LOCKSMITH: Context-Sensitive Correlation Analysis for Race Detection *

Polyvios Pratikakis
polyvios@cs.umd.edu

Jeffrey S. Foster
jfoster@cs.umd.edu

Michael Hicks
mwh@cs.umd.edu

DRAFT

Abstract

One common technique for preventing data races in multi-threaded programs is to ensure that all accesses to shared locations are consistently protected by a lock. We present a tool called LOCKSMITH for detecting data races in C programs by looking for violations of this pattern. We call the relationship between locks and the locations they protect *consistent correlation*, and the core of our technique is a novel constraint-based analysis that infers consistent correlation context-sensitively, using the results to check that locations are properly guarded by locks. We present the core of our algorithm for a simple formal language $\lambda_{\triangleright}$ which we have proven sound, and discuss how we scale it up to an algorithm that aims to be sound for all of C. We develop several techniques to improve the precision and performance of the analysis, including a sharing analysis for inferring thread locality; existential quantification for modeling locks in data structures; and heuristics for modeling unsafe features of C such as type casts. When applied to several benchmarks, including multi-threaded servers and Linux device drivers, LOCKSMITH found several races while producing a modest number of false alarms.

1 Introduction

Data races occur in multi-threaded programs when one thread accesses a memory location at the same time another thread writes to it. While some races are benign, those that are erroneous can have disastrous consequences [33, 40]. Moreover, race-freedom is an important program property in its own right, because race-free programs are easier to understand, analyze, and transform [4, 45]. For example, race freedom is necessary for reasoning about code that uses locks to achieve atomicity [17, 21].

In this paper, we present a static analysis tool called LOCKSMITH for automatically finding all data races in a C program. Our analysis aims to be sound so that any potential races are reported, modulo the unsafe features of C such as arbitrary pointer arithmetic and type casting. We check for data races by enforcing one of the most common techniques for race prevention: We ensure that for every shared memory location ρ there is some lock ℓ that is held whenever ρ is accessed. While this technique is not the only way to prevent races, it is common in multi-threaded software.

*University of Maryland, Computer Science Department Technical Report CS-TR-4789, UMIACS Technical Report UMIACS-TR-2006-13. This research was supposed in part by NSF CCF-0346989, CCF-0430118, and CCF-0524036.

Our goal is to produce a practical race-detection tool for C. A number of type systems have been developed for preventing races given specifications [5, 6, 14, 15, 24], but these can require considerable programmer annotations, limiting their practical application. Most completely-automatic static analyses have considered Java [2, 46, 18, 37, 30], thus avoiding many of the problematic features of C, such as type casts, low-level pointer operations, and non-lexically scoped locks. Those that consider C are either unsound, do not check certain idioms, or may have trouble scaling. A lengthy discussion of related work may be found in Section 5.

The core algorithm used by LOCKSMITH is an analysis that can automatically infer the relationship between locks and the locations they protect. We call this relationship *correlation*, and a key contribution of our approach is a new technique for inferring correlation context-sensitively. We present our correlation analysis algorithm for a formal language $\lambda_{\triangleright}$ that abstracts away some of the complications of operating directly on C code. Our analysis is constraint-based, using context-free language reachability [43, 44] and semi-unification [26] for context-sensitivity. Because each location must be consistently correlated with at least one lock, we use ideas from linear types to maintain a tight correspondence between abstract locks used by the static analysis and locks created at run time. We allow locks created in polymorphic functions to be treated distinctly at different call sites, and we use a novel type and effect system to ensure that this is safe.

In order to move from $\lambda_{\triangleright}$ to C, we use a number of additional techniques. One novel contribution is that we support *existential quantification*, which allows us to model correlations among fields of a data structure element, even after that element is merged into the "blob" typical of constraint-based alias analysis [10]. We use flow-sensitive analysis to model lock acquires and releases, which need not be lexically scoped. Our implementation includes a sharing analysis to model thread-local data, and uses heuristics to model type casts, including the special case of casts to and from `void *`. Finally, we use a lazy technique to efficiently model the large `struct` types typical of C programs.

We ran LOCKSMITH on a set of benchmarks, including programs that use POSIX threads and several Linux kernel device drivers. Our tool runs in seconds or minutes on our example programs, although for some other programs we have tried it does not complete due to resource exhaustion. LOCKSMITH found a number of data races, and overall produces few total warnings, making it easy to inspect the output manually. We also measured the effectiveness of the various analysis features mentioned above, and we found that all are useful, reducing the number of warnings in total by as much as a factor of three overall.

In summary, this paper makes the following contributions:

- We describe a context-sensitive correlation analysis for the language $\lambda_{\triangleright}$. Given a source program in $\lambda_{\triangleright}$, our analysis determines whether every memory location in the program is consistently correlated with a lock. Our analysis models locks linearly and uses a novel effect system to treat locks created in different calls to a function distinctly. (Section 2)

- We scale up our analysis to the C programming language with a series of additional techniques, including flow-sensitivity, existential quantification, and a sharing analysis to infer thread-local data. (Section 3)

- We evaluate our implementation on a small set of benchmarks. LOCKSMITH was able to find several races with few overall warning messages. (Section 4)

Although we focus on locking in this paper, we believe that the concept of correlation may be of independent interest. For example, a program may correlate a variable containing an integer

```
pthread_mutex_t L1 = ..., L2 = ...;
int x, y, z;

void munge(pthread_mutex_t *l, int *p) {
  pthread_mutex_lock(l);
  *p = 3;
  pthread_mutex_unlock(l);
}
...
munge(&L1, &x);
munge(&L2, &y);
munge(&L2, &z);
```

Figure 1: Locking Example in C

length with a array having that length [54]; it may correlate an environment structure with the closure that takes it as an argument [35]; or it may correlate a memory location with the *region* in which that location is stored [25, 27].

2 Race Freedom as Consistent Correlation

Consider the C program in Figure 1. This program has two locks, `L1` and `L2`, and three integer variables, `x`, `y`, and `z` (we omit initialization code for simplicity). The function `munge` takes a lock and a pointer and writes through the pointer with the lock held. Suppose that the program makes the three calls to `munge` as shown, and that this sequence of calls is invoked by two separate threads.

This program is race-free because for each location, there is a lock that is always held when that location is accessed. In particular, `L1` is held for all accesses to `x`, and `L2` is held for all accesses to both `y` and `z`. More formally, we say that a location ρ is *correlated* with a lock ℓ if at some point a thread accesses ρ while holding ℓ. We say that a location ρ and a lock ℓ are *consistently correlated* if ℓ is *always* held by the thread accessing ρ. Thus if all locations in a program are consistently correlated, then that program is race free.

Establishing consistent correlation is a two-step process. First, we determine what locks ℓ are held when the thread accesses some location ρ. Having gathered this information, we can then ask whether ρ is consistently correlated with some lock.

To simplify our presentation, we present the core of our algorithm for a small language $\lambda_{\triangleright}$ in which locations can be guarded by at most one lock (rather than a set of locks), and in which the lock correlated with a memory read or write is made explicit in the program text. This allows us to defer the problem of determining what locks are held at each dereference and focus on checking for consistent correlation. In Section 3, we describe how to extend our ideas to find data races in the full C programming language, including how to infer held lock sets at each program point.

2.1 The Language $\lambda_{\triangleright}$

Figure 2 presents the syntax of $\lambda_{\triangleright}$, a polymorphic lambda calculus extended with integers, comparisons, pairs, a primitive for generating mutual exclusion locks, and updatable references. We

$$
\begin{array}{rcl}
e & ::= & x \mid v \mid e_1\ e_2 \mid \texttt{if0}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\
& \mid & (e_1, e_2) \mid e.j \mid \texttt{let}\ f = v\ \texttt{in}\ e_2 \mid \texttt{fix}\, f.v \mid f^i \\
& \mid & \texttt{newlock} \mid \texttt{ref}\ e \mid !^{e_2} e_1 \mid e_1 :=^{e_3} e_2 \\
v & ::= & n \mid \lambda x.e \mid (v_1, v_2)
\end{array}
$$

Figure 2: $\lambda_{\triangleright}$ Syntax

```
let L1 = newlock in
let L2 = newlock in
let x = ref 0 in
let y = ref 1 in
let z = ref 2 in

let munge l p =
  p :=^l 3 in
munge^1 L1 x;
munge^2 L2 y;
munge^3 L2 z
```

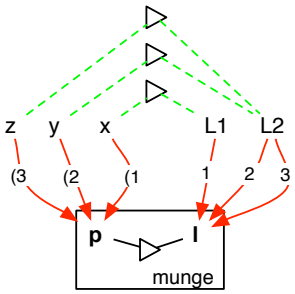


Figure 3: Locking example in $\lambda_{\triangleright}$ and its constraint graph

annotate function occurrences f^i with an *instantiation site* i, as in some other context-sensitive analyses [43]. Dereferences $!^e\, e_1$ and assignments $e_1 :=^e e_2$ take as an additional argument an expression e that evaluates to a lock, which is acquired for the duration of the memory access and released afterward. To keep the presentation simpler $\lambda_{\triangleright}$ does not include other language features such as recursive data structures, although those are handled by LOCKSMITH. The left side of Figure 3 gives the program in Figure 1 modeled in $\lambda_{\triangleright}$. The body of `munge` has been reduced to the expression $\texttt{p} :=^{\texttt{l}} 3$, indicating that `l` will be held during the assignment to `p`.

To check whether this program is consistently correlated, a natural approach would be to perform a points-to analysis for all of the pointers and locks in the program. At the assignment $\texttt{p} :=^{\texttt{l}} 3$ in the program, we could correlate all of the locations ρ to which `p` may point with the *singleton* lock ℓ to which `l` points. The lock `l` must point to a single ℓ or else some location ρ might be accessed sometimes with one lock and sometimes with another. Unfortunately, this condition is not satisfied in our example: the points-to set of `l` is $\{\texttt{L1}, \texttt{L2}\}$, since it will be `L1` at the first call to `munge` and `L2` at the second call. Thus our hypothetical algorithm would erroneously conclude that no single lock is held for all accesses, leading to false reports of possible races.

The problem is that correlation between `l` and `p` is not being treated *context-sensitively*. Even if we were to use a context-sensitive alias analysis [10], the points-to sets mentioned above would be the same, assuming that within the body of the function we summarized all calls, which is a standard technique.

We address this problem in two steps. First, we introduce *correlation constraints* of the form $\rho \triangleright \ell$, which indicate that the location ρ is correlated with the lock ℓ. Here, ρ and ℓ are location and lock *labels*, used to represent locations and locks that arise at run time. Our analysis generates correlation constraints based on occurrences of $!^e\, e_1$ and $e_1 :=^e e_2$ in the program. Second, we

formalize an analysis to propagate correlation constraints in a context-sensitive way throughout the program, by creating a variety of other (flow) constraints and solving them to determine whether correlations are consistent. We define consistent correlation precisely as follows.

Definition 1 (Correlation Set) *Given a location ρ and a set of constraints C, we define the* correlation set *of ρ in C as*

$$S(C, \rho) = \{\ell \mid C \vdash \rho \rhd \ell\}$$

Here we write $C \vdash \rho \rhd \ell$ to say that $\rho \rhd \ell$ can be proven from the constraints in C.

Definition 2 (Consistent Correlation) *A set of constraints C is* consistently correlated *if*

$$\forall \rho.\ |S(C, \rho)| \leq 1$$

Thus, a constraint set C is consistently correlated if all abstract locations ρ are either correlated with one lock, or are never accessed and so are correlated with no locks.

The right side of Figure 3 shows a graph of the constraints that our analysis generates for this example code. Each label in the program forms a node in the graph, and labeled, directed edges indicate data flow. Location flow edges corresponding to a function call are labeled with (i for the parameters at call site i, and any return values (not shown) are labeled with $)i$. Locks are modeled with unification in our system, and we label such edges simply with the call site, with the direction of the arrow into the type that was instantiated. For example, both `L1` and `x` are passed in at call site 1, so they connect to the parameters using edges labeled with (1. Undirected edges represent correlation. In this case, the body of `munge` requires that `l` and `p` are correlated.

After generating constraints we perform constraint resolution to propagate correlation constraints context-sensitively through the call graph. In this example, we copy `munge`'s correlation constraint out to each of the call sites, resulting in the three correlation constraints shown with dashed edges:

$$x \rhd L1 \qquad y \rhd L2 \qquad z \rhd L2$$

It is easy to see that these constraints are consistently correlated according to Definition 2.

