether introducing `confine` is safe.

We analyzed 589 whole device driver modules from the 2.4.9 Linux kernel. We used CQUAL in three different modes: without confine inference, with confine inference, and finally in a mode where all updates are assumed to be strong. In each case we measured the number of type errors reported by CQUAL in the flow-sensitive pass—here the number of type errors is the number of syntactic calls to `spin_lock()` and `spin_unlock()` where CQUAL could not verify that locks are held in the correct state. Since one application of `confine` is to enable strong updates in a flow-sensitive analysis, the last mode provides an upper bound on the number of spurious type errors that can be eliminated by adding `confine` annotations.

Of the 589 modules, 352 are free of type errors without the addition of any `confine` annotations. 85 of the remaining modules contain type errors, but not because of strong updates—using no `confine` at all yields the same type errors as assuming all updates are strong. Of the remaining 152 modules where CQUAL reports type errors, using `confine` inference produces the same type errors as assuming all updates are strong in 138 modules. That is, in 138 of 152 modules where `confine` inference could make a

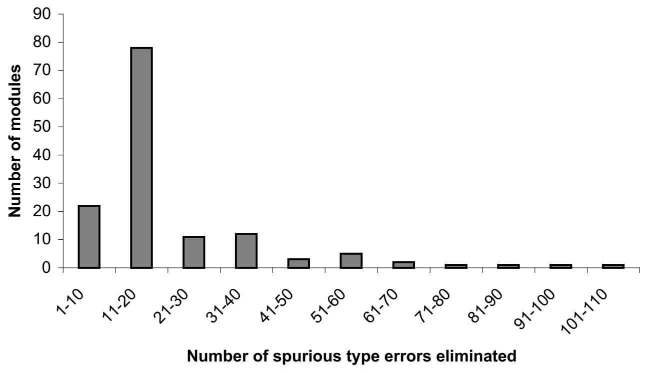

Figure 6: Spurious type errors eliminated by `confine` inference.

Module	Number of lock/unlock type errors		
	no confine inference	confine inference	all updates strong
wavelan_cs	22	16	15
trix	29	24	22
netrom	41	25	0
rose	47	28	0
usb-ohci	32	26	17
uhci	74	45	34
sb	31	24	22
ide-tape	58	47	41
mad16	29	24	22
emu10k1	198	60	35
trident	107	49	36
digi_aceleport	62	32	4
sbni	23	16	9
iph5526	39	34	32

Figure 7: Modules for which `confine` inference does not infer all possible strong updates.

difference, it removes all spurious type errors due to lack of strong updates. Figure 6 shows the distribution of type errors eliminated by `confine` inference for these modules. Summing the individual results for all modules, `confine` inference could potentially eliminate 3,277 type errors, and it succeeds in eliminating 3,116 type errors, or 95%. Note that the type error counts should be taken with a grain of salt, as multiple type errors often have a single root cause. Also, many modules share files, so even type errors reported in different modules are not independent.

Of the 152 modules, 14 contain sites where `confine` inference cannot infer that a potentially useful strong update is possible. We give the type error counts for each of the three modes of usage for these modules in Figure 7. We have examined the type error reports from several of these 14 modules to discover where `confine` inference fails. In some cases, our underlying may-alias analysis is unable to verify the addition of `confine` without programmer intervention (e.g., a type cast). In other cases, there is not a well-defined lexical scope for `confine`; these cases often involve quite tricky coding styles.

Although it is not the subject of this paper, as mentioned above even assuming that all updates are strong, CQUAL reports at least some type errors in 137 of the modules. The sources of these type errors are mostly the same as reported previously [15]. In particular, there are a few places where a path sensitive analysis would be useful, and others where we need the ability to model the sequential acquiring or releasing of a set of aliased locks at once. There are also just plain program errors, including 4 apparently new bugs we have found since the results reported previously [15]. These bugs were present in the code analyzed previously [15] but escaped notice simply because the large number of spurious type errors caused by the lack of `confine` inference. Finally, so far we have found one place where the addition of location polymorphism would remove a CQUAL type error.