2.2 Type System

We use a type and effect system for generating constraints C to check for consistent correlation. Our type system proves judgments of the form $C; \Gamma \vdash e : \tau; \varepsilon$, which means that expression e has type τ and effect ε under type assumptions Γ and constraint set C.

Figure 4 gives the type language and constraints used by our analysis. Types include integers, pairs, function types annotated with an effect ε, lock types with a label ℓ, and reference types with a label ρ. Effects are used to enforce linearity for locks (see below), and consist of the empty effect $\emptyset$, a singleton effect $\{\ell\}$, effect variables χ which are solved for during resolution, and both disjoint and non-disjoint unions of effects $\varepsilon \uplus \varepsilon'$ and $\varepsilon \cup \varepsilon'$, respectively. $\lambda_{\rhd}$ models context-sensitivity over labels using polytypes σ, introduced by `let` and `fix`. In our type language, polytype $(\forall.\tau, \vec{l})$ represents a universally quantified type, where τ is the base type and $\vec{l}$ is the set of *non*-quantified labels [26, 43]. Finally, C is a set of atomic constraints c. Within the type rules, the judgment $C \vdash c$ indicates that c can be proven by the constraint set C; in our algorithm, such judgments cause us to "generate" constraint c and add it C.

types	τ	::=	$int \mid \tau \times \tau \mid \tau \rightarrow^{\varepsilon} \tau' \mid lock\ \ell \mid ref^{\rho}\ \tau$	
labels	l	::=	$\ell \mid \rho$	
effects	ε	::=	$\emptyset \mid \{\ell\} \mid \chi \mid \varepsilon \uplus \varepsilon' \mid \varepsilon \cup \varepsilon'$	
polytypes	σ	::=	$(\forall.\tau, \vec{l})$	
constr. sets	C	::=	$\emptyset \mid \{c\} \mid C \cup C$	
constraints	c	::=	$\tau \leq \tau'$	(subtyping)
		$\mid$	$\ell = \ell'$	(lock unification)
		$\mid$	$\rho \leq \rho'$	(location flow)
		$\mid$	$\rho \rhd \ell$	(correlation)
		$\mid$	$\varepsilon \leq \chi$	(effect flow)
		$\mid$	$\varepsilon \leq_{\vec{l}} \chi$	(effect filtering)
		$\mid$	$effect(\tau) = \emptyset$	(effect emptiness)
		$\mid$	$\tau \preceq^{i}_{p} \tau'$	(type instantiation)
		$\mid$	$\ell \preceq^{i} \ell'$	(lock instantiation)
		$\mid$	$\rho \preceq^{i}_{p} \rho'$	(location inst.)
		$\mid$	$\varepsilon \preceq^{i} \chi$	(effect inst.)

Figure 4: Types and Constraints

Effects Effects ε form an important part of $\lambda_{\rhd}$'s type system by enforcing linearity for lock labels. Roughly speaking, a lock label ℓ is linear if it never represents two different run-time locks that could reside in the same storage or are simultaneously live. To understand why this is important, consider the following code, where hypothetical types and generated constraints are marked in comments, eliding the constraints for the references to locks. We use $e_1; e_2$ as the standard abbreviation for $(\lambda x.e_2)\ e_1$ where $x \notin fv(e_2)$.

```
let l = ref newlock in      // l : ref^ρ' (lock ℓ)
let x = ref 0 in            // x : ref^ρ int
  x :=^!l 1;                // ρ ▷ ℓ
  l := newlock;
  x :=^!l 2                 // ρ ▷ ℓ
```

This code violates consistent correlation because `x` is correlated with two different run-time locks due to the assignment. However, to give `l` a consistent type, ℓ is used to model both locks, violating linearity. As a result, the constraints mistakenly suggest the program is safe, because ρ is only ever correlated with ℓ.

We now turn to the monomorphic type rules for $\lambda_{\rhd}$, shown in Figure 5. The [Newlock] rule in this system requires that when we create a lock labeled ℓ we generate an effect $\{\ell\}$. The other rules, like [Pair], join the effects of their subexpressions with disjoint union $\uplus$, thus requiring that chosen lock labels not conflict. For example, with the given labeling, the above code has the effect $\{\ell\} \uplus \{\ell\}$. We implicitly require that disjoint unions are truly disjoint—during constraint resolution, we will check that this holds—and thus we would forbid `L1` and `L2` from being given the same label. On the other hand, location labels ρ, introduced in the rule [Ref] for typing memory allocation, do not add to the effect as memory locations need not be linear.

$$[\text{Id}]\frac{}{C;\Gamma, x:\tau \vdash x:\tau;\emptyset} \qquad [\text{Int}]\frac{}{C;\Gamma \vdash n:\mathit{int};\emptyset}$$

$$[\text{Lam}]\frac{\begin{array}{c}C;\Gamma, x:\tau \vdash e:\tau';\varepsilon \\ C \vdash \varepsilon \leq \chi \qquad \chi \text{ fresh}\end{array}}{C;\Gamma \vdash \lambda x.e:\tau \to^{\chi} \tau';\emptyset} \qquad [\text{App}]\frac{\begin{array}{c}C;\Gamma \vdash e_1:\tau \to^{\varepsilon} \tau';\varepsilon_1 \\ C;\Gamma \vdash e_2:\tau;\varepsilon_2\end{array}}{C;\Gamma \vdash e_1\ e_2:\tau';\varepsilon_1 \uplus \varepsilon_2 \uplus \varepsilon}$$

$$[\text{Pair}]\frac{C;\Gamma \vdash e_1:\tau_1;\varepsilon_1 \qquad C;\Gamma \vdash e_2:\tau_2;\varepsilon_2}{C;\Gamma \vdash (e_1,e_2):\tau_1 \times \tau_2;\varepsilon_1 \uplus \varepsilon_2} \qquad [\text{Proj}]\frac{C;\Gamma \vdash e:\tau_1 \times \tau_2;\varepsilon \qquad j=1,2}{C;\Gamma \vdash e.j:\tau_j;\varepsilon}$$

$$[\text{Sub}]\frac{C;\Gamma \vdash e:\tau_1;\varepsilon \qquad C \vdash \tau_1 \leq \tau_2}{C;\Gamma \vdash e:\tau_2;\varepsilon} \qquad [\text{Cond}]\frac{\begin{array}{c}C;\Gamma \vdash e_0:\mathit{int};\varepsilon_0 \\ C;\Gamma \vdash e_1:\tau;\varepsilon_1 \qquad C;\Gamma \vdash e_2:\tau;\varepsilon_2\end{array}}{C;\Gamma \vdash \texttt{if0}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2:\tau;\varepsilon_0 \uplus (\varepsilon_1 \cup \varepsilon_2)}$$

$$[\text{Newlock}]\frac{\ell \text{ fresh}}{C;\Gamma \vdash \texttt{newlock}:\mathit{lock}\ \ell;\{\ell\}} \qquad [\text{Ref}]\frac{C;\Gamma \vdash e:\tau;\varepsilon \qquad \rho \text{ fresh}}{C;\Gamma \vdash \texttt{ref}\ e:\mathit{ref}^{\rho}\ \tau;\varepsilon}$$

$$[\text{Deref}]\frac{\begin{array}{c}C;\Gamma \vdash e_1:\mathit{ref}^{\rho}\ \tau;\varepsilon_1 \\ C;\Gamma \vdash e_2:\mathit{lock}\ \ell;\varepsilon_2 \\ C \vdash \rho \rhd \ell\end{array}}{C;\Gamma \vdash !^{e_2}\ e_1:\tau;\varepsilon_1 \uplus \varepsilon_2} \qquad [\text{Assign}]\frac{\begin{array}{cc}C;\Gamma \vdash e_1:\mathit{ref}^{\rho}\ \tau;\varepsilon_1 & C;\Gamma \vdash e_2:\tau;\varepsilon_2 \\ C;\Gamma \vdash e_3:\mathit{lock}\ \ell;\varepsilon_3 & C \vdash \rho \rhd \ell\end{array}}{C;\Gamma \vdash e_1 :=^{e_3} e_2:\tau;\varepsilon_1 \uplus \varepsilon_2 \uplus \varepsilon_3}$$

Figure 5: $\lambda_{\rhd}$ Monomorphic Rules

Some other type-based systems for race detection [14, 24] and related systems for modeling dynamic memory allocation [49] avoid the need for this kind of effect by forcing newly-allocated locks (and/or locations) to be valid only within a lexical scope. That is, `newlock` is replaced with a construct `newlock` x `in` e, which at run time generates a new lock and substitutes it for x within e. When typing this construct, x's label ℓ is only valid in the expression e, ensuring the allocated lock cannot escape. Therefore subsequent invocations of the same `newlock` x `in` e (e.g., within a recursive function) cannot be confused. We can achieve the same effect using the [Down] rule, described below, and our approach matches the usage of `newlock` as it occurs in practice.

Typing Rules Turning to the remaining rules in Figure 5, [Id], [Int], and [Proj] are standard. [Lam] types a function definition, and the effect on the function arrow is the effect of the body. Notice that we always place effect variables χ on function arrows; this ensures constraints involving effects always have a variable on their right-hand side, simplifying constraint resolution. In [App] we apply a function e_1 to argument e_2, and the effect includes the effect of evaluating e_1, the effect of evaluating e_2, and the effect of the function body.

The [Sub] rule and subtyping rules, shown in Figure 7(a), are also standard. Note that in rule [Sub-Lock], we require ℓ and ℓ' to be equal. Thus we have no subtyping on lock labels, which makes it easier to enforce linearity by forcing lock labels that "flow" together to be unified. The rules in Figure 7(a) can be seen as judgments for reducing subtyping on types to constraints on labels, and during constraint resolution we assume that all subtyping constraints have been reduced in this way and thus eliminated.

[Cond] is mostly standard, except we use a non-disjoint union to join the effects of the two branches, since only one of e_1 or e_2 will be executed at run time. [Deref] accesses a location e_1

$$\text{[Let]}\ \frac{C;\Gamma \vdash v_1 : \tau_1;\emptyset \quad \vec{l} = \mathit{fl}(\Gamma) \quad C;\Gamma, f : (\forall.\tau_1,\vec{l}) \vdash e_2 : \tau_2;\varepsilon}{C;\Gamma \vdash \texttt{let}\ f = v_1\ \texttt{in}\ e_2 : \tau_2;\varepsilon} \qquad \text{[Inst]}\ \frac{C \vdash \tau \preceq^i_+ \tau' \quad C \vdash \vec{l} \preceq^i_\pm \vec{l}}{C;\Gamma, f : (\forall.\tau,\vec{l}) \vdash f^i : \tau';\emptyset}$$

$$\text{[Fix]}\ \frac{C;\Gamma, f : (\forall.\tau,\vec{l}) \vdash v : \tau';\emptyset \quad \vec{l} = \mathit{fl}(\Gamma) \quad C \vdash \tau' \leq \tau \quad C \vdash \tau \preceq^i_+ \tau'' \quad C \vdash \vec{l} \preceq^i_\pm \vec{l} \quad C \vdash \mathit{effect}(\tau) = \emptyset}{C;\Gamma \vdash \texttt{fix}\, f.v : \tau'';\emptyset} \qquad \text{[Down]}\ \frac{C;\Gamma \vdash e : \tau;\varepsilon \quad \vec{l} = \mathit{fl}(\Gamma) \cup \mathit{fl}(\tau) \quad C \vdash \varepsilon \leq_{\vec{l}} \chi \quad \chi \text{ fresh}}{C;\Gamma \vdash e : \tau;\chi}$$

Figure 6: $\lambda_\triangleright$ Polymorphic Rules (plus [Down])

while holding lock e_2, and generates a correlation constraint between the lock and location label, as does [Assign].

Polymorphism Figure 6 gives the rules for polymorphism. [Let] introduces polytypes. As is standard we only generalize the types of values. In [Let] the name f is bound to a quantified type where $\vec{l}$ is the set of free labels of Γ, i.e., the labels that cannot be generalized.

In [Inst], we use *instantiation constraints* to model a type instantiation. The constraint $\tau \preceq^i_+ \tau'$ means that there exists some substitution ϕ_i such that $\phi_i(\tau) = \tau'$, i.e., that at the use of f labeled by index i in the program, τ is instantiated to τ'. We also generate the constraint $\vec{l} \preceq^i_\pm \vec{l}$, which requires that all of the variables we could not quantify are renamed to themselves by ϕ_i, i.e., they are not instantiated.

The subscript $+$'s and $-$'s in an instantiation constraint are *polarities*, which represent the direction of subtyping through a constraint, either covariant ($+$) or contravariant ($-$). Instantiation constraints correspond to the edges labeled with parentheses in Figure 3. A constraint $\rho \preceq^i_+ \rho'$ corresponds to an output (i.e., a return value), and in constraint graphs we draw it as a directed edge $\rho \rightarrow^{)i} \rho'$. A constraint $\rho \preceq^i_- \rho'$ corresponds to an input (i.e., a parameter), and we draw it with a directed edge $\rho' \rightarrow^{(i} \rho$. We draw a constraint $\ell \preceq^i \ell'$ as an edge $\ell' \rightarrow^i \ell$, where there is no direction of flow since lock labels are unified but the arrow indicates the reverse direction of instantiation.

Instantiation constraints on types can be reduced to instantiation constraints on labels, as shown in Figure 7(b). In these rules we use p to stand for an arbitrary polarity, and in [Inst-Fun] we flip the direction of polarity for the function domain with the notation $\bar{p}$. For example, to generate the graph in Figure 3, we generated three instantiation constraints

$$\begin{array}{l}(\texttt{l} \times \texttt{p}) \rightarrow \mathit{int} \ \preceq^1_+ \ (\texttt{L1} \times \texttt{x}) \rightarrow \mathit{int} \\ (\texttt{l} \times \texttt{p}) \rightarrow \mathit{int} \ \preceq^2_+ \ (\texttt{L2} \times \texttt{y}) \rightarrow \mathit{int} \\ (\texttt{l} \times \texttt{p}) \rightarrow \mathit{int} \ \preceq^3_+ \ (\texttt{L2} \times \texttt{z}) \rightarrow \mathit{int}\end{array}$$

corresponding to the three instantiations and calls of `munge`. For full details on polarities, see Rehof et al [43].

Hiding Effects [Fix] introduces polymorphic recursion, which is decidable for label flow [36, 43]. However, in our system we instantiate effects, which because they contain disjoint unions may

$$[\text{Sub-Int}]\frac{}{C \vdash \mathit{int} \leq \mathit{int}} \qquad [\text{Sub-Pair}]\frac{C \vdash \tau_1 \leq \tau_1' \quad C \vdash \tau_2 \leq \tau_2'}{C \vdash \tau_1 \times \tau_2 \leq \tau_1' \times \tau_2'}$$

$$[\text{Sub-Lock}]\frac{C \vdash \ell = \ell'}{C \vdash \mathit{lock}\ \ell \leq \mathit{lock}\ \ell'} \quad [\text{Sub-Ref}]\frac{C \vdash \rho \leq \rho' \quad C \vdash \tau \leq \tau' \quad C \vdash \tau' \leq \tau}{C \vdash \mathit{ref}^{\rho}\ \tau \leq \mathit{ref}^{\rho'}\ \tau'}$$

$$[\text{Sub-Fun}]\frac{C \vdash \tau_2 \leq \tau_1 \quad C \vdash \tau_1' \leq \tau_2' \quad C \vdash \varepsilon_1 \leq \varepsilon_2}{C \vdash \tau_1 \to^{\varepsilon_1} \tau_1' \leq \tau_2 \to^{\varepsilon_2} \tau_2'}$$

(a) Subtyping

$$[\text{Inst-Int}]\frac{}{C \vdash \mathit{int} \preceq^{i} \mathit{int}} \qquad [\text{Inst-Pair}]\frac{C \vdash \tau_1 \preceq^{i}_{p} \tau_1' \quad C \vdash \tau_2 \preceq^{i}_{p} \tau_2'}{C \vdash \tau_1 \times \tau_2 \preceq^{i}_{p} \tau_1' \times \tau_2'}$$

$$[\text{Inst-Lock}]\frac{C \vdash \ell \preceq^{i} \ell'}{C \vdash \mathit{lock}\ \ell \preceq^{i}_{p} \mathit{lock}\ \ell'} \quad [\text{Inst-Ref}]\frac{C \vdash \rho \preceq^{i}_{p} \rho' \quad C \vdash \tau \preceq^{i}_{\pm} \tau'}{C \vdash \mathit{ref}^{\rho}\ \tau \preceq^{i}_{p} \mathit{ref}^{\rho'}\ \tau'}$$

$$[\text{Inst-Fun}]\frac{C \vdash \tau_1 \preceq^{i}_{\bar{p}} \tau_2 \quad C \vdash \tau_1' \preceq^{i}_{p} \tau_2' \quad C \vdash \varepsilon_1 \preceq^{i} \varepsilon_2}{C \vdash \tau_1 \to^{\varepsilon_1} \tau_1' \preceq^{i}_{p} \tau_2 \to^{\varepsilon_2} \tau_2'}$$

(b) Instantiation

Figure 7: Subtyping and Instantiation Constraints

grow without bound if a recursive function allocates a lock. Thus in [Fix], we require that recursive functions have an empty effect on their top-most arrow with the constraint $\mathit{effect}(\tau) = \emptyset$.

This is a strong restriction, and we would like to be able infer correlations for recursive functions that allocate locks. For example, consider the following two code snippets:

```
fix f.λx.                    let y = ref 0 in
  let l = newlock in         fix f.λx.
  let y = ref 0 in             let l = newlock in
    y :=^l 42;                   y :=^l 42;
    ... f 0 ...                  ... f 0 ...
```

Here f is a recursive function that creates a lock `l` and accesses a location `y`. In both cases the lock does not escape the function, and therefore the linear labels corresponding to the locks in different iterations of the function cannot interfere. However, in the second case the location `y` is allocated outside the function, meaning that with each iteration it will be accessed with a different lock held, violating consistent correlation. We want to allow the first case while rejecting the second.

Thus we add a final rule [Down] to our type system to hide effects on lock labels that are purely

local to a block of code [22]. In [Down], we generate a "filtering" constraint $\varepsilon \leq_{\vec{l}} \chi$, which means that χ should contain labels in ε that *escape* through $\vec{l}$, but not necessarily any other label. We determine escaping during constraint resolution. Formally, $C \vdash \mathit{escapes}(l, \vec{l})$, where l is either a ρ or ℓ, if

$$l \in \vec{l} \vee \exists c, l'. \left(C \vdash c \wedge l, l' \in c \wedge C \vdash \mathit{escapes}(l', \vec{l}) \right)$$

In other words, l escapes through $\vec{l}$ if it is in $\vec{l}$ or if it appears in a constraint in C with an l' that escapes in $\vec{l}$. For example, if $\rho \rhd \ell$ and ρ escapes, then ℓ escapes. This prevents `l` from being hidden in our second example above, while in the first example we can apply [Down] to hide the allocation effect successfully. Although [Down] is not a syntax-directed rule, it is only useful to apply it to terms whose effect may be duplicated in the type system. Hence we can make the system syntax-directed by assuming that [Down] is always applied once to e in rule [Lam], so that the effect on the function arrow has as much hidden as possible. Also note that we can easily encode the lexically-scoped lock allocation primitive `newlock` x `in` e as $(\lambda x.e)$ `newlock` and applying [Down] to the application.

Uses of [Down] are rare in C programs in our experience, which tend to use global locks. Some C programs also store locks in data structures, and in this case [Down] allows us to hide locks that are created and then packed inside of an existential type (Section 3.3) that contains the only reference to them.

2.3 Constraint Resolution

After we have applied the rules in Figures 5, 6, and 7 to a $\lambda_{\rhd}$ program, we are left with a set of constraints C. To check that a program is consistently correlated, we first reduce the constraints C into a *solved form*, from which we can easily extract correlations between locks and locations.

Figure 8 gives a series of left-to-right rewrite rules that we apply exhaustively to the constraints to compute their solution. Figure 8(a) gives rules to compute the "flow" of locations and locks; part (b) gives the rules for propagating correlations; and part (c) propagates effects so that we can check that disjoint unions are truly disjoint. The rules in part (a) are mostly standard, while parts (b) and (c) are new. Here, $C \cup \Rightarrow C'$ means $C \Rightarrow C \cup C'$.

The first rule of part (a) resolves equality constraints on lock labels and the second transitively closes subtyping constraints on location labels. The next rule is the standard semi-unification rule [26]: If a lock label ℓ_0 is instantiated at site i to two different lock labels ℓ_1 and ℓ_2, then ℓ_1 and ℓ_2 must be equal (because the substitution at site i has to substitute for ℓ_0 consistently). The final rule is for "matched flow." Recall the [Inst] rule from Figure 6: if f has polytype $(\forall.\, \mathit{ref}^{\rho_1}\ \tau_1 \rightarrow^{\emptyset} \mathit{ref}^{\rho_2}\ \tau_1, \emptyset)$, then instantiating this polytype at site i to the type $\mathit{ref}^{\rho_0}\ \tau_1 \rightarrow^{\emptyset} \mathit{ref}^{\rho_3}\ \tau_1$ requires that C contain instantiation constraints $\rho_1 \preceq^i_- \rho_0$ and $\rho_2 \preceq^i_+ \rho_3$ (according to [Inst-Fun] and [Inst-Ref]). The negative constraint corresponds to context-sensitive flow from the caller's argument to the function's parameter while the positive constraint corresponds to the returned value. Say that f is the identity function; then C would contain the constraint $\rho_1 \leq \rho_2$, indicating the function's parameter flows to its returned value. Thus the argument at site i should flow to the value returned at site i, and so the matched flow rule permits the addition of a flow edge $\rho_0 \leq \rho_3$. For a full discussion of this rule, see Rehof et al [43].

In the correlation propagation rules in part (b), the first rule says that if location ρ flows to a location ρ' that is correlated with ℓ, then ρ is correlated with ℓ also. Notice that there is no similar rule for flow on the right-hand side of a correlation, because we unify lock labels. The next

$$
\begin{array}{rcl}
C \cup \{\ell = \ell'\} & \Rightarrow & C[\ell \mapsto \ell'] \\
C \cup \{\rho_0 \leq \rho_1\} \cup \{\rho_1 \leq \rho_2\} & \cup \Rightarrow & \{\rho_0 \leq \rho_2\} \\
C \cup \{\ell_0 \preceq^i \ell_1\} \cup \{\ell_0 \preceq^i \ell_2\} & \Rightarrow & C[\ell_2 \mapsto \ell_1] \cup \{\ell_0 \preceq^i \ell_1\} \\
C \cup \{\rho_1 \preceq^i_- \rho_0\} \cup \{\rho_1 \leq \rho_2\} \cup \{\rho_2 \preceq^i_+ \rho_3\} & \cup \Rightarrow & \{\rho_0 \leq \rho_3\}
\end{array}
$$

(a) Flow of lock and location labels

$$
\begin{array}{rcl}
C \cup \{\rho \leq \rho'\} \cup \{\rho' \rhd \ell\} & \cup \Rightarrow & \{\rho \rhd \ell\} \\
C \cup \{\rho \preceq^i_p \rho'\} \cup \{\rho \rhd \ell\} \cup \{\ell \preceq^i \ell'\} & \cup \Rightarrow & \{\rho' \rhd \ell'\}
\end{array}
$$