The performance impact of `confine` inference on CQUAL is modest, because in our experiments the pointer-valued expressions that are `confine`d tend to be small and because we only try to `confine` arguments to `spin_lock` or `spin_unlock`. For example, in the largest module where `confine` inference eliminated some type errors (`ide-tape`) CQUAL ran in 28.5 seconds with `confine` inference and in 26.0 seconds without it. The running time of `confine` inference for other modules is a similarly small fraction of the overall time.

8. RELATED WORK

Effect systems were first described by Gifford and Lucassen for FX-87 [16, 22]. FX-87 includes subtyping, polymorphism, and notation for declaring the effects of expressions [16]. One of the best-known type and effect systems is the region type system proposed by Tofte and Talpin [27]. Our type systems, and particularly the system for automatically inferring where to place `restrict` and `confine` annotations, is related to region inference. The ρ annotations can be thought of as regions, and we can apply the rule (Down) (which is borrowed from a region type system [4]) whenever we discover that a location is purely local to a lexical scope of the computation [4]. One important difference between restrict/confine inference and region inference is that introducing restrict and confine requires that certain locations not be accessed within their scope, whereas introducing a new region never decreases the set of accessible locations.

Wang and Appel [29] use a technique very similar to `restrict` to check that covariant subtyping under reference types is safe. This can be seen as another application of `restrict`.

Automatic alias analysis has been heavily studied in recent years; a few of the many proposed analyses are [1, 7, 10, 18, 20, 21, 24, 26, 30]. Our type system incorporates may-alias analysis to check the correctness of `restrict` and `confine`. The may-alias analysis we use is very conservative, and it is possible that a more expressive (and expensive) may-alias analysis would be useful in practice to improve the precision of `restrict` and `confine` checking and inference.

One of the limitations of our approach is that `restrict` and `confine` must be lexically scoped. This assumption fits well with many, but not all, uses of `restrict` and `confine` in practice. Other type-based systems that model strong and weak updates and do not have lexical scoping restrictions [9, 12, 25] are more expressive, but also less suited to tractable automatic inference than our approach. For example, Boyland [3] shows how to check several programming paradigms using non-lexically scoped linearities and flow-sensitive aliasing information.

There are several systems for modeling *uniqueness* in object oriented programming languages [19, 23, 3]. In these systems a unique object always has exactly one pointer pointing to it. In contrast, in our system a location pointed to by a restricted pointer may be pointed to by arbitrarily many pointers. However, some of the techniques from the literature on uniqueness may be applicable to `restrict` and `confine`. For example, a type system by Clarke and Wrigstad [6] allows a unique object to have a non-unique view in a scope while leaving the unique pointer to the object alive by forbidding the escape of aliases created in the scope.

As discussed in the introduction, one of the most interesting properties of `restrict` and `confine` is that they allow us to locally recover the ability to treat a pointer as a reference to a unique value, which allows analyses that use `restrict` and `confine` information to perform strong updates [5]. This idea is the subject of previous work [15] that combines `restrict` with ideas from flow-sensitive type systems [25]. The resulting system can be used to check flow-sensitive program properties.

Several other systems, such at Meta-level compilation [17] and ESP [8], check flow-sensitive program properties using approaches more directly based on dataflow analysis. In these systems, arbitrary dataflow facts are associated with each program point. In contrast, our approach can be seen as associating a more restricted language of facts, i.e., the qualified type of each abstract location ρ, with each program point. This yields a quite different design tradeoff: Meta-level compilation and ESP support a richer language of facts that is correspondingly more complex to reason about. In CQUAL, the facts are easier to reason about but less expressive. The `restrict` and `confine` constructs regain some expressiveness by enriching the set of abstract locations, which correspondingly enriches the set of possible facts at each program point.

9. CONCLUSION

In this paper we have presented `restrict`, a language construct that allows a programmer to specify that certain pointers are not aliased within a lexical scope. We have shown both how to automatically check the correctness of `restrict` annotations using an alias and effect system, and how to automatically infer which `let` bindings may be safely changed to `restrict` bindings.