(b) Correlation propagation

$$
\begin{array}{rcl}
C \cup \{\emptyset \leq \chi\} & \Rightarrow & C \\
C \cup \{\varepsilon \cup \varepsilon' \leq \chi\} & \Rightarrow & C \cup \{\varepsilon \leq \chi\} \cup \{\varepsilon' \leq \chi\} \\
C \cup \{\varepsilon \leq \chi\} \cup \{\chi \leq \chi'\} & \cup \Rightarrow & \{\varepsilon \leq \chi'\} \\
C \cup \{\varepsilon \leq \chi\} \cup \{\chi \leq_{\vec{l}} \chi'\} & \cup \Rightarrow & \{\varepsilon \leq_{\vec{l}} \chi'\} \\
C \cup \{\varepsilon \leq \chi\} \cup \{\chi \preceq^i \chi'\} & \cup \Rightarrow & \{\varepsilon \preceq^i \chi'\} \\
\hline
C \cup \{\emptyset \preceq^i \chi\} & \Rightarrow & C \\
C \cup \{\{\ell\} \preceq^i \chi\} & \Rightarrow & C \cup \{\ell \preceq^i \ell'\} \cup \{\{\ell'\} \leq \chi\} \\
 & & \quad \ell' \text{ fresh} \\
C \cup \{\varepsilon \uplus \varepsilon' \preceq^i \chi_0\} & \Rightarrow & C \cup \{\varepsilon \preceq^i \chi\} \cup \{\varepsilon' \preceq^i \chi'\} \\
 & & \quad \cup \{\chi \uplus \chi' \leq \chi_0\} \\
 & & \quad \chi, \chi' \text{ fresh} \\
C \cup \{\varepsilon \cup \varepsilon' \preceq^i \chi\} & \Rightarrow & C \cup \{\varepsilon \preceq^i \chi\} \cup \{\varepsilon' \preceq^i \chi\} \\
C \cup \{\chi \preceq^i \chi'\} \cup \{\chi \preceq^i \chi''\} & \Rightarrow & C[\chi' \mapsto \chi''] \cup \{\chi \preceq^i \chi''\} \\
\hline
C \cup \{\emptyset \leq_{\vec{l}} \chi\} & \Rightarrow & C \\
C \cup \{\{\ell\} \leq_{\vec{l}} \chi\} & \Rightarrow & C \cup \{\{\ell\} \leq \chi\} \\
 & & \quad \text{if } C \vdash \mathit{escapes}(\ell, \vec{l}) \\
C \cup \{\varepsilon \uplus \varepsilon' \leq_{\vec{l}} \chi_0\} & \Rightarrow & C \cup \{\varepsilon \leq_{\vec{l}} \chi\} \cup \{\varepsilon' \leq_{\vec{l}} \chi'\} \\
 & & \quad \cup \{\chi \uplus \chi' \leq \chi_0\} \\
C \cup \{\varepsilon \cup \varepsilon' \leq_{\vec{l}} \chi\} & \Rightarrow & C \cup \{\varepsilon \leq_{\vec{l}} \chi\} \cup \{\varepsilon' \leq_{\vec{l}} \chi\}
\end{array}
$$

(c) Effect propagation

Figure 8: Constraint Resolution

rule propagates correlations at instantiation sites. Similarly to location propagation, if we have a correlation constraint $\rho \rhd \ell$ on the labels in a polymorphic function, and we instantiate ℓ to ℓ' and ρ to ρ' at some site i, then we propagate the correlation to ℓ' and ρ'. For example, Figure 3 depicts the following three constraints, among others (recall an edge $l' \rightarrow^{(i} l$ in the figure corresponds to a constraint $l \preceq^i_- l'$):

$$\texttt{l} \preceq^1_- \texttt{L1} \qquad \texttt{p} \preceq^1_- \texttt{x} \qquad \texttt{p} \rhd \texttt{l}$$

Using our resolution rule yields the constraint $\texttt{x} \rhd \texttt{L1}$, shown in Figure 3 with a dashed line. Note that the polarity of the instantiation constraint on ρ is irrelevant for this propagation step, because locks can correlate with both inputs (parameters) and output (returns).

Part (c), presented as three blocks of rules, propagates effect constraints. The first block of rules discards useless effect subtyping, replaces standard unions by two separate constraints, and computes transitivity of subtyping on effects. The next block of rules handles instantiation constraints. The constraint $\emptyset \preceq^i \chi$ can be discarded, because it places no constraint on χ. (It is not even the case that χ must be empty, because it may have subtyping constraints on it from other effects.) In the next rule we model instantiation of a function with a single effect $\{\ell\}$. In our system, each time we call a function that invokes `newlock` we wish to treat the locks from different calls differently. Thus we create a fresh lock label ℓ' that flows to χ and require that ℓ is instantiated to ℓ'. The remaining rules copy disjoint unions across an instantiation site, expand non-disjoint unions, and require that effect variables are instantiated consistently.

The last block of rules propagates effects across filtering constraints. The only interesting rule is the second one, which propagates an effect $\{\ell\}$ to χ only if ℓ escapes in the set $\vec{l}$; this corresponds to "hiding" effects χ that are only used within a lexical scope.

After applying the rewrite rules, there are three conditions we need to check. First, we need to ensure that all disjoint unions formed during type inference and constraint resolution are truly disjoint. We define $\mathit{occurs}(\ell, \varepsilon)$ to be the number of times label ℓ occurs disjointly in ε:

$$\begin{array}{rcll}
\mathit{occurs}(\ell, \emptyset) & = & 0 & \\
\mathit{occurs}(\ell, \chi) & = & \max_{\varepsilon \leq \chi} \mathit{occurs}(\ell, \varepsilon) & \\
\mathit{occurs}(\ell, \{\ell\}) & = & 1 & \\
\mathit{occurs}(\ell, \{\ell'\}) & = & 0 & \ell \neq \ell' \\
\mathit{occurs}(\ell, \varepsilon \uplus \varepsilon') & = & \mathit{occurs}(\ell, \varepsilon) + \mathit{occurs}(\ell, \varepsilon') & \\
\mathit{occurs}(\ell, \varepsilon \cup \varepsilon') & = & \max(\mathit{occurs}(\ell, \varepsilon), \mathit{occurs}(\ell, \varepsilon')) &
\end{array}$$

We require for every effect ε created during type inference (including constraint resolution), and for all ℓ, that $\mathit{occurs}(\ell, \varepsilon) \leq 1$. We enforce the constraint $\mathit{effect}(\tau) = \emptyset$ by extracting the effect ε from the function type τ and ensuring that $\mathit{occurs}(\ell, \varepsilon) = 0$ for all ℓ.

Finally, we ensure that locations are consistently correlated with locks. We compute $S(C, \rho)$ for all locations ρ and check that it has size ≤ 1. This computation is easy with the constraints in solved form; we simply walk through all the correlation constraints generated in Figure 8(b) to count how many different lock labels appear correlated with each location ρ.

We now analyze the running time of our algorithm for each part of constraint resolution. Let n be the number of constraints generated by walking over the source code of the program. Then the rules in Figure 8(a) take time $O(n^3)$ [43], as do the rules in Figure 8(b), since given n constraints there can be only $O(n^2)$ correlations among locations and locks mentioned in the constraints. Constraint resolution rules like those given in parts (a) and (b) have been shown to be efficient in practice [10].

There exist constraint sets C for which the rules in Figure 8(c) will not terminate. This is because a cycle in the instantiation constraints might result in a single effect being repeatedly copied and renamed. We believe that this cannot occur in our type system, however, because we forbid recursive functions from having effects. Even so, effect propagation can still be $O(2^n)$, because a single effect might be copied through a chain of instantiations that double the effect each time.

2.4 Soundness

We have proven that a version of our type system $\lambda_{\triangleright}^{cp}$ based on polymorphically constrained types [36] is sound, and that the system presented here reduces to that system. We define a call-by-value operational semantics as a series of rewriting rules, using evaluation contexts $\mathbb{E}$ to define evaluation order, as is standard. The evaluation rule for `newlock` generates a fresh lock constant L, and `ref` v generates a fresh location constant R. We extend labels l to include L and R and define typing rules for them. We also introduce *allocation constraints* $L \leq^1 \ell$ to indicate that lock variable ℓ has been allocated as constant L. We then refine $S(C, \rho)$ to $S_g(C, \rho)$, which only refers to concrete lock labels:

$$S_g(C, \rho) = \{L \mid C \vdash \rho \triangleright \ell \wedge C \vdash L \leq^1 \ell\}$$

Thus $S_g(C, \rho)$ is the set of concrete locks correlated with ρ in C.

Next we define valid evaluation steps, which are those such that if a location R is accessed with lock L, then $L \in S_g(C, R)$.

Definition 3 (Valid Evaluation) *We write $C \vdash e \longrightarrow e'$ iff $e \equiv \mathbb{E}[!^{[L]}\, v^R]$ or $e \equiv \mathbb{E}[v'^R :=^{[L]} v]$ implies $S_g(C, R) = \{L\}$.*

Notice that definition enforces consistent correlation: a concrete R must be associated only with a single concrete lock L by S_g. We define an auxiliary judgment $\varepsilon \vdash_{ok} C$, which holds if in C all locations are consistently correlated and no lock labels in ε have been allocated.

We write $\vdash_{cp}$ for the type judgment in $\lambda_{\triangleright}^{cp}$. We then show preservation, which implies soundness.

Lemma 4 (Preservation) *If $C; \Gamma \vdash_{cp} e : \tau; \varepsilon$ where $\varepsilon \vdash_{ok} C$ and $e \longrightarrow e'$, then there exists some C', ε', such that $(\varepsilon' - \varepsilon) \cap \mathit{fl}(C) = \emptyset$; and $C' \supseteq C$; and $C' \vdash e \longrightarrow e'$; and $\varepsilon' \vdash_{ok} C'$; and $C'; \Gamma \vdash_{cp} e' : \tau; \varepsilon'$.*

(The proof is by induction on $C; \Gamma \vdash_{cp} e : \tau; \varepsilon$.) This lemma shows that if we begin with a consistently correlated constraint system and take a step for an expression e whose effect is ε, then the evaluation is valid. Moreover, there is some consistently correlated C' that entails C, where C' may contain additional constraints if the evaluation step allocated any locks or locations. Notice that since $C' \supseteq C$ (and thus $C' \vdash C$), any correlations that hold in C also hold in C'. Since at each evaluation step we preserve existing correlations and maintain consistent correlation, a well-typed program is always consistently correlated—each location R to a single lock L—during evaluation.

Finally, we can prove that we can reduce judgments in $\lambda_{\triangleright}$ to $\lambda_{\triangleright}^{cp}$. This reduction-based proof technique follows Fähndrich et al [13].

Lemma 5 (Reduction) *Given a derivation of $C; \Gamma \vdash e : \tau; \varepsilon$, then $C^*; \Gamma^* \vdash_{cp} e : \tau; \varepsilon^*$.*

where C^* is the set of constraints closed according to the rules in Figure 8(a) and (b), ε^* is the set of locks in ε according to the rules in Figure 8(c), and Γ^* is a translation from $\lambda_\triangleright$ to $\lambda_\triangleright^{cp}$ type assumptions.

Full proofs can be found in the Appendix.

3 Locksmith: Race Detection for C

Locksmith applies the ideas of Section 2 to the full C programming language. We implemented Locksmith using CIL [38] as a C front-end and using BANSHEE [32] to encode portions of the constraint graph and to apply the resolution rules in Figure 8(a). We use our own constraint solver for the rest of the analysis.

Locksmith is structured as a set of modules implementing different phases of the analysis. The first phase traverses source code and generates constraints akin to $\lambda_\triangleright$ constraints. However, while $\lambda_\triangleright$ programs specify a correlation between a lock and a location explicitly, in C such correlations must be inferred. Using some additional constraint forms, Locksmith infers which locks are held at each program point, and generates correlations accordingly to detect potential races. As an optimization, Locksmith includes a middle phase to compute which locations are always thread-local and therefore can be ignored for purposes of checking correlation. Locksmith also includes two additional features to improve precision for C. We support existential types, to model locks stored in data structures, and we try to model pointers to `void` precisely and structures efficiently.

3.1 Flow-Sensitive Race Detection

Locksmith extends $\lambda_\triangleright$ type judgments to include *state variables* ψ [22] to model the flow-sensitive events needed to infer correlations. Judgments include both an input and an output state variable, representing the point just before and just after, respectively, execution of the expression. Function types also have an input and output ψ, to represent the initial and final states of the function. Control flow from state ψ to state ψ' is indicated by a *control flow constraint* $\psi \leq \psi'$, and we also include instantiation on states, written $\psi \preceq_p^i \psi'$. Each state is assigned a *kind* that describes how that state differs from preceding states.

As an example, the typing rule for acquiring a lock is

$$[\text{Acquire}]\frac{\begin{array}{c}\psi; C; \Gamma \vdash e : \mathit{lock}\ \ell; \psi'; \varepsilon \qquad \psi'' \text{ fresh} \\ C \vdash \psi' \leq \psi'' \qquad C \vdash \psi'' : \texttt{Acquire}(\ell)\end{array}}{\psi; C; \Gamma \vdash \texttt{acquire}\ e : \mathit{int}; \psi''; \varepsilon}$$

This rule says that to infer a type for `acquire` e beginning in state ψ, we infer a labeled lock type for e, whose evaluation produces the state ψ'. We create a new state ψ'' that immediately follows ψ' and in which ℓ is acquired. A similar rule for `release` e annotates states with kind $\texttt{Release}(\ell)$, and a rule for dereferences and assignments annotates states with kind $\texttt{Deref}(\rho)$ for reads of writes to location ρ. An example of control-flow constraints is shown below in Section 3.3.