We have also developed `confine`, which allows an expression to be `restrict`ed, and shown how to automatically add `confine` annotations to a program. We have shown that automatic `confine` inference can be used to recover nearly all important strong updates needed for a flow-sensitive analysis to check locking behavior in Linux kernel device drivers. Although our experiments to date have focused on using `restrict` and `confine` to aid CQUAL, we believe that `restrict` and `confine` can be profitably applied as programmer annotations in other systems.

In our view, the key benefit of our approach is that `restrict` and `confine` give the programmer a handle on an alias analysis and, by extension, any subsequent analyses (e.g., program verification tools) based on aliasing information. We feel that this exposure of aliasing information is important, especially to express critical aliasing invariants needed to verify flow-sensitive program properties. However, we also believe that many uses of `restrict` and `confine` are routine. Thus, we believe a workable approach is to support both automatic inference of `restrict` and `confine` as well as programmer `restrict` and possibly `confine` annotations.

10. REFERENCES

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, May 1994.

[2] ANSI. *Programming languages – C*, 1999. ISO/IEC 9899:1999.

[3] J. Boyland. Alias burying: Unique variables without destructive reads. *Software—Practice and Experience*, 31(6):533–553, May 2001.

[4] C. 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, Jan. 2001.

[5] D. R. Chase, M. Wegman, and F. K. 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.

[6] D. Clarke and T. Wrigstad. External Uniquness. In *Proceedings of the 10th International Workshop on Foundations of Object-Oriented Languages*, Jan. 2003.

[7] M. 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.

[8] M. Das, S. Lerner, and M. 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.

[9] R. DeLine and M. 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.

[10] M. Emami, R. Ghiya, and L. 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.

[11] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. 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, Oct. 2001.

[12] M. Fähndrich and R. 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.

[13] J. S. Foster. *Type Qualifiers: Lightweight Specifications to Improve Software Quality*. PhD thesis, University of California, Berkeley, Dec. 2002.

[14] J. S. Foster and A. Aiken. Checking Programmer-Specified Non-Aliasing. Technical Report UCB//CSD-01-1160, University of California, Berkeley, Oct. 2001.

[15] J. S. Foster, T. Terauchi, and A. 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.

[16] D. K. Gifford, P. Jouvelot, J. M. Lucassen, and M. A. Sheldon. FX-87 Reference Manual. Technical Report MIT/LCS/TR-407, MIT Laboratory for Computer Science, Sept. 1987.

[17] S. Hallem, B. Chelf, Y. Xie, and D. 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.

[18] N. Heintze and O. 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.

[19] J. Hog. Islands: aliasing protection in object-oriented languages. In *Proceedings of the sixth annual conference on Object-oriented programming systems, languages, and applications*, pages 271–285, Oct. 1991.

[20] W. Landi and B. 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.

[21] D. Liang and M. J. Harrold. Efficient Computation of Parametrized Pointer Information for Interprocedural Analyses. In P. Cousot, editor, *Static Analysis, Eighth International Symposium*, volume 2126 of *Lecture Notes in Computer Science*, pages 279–298, Paris, France, July 2001. Springer-Verlag.

[22] J. M. Lucassen and D. 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, Jan. 1988.

[23] N. Minsky. Towards Alias-Free Pointers. In *Proceedings of the tenth European Conference on Object Oriented Programming*, 1996.

[24] J. Rehof and M. 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, Jan. 2001.

[25] F. Smith, D. Walker, and G. Morrisett. Alias Types. In G. Smolka, editor, *9th European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–381, Berlin, Germany, 2000. Springer-Verlag.

[26] B. 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, Jan. 1996.

[27] M. Tofte and J.-P. 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, Jan. 1994.

[28] J. Vitek and B. Bokowski. Confined Types. In *Proceedings of the fourteenth annual conference on Object-oriented programming systems, languages, and applications*, pages 82–96, Oct. 1999.

[29] D. C. Wang and A. W. Appel. Type-Preserving Garbage Collectors. In *Proceedings of the 28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, London, United Kingdom, Jan. 2001.

[30] R. P. Wilson and M. 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.