Computing Held Locks Given a control-flow constraint graph, Locksmith computes the locks held at each program point represented by a state ψ. Assume for the moment that all locks are linear. Then at a node ψ such that $C \vdash \psi : \texttt{Acquire}(\ell)$, the lock ℓ is clearly held. We iteratively propagate this fact forward through constraints $\psi \leq \psi'$ (and likewise for instantiation constraints)

stopping propagation at any node ψ for which $C \vdash \psi : \texttt{Release}(\ell)$. At joins we intersect the sets of acquired locks. This continues until we reach a fixed point.

In essence this analysis computes the set of locks that *must* be acquired at each program point. Notice that because the analysis is necessarily conservative, we may decide at a program point that lock ℓ is not held even if it is at run time. This is safe because if our analysis inaccurately determines that a lock is released, at worst it will report a data race where no race is possible.

At function calls, denoted by another kind of ψ variable, we "split" the set of locks. At a split, we propagate the state of ℓ to the function's input state only if that function actually changes the state of (acquires or releases) ℓ, since otherwise the function must be polymorphic in ℓ's state. (Which locks are (transitively) mentioned by a function is determined by a standard, context-sensitive effect analysis.) The state of other locks is added to the output state of the function upon return. This is similar to *Merge* nodes in CQual [22]. Crucially, this optimization ensures we do not conflate lock states at calls to library functions such as `printf`. At instantiation sites $\psi \preceq^i_p \psi'$, we use the renaming defined by $\preceq^i_+$ to copy the states of any locks in the domain of the substitution corresponding to i from ψ to ψ', and vice-versa for $\preceq^i_-$ constraints.

Inferring Correlations and Finding Races Now, for each state variable ψ of kind $\texttt{Deref}(\rho)$, we generate a correlation constraint $\rho \rhd \{\ell_1, \ldots, \ell_n\}$, where the ℓ_i are the set of locks held at ψ. (We have extended correlation constraints to include a set of locks rather than a single lock.)

Given the correlation constraints, we could apply the rules in Figure 8(b) to infer all correlations. However, because we "split" lock states at function calls, this would result in many false alarms, since a correlation constraint $\rho \rhd \{\ell_1, \ldots, \ell_n\}$ generated inside a function really means that locks ℓ_i are held *in addition* to any locks held at the function's callers. Thus in LOCKSMITH, rather than inserting each correlation constraint into a global set C, we define a family of constraint sets C_ψ, one per ψ, and propagate them backwards along the control-flow constraint graph. When we reach a split node in the constraint graph, we add to each correlation constraint any held locks that were split off previously.

When we are done propagating, we check for consistent correlation among the correlation constraints $C^{\texttt{main}}_\psi$ which correspond to the initial state ψ of `main()`. As correlation constraints now refer to lock sets, we define

$$S(C, \rho) = \{\{\ell_1, \ldots, \ell_n\} \mid C \vdash \rho \rhd \{\ell_1, \ldots, \ell_n\}\}$$

A location labeled ρ is consistently correlated if

$$|\bigcap S(C, \rho)| \geq 1$$

i.e., if there is at least one lock held every time ρ is accessed.

In the discussion thus far we have assumed that all locks are linear, but as per discussion in Section 2 this could be unsound. Rather than forbid non-linear locks, in LOCKSMITH we treat them as always released. Therefore non-linear locks are never included in correlation constraints, and so they do not prevent races from being reported. Our implementation currently allows most linearity checking to be optionally disabled, as we have found it is not very helpful in practice and can have a steep performance penalty. Our implementation also omits the right disjunct of the $\mathit{escapes}(\ell, \vec{l})$ check used for [Down] because it is not supported by BANSHEE. While possible, it is highly improbable that this omission will result in missed races.

```
int x;

void *f(...) {
  int *p = (int *) malloc(...);
  *p = x;
}

int main(void) {
  x = 42;
  pthread_create(..., f, ...);
  pthread_create(..., f, ...);
}
```

Figure 9: Example of Sharing Analysis

3.2 Shared Locations

As an optimization, LOCKSMITH generates correlation constraints at states $\texttt{Deref}(\rho)$ only when ρ may be thread-shared. Thus thread-local data need not be consistently correlated, which in practice substantially improves the precision and efficiency of LOCKSMITH.

We use several techniques to infer sharing. Our core technique is based on *continuation effects*. In the standard approach, the effect of an expression e denotes those locations read and written by e. In our approach, each expression has both input and output effects, ϵ_i and ϵ_o, where ϵ_o denotes the locations read and written in the program executed after e (including forked threads), while ϵ_i contains ϵ_o and those locations read and written in e itself. We compute continuation effects context-sensitively using BANSHEE.

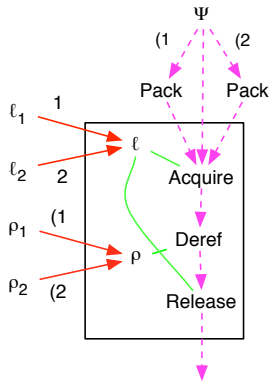
When a new thread is created, we determine the locations it might share with its parent (or other threads the parent forks) as follows. Let ϵ_t be the input effect of the child thread, and let $\epsilon*$ be the *input closure* of a continuation effect ϵ, defined as all those locations ρ' that could flow to locations $\rho \in \epsilon$. Then $S = \epsilon_t^* \cap \epsilon_o^*$ is the possibly-shared locations due to the fork, where ϵ_o is the output effect of the parent. We prune S further to only mention ρ if it is written in either ϵ_t^* or ϵ_o^* (so that read-only access is not a race). For each $\texttt{Deref}(\rho)$ state, we generate a correlation constraint for ρ if

$$\rho^* \cap \bigcup_{\text{all } S} S \neq \emptyset$$

We make two improvements to this basic technique. First, rather than intersect ρ^* with *all* S, we consider only those S due to the forking of the current thread, its ancestors, or any child threads created by the current thread prior to the dereference of ρ; all dereferences in `main` prior to forking the first thread are considered unshared. This allows data to be accessed thread locally without protection, and only once it becomes shared must it be consistently correlated. Second, we apply a *Down-Fork* rule to further filter from S locations that do not escape a forked thread, and thus cannot be shared with its parent. In particular, suppose we spawn a thread t that may access location ρ. Then we observe that if $\rho^* \cap fl(\Gamma)^* = \emptyset$, where $fl(\Gamma)$ is the set of free labels in the types of variables visible at the point of the fork, then ρ is not visible outside the child thread and thus cannot be shared.

```
let m = newlock^ℓ1 in
let x = ref^ρ1 1 in
let p =
  if0 b then
    pack^1 (m, x)
  else
    pack^2 (newlock^ℓ2, ref^ρ2
in
  unpack (l, r) = p in
    acquire l;
    r := 3;
    release l
```

(a) Source code



(b) Constraint graph

Figure 10: Existential Quantification

To see the benefit of these techniques, consider the code in Figure 9. This program initializes a global variable `x` and then forks two threads (using `pthread_create`) that invoke the function `f`, which reads `x` and writes it to freshly-allocated storage. Sharing analysis determines that `x` is never written after it becomes shared, and hence does not require consistent correlation. Also notice that both copies of `f` allocate a location at the same syntactic position in the program. Thus our analysis assigns both allocations the same location ρ. A naive analysis would determine that ρ is shared because it is accessed by both child threads. Using Down-Fork, however, we observe that ρ does not escape the body of `f` and hence is thread-local.

Finally, our implementation also includes a *uniqueness* analysis. We perform a very basic, intraprocedural, flow-sensitive alias analysis to determine when local variables definitely point to thread-local memory. For example, consider the following code:

```
int* x = (int *) malloc(sizeof(int));
*x = 2;
lock(l);
shared = x; /* becomes shared */
```

Here `x` points to newly-allocated memory that is subsequently initialized. Then `x` is assigned to `shared`—a variable visible to another thread—after acquiring lock `l`. The assignment causes `*x` to be an alias of `shared`; if this code occurs in a routine run by multiple threads, our earlier sharing analysis will think that `x` is always thread-shared. But our local uniqueness analysis observes that at the write to `*x`, the variable `x` has not yet escaped, and hence the write is ignored for purposes of correlation.

3.3 Existential Quantification for Data Structures

In applying our system to C programs, we found several examples where locks are stored in heap data structures along with the data they protect. Standard context-sensitive analyses typically merge all elements of the same data structure into an indistinguishable "blob," which would cause us to lose track of the identities of locations and the linearities of locks in data structures. In this

subsection we briefly sketch an approach to solving this problem that has proven effective for one of our benchmarks.

As an example, consider the program in Figure 10(a). This program first binds m to a new lock labeled ℓ_1, and then binds x to a new reference labeled ρ_1 (here for convenience we mark labels in the source code directly). The program then sets p to be one of two pairs. The *pack* operation alerts our analysis that the pairs should be treated abstractly so that we can conflate them without losing correlations. Next the program *unpacks* p and acquires the pair's lock before dereferencing its pointer.

Notice that although r may be either ρ_1 or ρ_2 at runtime, and l may be either ℓ_1 or ℓ_2, in either case the correct lock will be acquired. Because we used *pack* before the data structure was conflated, our analysis gives p the type

$$\exists \ell, \rho[\rho \rhd \{\ell\}].\, lock\ \ell \times ref^{\rho}\ int$$

meaning that p contains some lock ℓ and some location ρ where ℓ and ρ are correlated.

One key novelty of LOCKSMITH is that, given a program with `pack` and `unpack` annotations, we perform *inference* on existential types using constraint resolution rules similar to those in Figure 8. Figure 10(b) shows the constraint graph for our example. Rather than give resolution rules explicitly, we discuss the algorithm informally on this example. Existential inference using this basic technique is sound for the related problem of label flow [41].

In this figure, we represent data flow from labels ℓ_i and ρ_i to the packed labels ℓ and ρ with directed edges annotated with the `pack` site. It is no coincidence that this is the same notation used for universal quantification in Figure 3—it is the duality of universal and existential quantification that lets us use similar techniques for both. The remaining edges show the states at the various program points. Initially we are in some state ψ. Then we pack one of the two pairs, represented by a split labeled with (i for pack site i. Within the unpack (shown in the box), we acquire lock l, dereference r, and then release l. At the dereference site, lock l is held, and so we generate a constraint $r \rhd \{l\}$ (not shown in the graph). We propagate this correlation constraint using matched flow as in Figure 8(b) and generate two constraints, $\rho_1 \rhd \{\ell_1\}$ and $\rho_2 \rhd \{\ell_2\}$. Had we not used existential quantification here, we would not have been able to track correlation precisely, because ℓ_1 and ℓ_2 would have been non-linear, and there would have been no way to tell which goes with ρ_1 and which goes with ρ_2.

LOCKSMITH supports existential types for `structs`. To use existentials, the programmer annotates aggregates that can be packed to indicate which fields should have bound types after packing. We extend C with a special `pack(x)` statement that makes `x`'s type existentially quantified. For unpacking, the programmer inserts `start_unpack(x)` and `end_unpack(x)` statements, which begin and end the scope of the unpack, possibly non-lexically. In Section 4, we show that existential quantification is useful for one of our benchmarks. We needed to add a total of 29 pack, unpack, and field annotations to the program that could benefit, and 3 of the 12 `start_unpack` operations are not lexically scoped.

3.4 Analysis of `void*` and Aggregates

We aim to be sound, and so we use a number of techniques to conservatively model the unsafe aspects of C without losing too much precision. At type casts of dissimilar types we conflate locations in the actual and cast-to type. However, for `void*` pointers, we instead maintain a set of

Benchmark	Size (KLOC)	Time	Warn.	Unguarded	Races
aget	1.6	0.8s	15	15	15
ctrace	1.8	0.9s	8	8	2
pfscan	1.7	0.7s	5	0	0
engine	1.5	1.2s	7	0	0
smtprc	6.1	6.0s	46	1	1
knot	1.7	1.5s	12	8	8

Table 1: Summary of Experimental Results: POSIX Apps

non-`void*` types that are cast to or from them [31]. Any type cast to or from that `void*` at the same type is unified with the matching type stored in the `void*`. This technique enables us to model the common case when `void*` is used for polymorphism, which is important because the POSIX `pthread_create` routine takes a `void*` argument that is passed into the function called when the new thread starts. However, using this model of `void*` is unsound, because it does not handle up- or downcasts of `struct` types and assumes programmers use `void*` safely. Nevertheless, we have found it effective in practice, and it makes the output of LOCKSMITH much easier to interpret by reducing conflation.

Some `struct` types in C programs may have many fields (we have seen cases of 100 or more), many of which themselves have `struct` types containing many fields. For precision, we wish to assign fresh labels to fields of different instances of the same `struct` type. However, if we model these types naively by representing all fields of all instances of aggregates, then the analysis becomes very inefficient. Instead, we represent structure fields lazily [31], so that we only model those fields that are actually used. This optimization does not affect precision, but can provide a significant speedup.

4 Experiments

We evaluated LOCKSMITH on a modest set of benchmarks, including several small applications and medium-sized Linux kernel drivers. We conducted our experiments on a dual Xeon@2.8GHz PC with 3.5GB of RAM, running RedHat Enterprise Linux, kernel version 2.4.21. LOCKSMITH was compiled using OCaml 3.08.1, with all C code (BANSHEE and the OCaml runtime system) compiled with `gcc` 3.2.3-53 at optimization level `-O2`. Reported elapsed times are the median of 7 runs.

4.1 POSIX Threads Applications

We selected several multi-threaded programs largely gathered from `sourceforge.net`. *Aget* is an FTP client in which multiple threads download chunks of a file. *Ctrace* is a library for tracing the execution of multi-threaded programs; we analyzed a sample application that came with its distribution. *Pfscan* is a multithreaded file scanner that combines the functionality of `find`, `xargs`, and `fgrep`. *Engine* issues requests to several search engines in parallel and collates the responses. *Smtprc* is an open mail relay scanner that looks for potential configuration problems. Finally, *knot* is a multi-threaded webserver distributed with the Capriccio user-level threads package [51].

In our experiments we measured the number of warnings reported by LOCKSMITH and how

many of those warnings correspond to races. Here is a somewhat simplified warning taken from aget:

```
Possible data race on
  &bwritten(aget_comb.c:943)
References:
  dereference at aget_comb.c:1079
  locks acquired at dereference:
    &bwritten_mutex(aget_comb.c:996)
  in: FORK at aget_comb.c:468 ->
    http_get aget_comb.c:468

  dereference at aget_comb.c:984
  locks acquired at dereference:
    (none)
  in: FORK at aget_comb.c:193 ->
    signal_waiter(aget_comb.c:193) ->
    sigalrm_handler(aget_comb.c:957)
```

The first part indicates where the data that might be accessed in race is allocated, in this case the global variable `bwritten` defined at line 943. The second part lists where that location may be dereferenced, along with the locks held at that point and the context-sensitive control-flow path that led to the dereference. Above we show two (out of many) accesses. The first is in a thread running the function `http_get` with the mutex `&bwritten_mutex` held. The second access is in another thread running `signal_waiter`, which has called the function `sigalrm_handler`; this takes place with no lock held, violating consistent correlation. In practice this situation could arise when the user terminates aget abruptly with a signal, which causes it to save its current state to disk. The race on `bwritten` could cause it to be confused when the program restarts.

Table 1 shows the experimental results for these application programs. We merge multi-file programs into a single C file using the CIL merger, which eliminates duplicate and unused declarations. The table lists the size of the merged program in lines of preprocessed code. Since the application programs use the standard C library, we constructed a stub file containing function definitions that model the data flow and effects of libc routines used in our benchmarks, totaling roughly 400 LOC (not counted in the table).

The third column shows the total number of warnings of potential races issued by Locksmith. The "Unguarded" column lists the number of warnings that constitute true violations of consistent correlation. Some of these may not be races because a shared location is protected using other techniques. The last column lists the number of true races, in which data could be accessed simultaneously by two threads and one of the accesses is a write.

For aget, most of the races are similar to the example above. For knot, all the races seem to be benign; most are due to unprotected accesses to global variables used to gather statistics. Ctrace uses semaphores to communicate among threads, so while 6 guarded-by violations reported are legitimate, the data is not subject to races. The two real races are on global variables; one is benign, but the other is used to communicate information between threads, and a race could cause messages to be lost. Finally, smtprc has one race that occurs when a reaper thread sets a global counter that is also set and read by the main thread, which could cause unpredictable behavior.

Locksmith also reported a number of false alarms, mostly arising from two coding idioms that Locksmith does not handle. One is when a parent thread accesses previously shared data after its

Benchmark	Size (KLOC)	Time	Warn.	Unguarded	Races
plip	19.1	24.9s	11	2	1
eql	16.5	3.2s	3	0	0
3c501	17.4	240.1s	24	2	2
sundance	19.9	98.2s	3	1	0
sis900	20.4	61.0s[1]	8	2	1
slip	22.7	16.5s[1]	19	1	0
hp100	20.3	31.8s[1]	23	2	0

Table 2: Summary of Experimental Results: Linux Drivers

child threads have died (3 for engine, 1 for pfscan), as determined by `pthread_mutex_join` or other signaling mechanisms. Another is when a global data structure points to thread-local data, indexed by thread identifier (4 in engine, 44 in smtprc). The remaining false alarms could be handled with some improvements to the local sharing analysis (3 for knot), and to allowable idioms of existential initialization (3 for pfscan).

We also tried to run LOCKSMITH on several larger programs, but were ultimately unsuccessful due to resource exhaustion. We do not believe these problems are fundamental, and plan to continue to investigate how LOCKSMITH can be applied to larger programs.

4.2 Device Drivers

In addition to application code, we applied LOCKSMITH to a set of Linux device drivers. We found that determining synchronization assumptions for device drivers is challenging because the internal Linux API is complex and sparsely documented. Complicating matters, earlier versions of the kernel used a single spin lock ("the big kernel lock" or BKL) to prevent parallel access within the kernel, and remnants of this discipline remain. For example, as far as we can tell, character driver operations are always called with the BKL held, removing the need for multi-processor synchronization.

Therefore, we chose to apply LOCKSMITH to network device drivers, which must use internal locking and are relatively well-documented. We focused on seven drivers from the 2.6.12 kernel: plip (parallel-line IP), slip (serial line IP), eql (network traffic equalizer), and sis900, 3c501, hp100 and sundance (ethernet card drivers). We constructed stub files with a special `main` routine that simulates the kernel's concurrent interaction with the driver via interrupts, timeouts, and user process-induced calls. LOCKSMITH models kernel spin locks in the same way it models POSIX mutexes. We ran LOCKSMITH with some linearity checking disabled, because currently our semi-unification algorithm does not terminate for some drivers. We believe this can be fixed by implementing an extended occurs check [26].

Table 2 lists our results. We found a total of 4 races. The races in plip and sis900 are both benign races on counters. One race in 3c501 is a presumably benign race on a debugging flag. The other race in 3c501 occurs on a flag that tracks whether the driver is transmitting. We believe a race on this flag could cause errors if it occurs in the middle of a send operation. The guarded-by violations that were not races were due to the use of *atomic operations* which are always thread safe. These are implemented with inline assembly code that LOCKSMITH processes conservatively.

[1]Did not perform linearity checking

The main source of false alarms in drivers is due to conflation and other conservatism due to type casts. This causes LOCKSMITH mistakenly to think locations could be shared when they are not. Several of the false alarms could be addressed using existentials, in principle, to model a lock stored in a data structure, but we currently cannot check the initialization pattern. Finally, in many cases synchronization is context dependent, employing state variables and other non-lock-based forms. It was difficult to tell in these cases whether a race existed or not, since we are not kernel experts, so we considered them to be non-races in the table.

4.3 Per-feature Effectiveness

We examined the various features described in Section 3 for improving the precision of the analysis. Table 3 shows the number of warnings issued by the tool depending on which techniques are enabled, along with the corresponding running time. In particular, we measured the cumulative effectiveness of four techniques: (1) our technique for modeling a `void*` cast to/from a single type precisely (Void), as compared to conflating all locations at all levels of a type cast to `void*`; (2) the use of Down-Fork to reduce false sharing (DownF); (3) flow-sensitive uniqueness analysis of local variables (Uniq); and (4) using existential quantification to model locks local to data structures (Exist), which only affects the knot benchmark. All but the last feature are fully automatic, while existentials require manual insertion of packs and unpacks; we used 29 annotations for knot.

For a more visual comparison, we show the normalized effect of each technique on precision in Figure 11. The non-black portion of each bar is the scaled improvement due to the addition of that particular feature. For example, we can see that for aget, 60% (26 of 43) warnings were eliminated due to precise handling of `void*` while an additional 4.5% (2 of 43) were removed due to Down-Fork. Void and DownF are clearly the most useful overall. In contrast to Void and DownF, Exists and Uniq are useful only in a few cases, and may increase running time.

5 Related Work

A number of systems have been developed for detecting data races and other concurrency errors in multi-threaded programs, including dynamic analysis, static analysis, and hybrid systems.

Dynamic systems such as Eraser [47] instrument a program to find data races at run time and require no annotations. The efficiency and precision of dynamic systems can be improved with static analysis [8, 39, 1]. Dynamic systems are fast and easy to use, but cannot prove the absence of races, and require comprehensive test suites.

Researchers have developed type checking systems against races [14] for several languages, including Java [15], Java variants [6], and Cyclone [24]. In general, systems based on type checking perform very well, but require a significant number of programmer annotations, which can be time consuming when checking large code bases [11, 16]. Static race detection in ESC/Java [20],which employs a theorem prover, similarly requires many annotations.

Some researchers have developed tools to automatically infer the annotations needed by the Java-based type checking systems just mentioned. Most target Java 1.4, which simplifies the problem by permitting only lexically-acquired locks via `synchronized` statements, whereas C (and Java 1.5) programs may acquire and release locks at any program point. Houdini [16] can infer types for the original race-free Java system [15], but lacks context-sensitivity. More recently Agarwal and Stoller [2] and Rose et al [46] have developed algorithms that infer types based on dynamic traces,

Benchmark	All off	+Void	+DownF	+Uniq	+Exist
aget	**43**	**17**	**15**	**15**	**15**
	1.5s	0.9s	0.8s	0.8s	0.8s
ctrace	**9**	**8**	**8**	**8**	**8**
	1.1s	0.9s	0.9s	0.9s	0.9s
pfscan	**6**	**6**	**5**	**5**	**5**
	0.7s	0.7s	0.7s	0.7s	0.7s
engine	**11**	**11**	**7**	**7**	**7**
	1.0s	1.0s	1.0s	1.2s	1.2s
smtprc	**73**	**73**	**46**	**46**	**46**
	5.6s	5.8s	5.0s	6.0s	6.0s
knot	**30**	**29**	**20**	**14**	**12**
	1.2s	1.1s	1.0s	0.9s	1.5s
plip	**25**	**11**	**11**	**11**	**11**
	27.5s	23.8s	24.0s	24.9s	24.9s
eql	**22**	**3**	**3**	**3**	**3**
	2.9s	3.1s	3.1s	3.2s	3.2s
3c501	**24**	**24**	**24**	**24**	**24**
	233.4s	238.3s	238.7s	240.1s	240.1s
sundance	**52**	**3**	**3**	**3**	**3**
	53.4s	98.5s	99.6s	98.2s	98.2s
sis900	**57**	**8**	**8**	**8**	**8**
	40.5s	59.6s	60.5s	61.0s	61.0s
slip	**25**	**19**	**19**	**19**	**19**
	7.7s	16.2s	16.4s	16.5s	16.5s
hp100	**28**	**24**	**23**	**23**	**23**
	18.2s	31.1s	31.6s	31.8s	31.8s

Table 3: Summary of per-feature effects

but these require sizeable test suites to avoid excessive false alarms. Flanagan and Freund [18] have proposed a system for inference which is formulated to support parameterized classes and dependent types. Though the problem is NP-complete, their SAT-based approach can analyze 30K lines of Java code in 46 minutes. Von Praun and Gross's dataflow-based system also requires no annotations and performs well, checking 2000-line programs in a few seconds.

Naik, Aiken, and Whaley present a race detection system for Java [37]. Their system scales well to large Java programs and has found a number of races. They use a cloning-based alias analysis, and hence their approach does not suffer the summarization problem mentioned in Section 2.1 for other context-sensitive analyses. Analyzing Java 1.4 avoids some problems we encountered analyzing C code, such as flow sensitive locking, low-level pointer operations, and unsafe type casts. They also omit linearity checking, which we include in $\lambda_\triangleright$ but occasionally disable in LOCKSMITH.

Several completely automatic static analyses have been developed for finding races in C code. Polyspace [29] is a proprietary tool that uses abstract interpretation to find data races (and other problems). The Blast model checker has been used to find data races in programs written in NesC, a variant of C [28]. Race checking is not limited to checking for consistent correlation and can be state dependent, but is limited to checking global variables and can be quite expensive. Seidl et al [48] propose a framework for analyzing multi-threaded programs that interact through global variables. Using their framework they develop a race detection system for C and apply it to a small

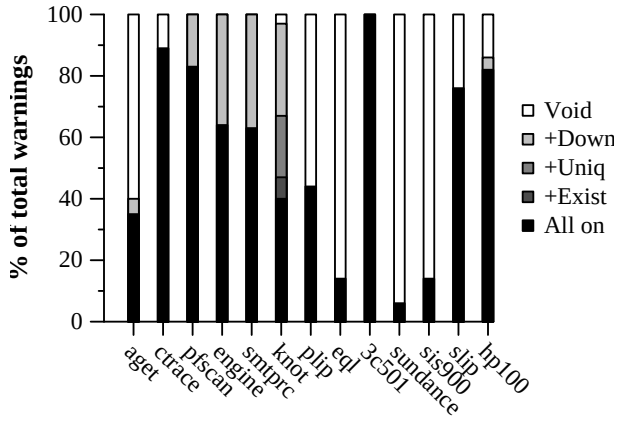


Figure 11: Per-feature precision improvements.

set of benchmarks, finding a number of data races. It is unclear whether their analysis supports context sensitivity and how it models data structures. RacerX [11] does not soundly model some features of C for better scalability and to reduce false alarms, but may miss races as a result. KISS [42] builds on model checking techniques, and has been shown to find many races, but ignores possible thread interleavings, possibly missing the most subtle bugs.

Work that detects violations of *atomicity*, either dynamically [17] or statically [21, 19] typically requires a program to be free of races.

Our analysis is based on ideas initially explored by Reps et al [44] and Rehof and Fähndrich [43], who showed how to encode context-sensitive analysis as a context-free language reachability problem. Our support for existential types is related to `restrict` or `focus` for alias analysis [3, 12]. Our flow-sensitive analysis is a significant extension of our previous work on flow-sensitive type qualifiers [22], which used a similar flow-sensitive constraint graph. Both systems can be seen as inference for a variant of the calculus of capabilities [9].

Correlation between locks and locations is similar to correlation between regions and pointers, and several researchers have looked at the problem of region inference, including the Tofte and Birkedal system for the ML Kit [50]. Henglein et al [27] use a control-flow-sensitive and context-sensitive type system to check that regions with non-lexical allocation and deallocation are used correctly. Our treatment of lock allocation is similar to Henglein et al's treatment of region allocation, but our formal system supports higher-order functions, and we present a constraint-based inference algorithm.

6 Conclusion

We have developed a tool, LOCKSMITH, that aims to prove the absence of data races in a C program. The core component of LOCKSMITH is a context-sensitive *correlation analysis* that determines whether there exists a lock that is held consistently each time a memory location is accessed. This paper formalizes correlation analysis as a constraint-based type and effect system for a simple language $\lambda_{\triangleright}$ which we have proven sound. A novel feature of our formalism is its use of effects to ensure that dynamically-allocated locks can be accurately tracked, with a means to safely hide

effects to better support recursive functions. LOCKSMITH uses a series of techniques to scale correlation analysis to the full C language, including flow-sensitive state tracking, existential types, sharing analysis, and heuristics to model type casts to and from `void*`. When applied to a set of benchmarks, LOCKSMITH discovered a number of real data races with a modest rate of false alarms. We are continuing to explore how to scale LOCKSMITH to large code bases.

Acknowledgments

This research was supposed in part by NSF CCF-0346989, CCF-0430118, and CCF-0524036. We thank Dan Grossman, Greg Morrisett, Boniface Hicks, Nik Swamy, and the anonymous reviewers for their helpful comments. We also thank Will Dogan, Iulian Neamtiu, and Pavlos Papageorgiou for help with the Linux drivers.

References

[1] Rahul Agarwal, Amit Sasturkar, Liqiang Wang, and Scott D. Stoller. Optimized run-time race detection and atomicity checking using partial discovered types. In *ASE*, 2005.

[2] Rahul Agarwal and Scott D. Stoller. Type Inference for Parameterized Race-Free Java. In *VMCAI*, 2004.

[3] Alex Aiken, Jeffrey S. Foster, John Kodumal, and Tachio Terauchi. Checking and Inferring Local Non-Aliasing. In *PLDI*, 2003.

[4] Andrei Alexandrescu, Hans Boehm, Kevlin Henney, Ben Hutchings, Doug Lea, and Bill Pugh. Memory model for multithreaded C++: Issues, 2005. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1777.pdf`.

[5] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *OOPSLA*, 2002.

[6] Chandrasekhar Boyapati and Martin Rinard. A Parameterized Type System for Race-Free Java Programs. In *OOPSLA*, 2001.

[7] Cristiano Calcagno. Stratified Operational Semantics for Safety and Correctness of The Region Calculus. In *POPL*, 2001.

[8] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. In *PLDI*, 2002.

[9] Karl Crary, David Walker, and Greg Morrisett. Typed Memory Management in a Calculus of Capabilities. In *POPL*, 1999.

[10] Manuvir Das, Ben Liblit, Manuel Fähndrich, and Jakob Rehof. Estimating the Impact of Scalable Pointer Analysis on Optimization. In *SAS*, 2001.

[11] Dawson Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *SOSP*, 2003.

[12] Manuel Fähndrich and Robert DeLine. Adoption and Focus: Practical Linear Types for Imperative Programming. In *PLDI*, 2002.

[13] Manuel Fähndrich, Jakob Rehof, and Manuvir Das. From Polymorphic Subtyping to CFL Reachability: Context-Sensitive Flow Analysis Using Instantiation Constraints. Technical Report MSR-TR-99-84, Microsoft Research, 1999.

[14] Cormac Flanagan and Martín Abadi. Types for Safe Locking. In *ESOP*, 1999.

[15] Cormac Flanagan and Stephen N. Freund. Type-Based Race Detection for Java. In *PLDI*, 2000.

[16] Cormac Flanagan and Stephen N. Freund. Detecting race conditions in large programs. In *PASTE*, 2001.

[17] Cormac Flanagan and Stephen N. Freund. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. In *POPL*, 2004.

[18] Cormac Flanagan and Stephen N. Freund. Type Inference Against Races. In *SAS*, 2004.

[19] Cormac Flanagan, Stephen N. Freund, and Marina Lifshin. Type Inference for Atomicity. In *TLDI*, 2005.

[20] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *PLDI*, 2002.

[21] Cormac Flanagan and Shaz Qadeer. A Type and Effect System for Atomicity. In *PLDI*, 2003.

[22] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-Sensitive Type Qualifiers. In *PLDI*, 2002.

[23] David K. Gifford, Pierre Jouvelot, John M. Lucassen, and Mark A. Sheldon. FX-87 Reference Manual. Technical Report MIT/LCS/TR-407, MIT Laboratory for Computer Science, 1987.

[24] Dan Grossman. Type-Safe Multithreading in Cyclone. In *TLDI*, 2003.

[25] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *PLDI*, 2002.

[26] Fritz Henglein. Type Inference with Polymorphic Recursion. *TOPLAS*, 15(2), 1993.

[27] Fritz Henglein, Henning Makholm, and Henning Niss. A Direct Approach to Control-Flow Sensitive Region-Based Memory Management. In *PPDP*, 2001.

[28] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. In *PLDI*, 2004.

[29] Chris Hote. Run-Time Error Detection Through Semantic Analysis, 2004. `http://www.polyspace.com/pdf/Semantics_Analysis.pdf`.

[30] David Hovemeyer and William Pugh. Finding bugs is easy. In *OOPSLA Companion*, 2004.

[31] Rob Johnson and David Wagner. Finding User/Kernel Bugs With Type Inference. In *Proceedings of the 13th Usenix Security Symposium*, 2004.

[32] John Kodumal and Alex Aiken. Banshee: A scalable constraint-based analysis toolkit. In *SAS*. London, United Kingdom, 2005.

[33] Nancy Leveson and Clark S. Turner. An investigation of the therac-25 accidents, July 1993.

[34] John M. Lucassen and David K. Gifford. Polymorphic Effect Systems. In *POPL*, 1988.

[35] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *POPL*, 1996.

[36] Christian Mossin. *Flow Analysis of Typed Higher-Order Programs.* PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, 1996.

[37] Mayur Naik, Alex Aiken, and John Whaley. Effective Static Race Detection for Java. In *PLDI*, 2006. To appear.

[38] George C. Necula, Scott McPeak, S. P. Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *ICCC*, 2002.

[39] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *PPoPP*, 2003.

[40] Kevin Poulsen. Tracking the blackout bug. `http://www.securityfocus.com/news/8412`, 2004.

[41] Polyvios Pratikakis, Michael Hicks, and Jeffrey S. Foster. Existential Label Flow Inference via CFL Reachability. Technical Report CS-TR-4700, Department of Computer Science, UMD, 2005. Forthcoming.

[42] Shaz Qadeer and Dinghao Wu. KISS: keep it simple and sequential. In *PLDI*, 2004.

[43] Jakob Rehof and Manuel Fähndrich. Type-Based Flow Analysis: From Polymorphic Subtyping to CFL-Reachability. In *POPL*, 2001.

[44] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL*, 1995.

[45] John C. Reynolds. Towards a Grainless Semantics for Shared Variable Concurrency. In *POPL*, 2004.

[46] James Rose, Nikhil Swamy, and Michael Hicks. Dynamic inference of polymorphic lock types. *Science of Computer Programming*, 2005.

[47] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. In *SOSP*, 1997.

[48] Helmut Seidl, Varmo Vene, and Markus Müller-Olm. Global Invariants for Analyzing Multithreaded Applications. In *Proc. of Estonian Academy of Sciences: Phys., Math.*, volume 52, pages 413–436, 2003.

[49] Frederick Smith, David Walker, and Greg Morrisett. Alias Types. In *ESOP*, 2000.

[50] Mads Tofte and Lars Birkedal. A Region Inference Algorithm. *TOPLAS*, 20(4), 1998.

[51] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: Scalable threads for internet services. In *SOSP*, 2003.

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

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

[54] Hongwei Xi and Frank Pfenning. Dependent Types in Practical Programming. In *POPL*, 1999.

A $\lambda_{\triangleright}^{cp}$: Correlation with Polymorphically-Constrained Types

In this appendix we prove the soundness of $\lambda_{\triangleright}$ in two steps. First, we present a type checking system for $\lambda_{\triangleright}$ based on polymorphically-constrained types [36]; we refer to the new type system as $\lambda_{\triangleright}^{cp}$. We prove that $\lambda_{\triangleright}^{cp}$ is sound using the standard syntactic technique based on subject reduction (a.k.a. preservation) [53]. That is, programs that are type-correct under $\lambda_{\triangleright}^{cp}$ exhibit consistent correlation. The key technical challenge in $\lambda_{\triangleright}^{cp}$ is typing the `newlock` operation in a way that supports polymorphism and allows locks to be hidden with [Down], which we discuss below.

Second, we prove that $\lambda_{\triangleright}$ is sound by showing that any correct typing derivation in $\lambda_{\triangleright}$ reduces to a correct $\lambda_{\triangleright}^{cp}$ typing derivation. This second step closely follows Rehof et al [43]. We do not show completeness (that every correct $\lambda_{\triangleright}^{cp}$ derivation has a correct $\lambda_{\triangleright}$ analogue); indeed, we believe completeness fails due to restrictions on recursive functions. We have not seen this as a limitation in practice with LOCKSMITH.

The remainder of this section introduces $\lambda_{\triangleright}^{cp}$ and proves it sound. The reduction is presented in Appendix B.

A.1 Operational Semantics

We begin by formalizing the operational semantics for our source language (which can be found in Figure 2). The operational semantics is defined using a single-step reduction relation between expressions, as shown in Figure 12. We use evaluation contexts $\mathbb{E}$ along with the (Context) rule to encode the (call-by-value) evaluation strategy, as is standard. The rules (β), (δ-if), (δ-pair), (δ-let), (δ-fix) are also standard. Rule (δ-newlock) allocates a new lock $[L]$ that is *fresh*, meaning that it is allocated once per evaluation derivation.

The semantics for references is non-standard. Typically, references are modeled using a heap H, which is a map from run-time locations R to values v, and allocation `ref` v creates a fresh location $R \notin dom(H)$, updating H to map R to v. As the point of $\lambda_{\triangleright}$ is merely to prove consistent correlation, we omit modeling the heap. (δ-ref) creates a fresh location label R and annotates the argument v with that location. When such annotated values are "dereferenced" according to (δ-deref), the label R is merely stripped off. The "assignment" operation (δ-assign) behaves the same as a dereference of the left-hand side, returning the right-hand side. Thus references are simply

$$
\begin{array}{lrcll}
(\beta) & (\lambda x.e)\ v & \longrightarrow & e[x \mapsto v] & \\
(\delta\text{-if}) & \texttt{if0}\ 0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 & \longrightarrow & e_1 & \\
 & \texttt{if0}\ n\ \texttt{then}\ e_1\ \texttt{else}\ e_2 & \longrightarrow & e_2 & n \neq 0 \\
(\delta\text{-pair}) & (v_1, v_2)\,.j & \longrightarrow & v_j & j \in \{1,2\} \\
(\delta\text{-let}) & \texttt{let}\ f = v\ \texttt{in}\ e & \longrightarrow & e[f \mapsto v] & \\
(\delta\text{-fix}) & \texttt{fix}\, f.v & \longrightarrow & v[f \mapsto \texttt{fix}\, f.v] & \\
(\delta\text{-ref}) & \texttt{ref}\ v & \longrightarrow & v^R & R \text{ fresh} \\
(\delta\text{-newlock}) & \texttt{newlock} & \longrightarrow & [L] & L \text{ fresh} \\
(\delta\text{-deref}) & !^{[L]}\, v^R & \longrightarrow & v & \\
(\delta\text{-assign}) & v'^R :=^{[L]} v & \longrightarrow & v &
\end{array}
$$

$$
(\text{Context}) \qquad \frac{e_1 \longrightarrow e_2}{\mathbb{E}[e_1] \longrightarrow \mathbb{E}[e_2]}
$$

$$
\begin{array}{rcl}
\mathbb{E} & ::= & [] \mid \mathbb{E}\ e \mid e\ \mathbb{E} \mid \texttt{if0}\ \mathbb{E}\ \texttt{then}\ e\ \texttt{else}\ e' \mid (\mathbb{E}, e) \mid (e, \mathbb{E}) \mid \mathbb{E}.j \\
 & \mid & \mid \texttt{ref}\ \mathbb{E} \mid !^e\, \mathbb{E} \mid !^{\mathbb{E}}\, e \mid \mathbb{E} :=^{e_2} e_1 \mid e_1 :=^{e_2} \mathbb{E} \mid e_1 :=^{\mathbb{E}} e_2 \\
 & & \\
L & ::= & \langle \text{constant lock labels} \rangle \\
R & ::= & \langle \text{constant location labels} \rangle \\
v & ::= & \ldots \mid [L] \mid v^R
\end{array}
$$

Figure 12: Operational Semantics and Target Language Syntax Extensions

functional "boxes" that are dynamically allocated, and there is no aliasing in this semantics. For the purposes of correlation, we wish to prove that for any value v^R, if program evaluation yields redexes $!^{[L]}\, v^R$ and $!^{[L']}\, v^R$ then $L = L'$, and similarly for redexes $v^R :=^{[L]} v'$, i.e., the "boxes" represented by references are always accessed with the correct lock. We believe that it is straightforward to add explicit heap modeling to this system.

A.2 Typing

Typing judgments in $\lambda^{cp}_{\triangleright}$ have the form

$$C; \Gamma \vdash_{cp} e : \tau; \varepsilon$$

Here, C is a set of *constraints*; Γ is an *environment* mapping variables x to polytypes $\forall \vec{\alpha}[C].\tau$ (we write τ to denote polytype $\forall[\emptyset].\tau$); and ε is an *effect* that tracks lock allocations. This judgment is read, "Given constraints C, in environment Γ expression e has type τ and when evaluated will allocate locks ε."

The type and constraint language for $\lambda^{cp}_{\triangleright}$ is shown in Figure 13. Function types are annotated with an effect, listing the locks allocated when the function is called. Lock labels ϑ include lock variables ℓ and lock constants L, while location labels φ include location variables ρ and constants R. Variables ℓ and ρ can be quantified in polytypes (and are collectively referred to using metavariables α, β). In our type rules, we use *substitutions* ϕ that map label variables to labels.

Definition 6 (Substitution) *We define a substitution ϕ as a function from label variables $\vec{\alpha}$ to*

types	τ	::=	$int \mid \tau \times \tau \mid \tau \to^{\varepsilon} \tau' \mid lock\ \ell \mid ref^{\rho}\ \tau$	
lock labels	ϑ	::=	$\ell \mid L$	
location labels	φ	::=	$\rho \mid R$	
label variables	α, β	::=	$\ell \mid \rho$	
labels	l	::=	$\vartheta \mid \varphi$	
effects	ε	::=	$\emptyset \mid \{\ell\} \mid \varepsilon \uplus \varepsilon' \mid \varepsilon \cup \varepsilon'$	
polytypes	σ	::=	$\forall\vec{\alpha}[C].\tau$	
constraint sets	C	::=	$\emptyset \mid \{c\} \mid C \cup C$	
constraints	c	::=	$\varphi \leq \rho$	(location flow)
		$\mid$	$\rho \rhd \ell$	(correlation)
		$\mid$	$L \leq^1 \ell$	(lock allocation)
		$\mid$	$\nu\vec{\alpha}[C;\varepsilon]$	(encapsulated constraints)

Figure 13: $\lambda_{\rhd}^{cp}$ Types and Constructors

labels l. We write $dom(\phi)$ to denote those labels for which ϕ is not the identity, and similarly write $rng(\phi)$ as the image of ϕ applied to $dom(\phi)$.

Intuitively, the type $\forall\vec{\alpha}[C].\tau$ stands for any type $\phi(\tau)$ where $\phi(C)$ is satisfied, for any substitution ϕ with $dom(\phi) = \vec{\alpha}$. Reference types and lock types are annotated with label variables describing the run-time location or lock annotations of their respective values.

There are four kinds of constraints c that make up constraint sets C. The first two kinds of constraints also appear in $\lambda_{\rhd}$. Constraints $\varphi \leq \rho$ describe flow from φ to ρ; these are introduced by subtyping and reference allocation. Constraints $\varphi \rhd \ell$ indicate *correlation*: φ is correlated with ℓ, as indicated by a dereference or assignment. The last two kinds of constraints are new to $\lambda_{\rhd}^{cp}$. Constraints $L \leq^1 \ell$ indicate that a `newlock` expression of type $lock\ \ell$ has been evaluated, generating a fresh lock constant $[L]$. As such, these constraints are not necessary for type checking source programs, but are rather needed for the preservation proof. Constraints $\nu\vec{\alpha}[C;\varepsilon]$ describe *encapsulated constraints*. These are used to handle recursion, and otherwise avoid clashes of lock names. We describe these in greater detail below. Notice that in $\lambda_{\rhd}^{cp}$, there are no instantiation constraints, as $\lambda_{\rhd}^{cp}$ includes explicit constraint copying.

Definition 7 (Bound and Free Labels) *We write $fl(\cdot)$ to denote those labels that are not bound in some structure $\cdot$, where $\cdot$ could be C, Γ, τ, ε, or σ. Figure 14 gives a formal definition. We write $strip(c)$ to "strip off" the binders of encapsulated constraints; i.e., $strip(\nu\vec{\alpha}[C;\varepsilon]) = C$, but $strip(c) = c$ for other kinds of constraints c. The transitive closure of this operation is written $strip^*$. Using this, we define the bound labels of a constraint set C as $bl(C) = fl(strip^*(C)) \setminus fl(C)$.*

The typing rules are shown in Figures 15 (monomorphic rules) and Figure 16 (polymorphic rules). Most of the monomorphic rules are standard. The [Newlock] and [Ref] rules construct values of types $lock\ \ell$ and $ref^{\rho}\ \tau$, respectively; operationally these values have the form $[L]$ and v^R. For [Newlock] the lock label must be *linear*. Roughly speaking, a lock label ℓ is linear if it never represents two different run-time locks that could reside in the same storage or are simultaneously live. Therefore we require ℓ to be a fresh variable in the derivation, which is achieved by putting ℓ

$$
\begin{array}{rcl}
\mathit{fl}(\mathit{int}) & = & \emptyset \\
\mathit{fl}(\tau_1 \times \tau_2) & = & \mathit{fl}(\tau_1) \cup \mathit{fl}(\tau_2) \\
\mathit{fl}(\tau_1 \to^{\varepsilon} \tau_2) & = & \varepsilon \cup \mathit{fl}(\tau_1) \cup \mathit{fl}(\tau_2) \\
\mathit{fl}(\mathit{lock}\ \ell) & = & \{\ell\} \\
\mathit{fl}(\mathit{ref}^{\rho}\ \tau) & = & \{\rho\} \cup \mathit{fl}(\tau) \\
\mathit{fl}(\Gamma, f : \forall\vec{\alpha}[C].\tau) & = & \mathit{fl}(\Gamma) \cup ((\mathit{fl}(\tau) \cup \mathit{fl}(C)) \setminus \vec{\alpha}) \\
\mathit{fl}(\Gamma, x : \tau) & = & \mathit{fl}(\Gamma) \cup \mathit{fl}(\tau) \\
\mathit{fl}(C \cup \{c\}) & = & \mathit{fl}(C) \cup \mathit{fl}(c) \\
\mathit{fl}(\rho \leq \rho') & = & \{\rho, \rho'\} \\
\mathit{fl}(\rho \rhd \ell) & = & \{\rho, \ell\} \\
\mathit{fl}(L \leq^1 \ell) & = & \{\ell\} \\
\mathit{fl}(\nu\vec{\alpha}[C; \varepsilon]) & = & \mathit{fl}(C) \setminus \vec{\alpha}
\end{array}
$$

Figure 14: Free Labels

in an effect ε that must be disjoint with effects in subderivations. For example, in the [App], [Pair], and [Assign] rules, the overall effects are the *disjoint* union of their constituent parts. The [Cond] rule is similar, except that we use non-disjoint union to combine the effects of the two branches, since only one branch is evaluated at run-time (we could also have required the effects of both branches to be the same, and then added a rule to allow arbitrary expansion of an effect). We do not use effects for locations because they need not be linear.

Also noteworthy are [Deref] and [Assign], each of which have the premise $C \vdash \rho \rhd \ell$ to indicate that constraints C can prove the lock expression is correlated with the reference being accessed. Finally, the [Lock] and [Loc] rules are for typing allocated locks and locations, respectively (and thus do not apply to source programs but only programs during evaluation). In both cases, a lock's type (respectively, a location's type) always refers to a lock variable ℓ (respectively, a location variable ρ); we relate the lock constant to the variable by requiring $C \vdash L \leq^1 \ell$ (respectively, $C \vdash R \leq \rho$).

Turning to the polymorphic rules in Figure 16, we see that universal polymorphism is introduced in [Let] and [Fix]. As is standard, we allow generalization only of label variables that are not free in the type environment Γ. Notice that in both these rules, the constraints C' that we use to type check v_1 (or v) become the bound constraints in the polymorphic type. Whenever a variable with a universally quantified type is used in the program text, its type is *instantiated*. The [Inst] rule can only be applied if the instantiation $\phi(C')$ of the polymorphic type's constraints can be proven by the constraints C at that point.

[Down] is based on the observation that constraints and effects on labels that are no longer in use—neither part of the result computed by an expression, nor accessible through the environment—can be removed from consideration [23, 34, 7]. In region and effect systems, these labels are removed from the effect set, but in our system they are also *encapsulated* into a separate constraint set $\nu\vec{\alpha}[C; \varepsilon]$ which we term *encapsulated constraints*. As shown below, encapsulated constraints do not permit directly proving flow or correlation judgments, but rather permit reasoning about the entire constraint set independent of a particular point in a typing derivation. Roughly speaking, this constraint is read: "there exist fresh labels $\vec{\alpha}$ such that the constraints C hold, where locks labeled ε are allocated by the program." (We use the quantifier ν rather than $\exists$ to emphasize that these labels must be fresh, as in alias types [52]).

With this rule, we introduce the idea of an *alpha-converting substitution*. This is a technical

$$[\text{Id}]\frac{}{C;\Gamma, x:\tau \vdash_{cp} x:\tau;\emptyset} \qquad [\text{Int}]\frac{}{C;\Gamma \vdash_{cp} n : \mathit{int};\emptyset}$$

$$[\text{Lam}]\frac{C;\Gamma, x:\tau \vdash_{cp} e:\tau';\varepsilon}{C;\Gamma \vdash_{cp} \lambda x.e : \tau \to^{\varepsilon} \tau';\emptyset}$$

$$[\text{App}]\frac{C;\Gamma \vdash_{cp} e_1 : \tau \to^{\varepsilon} \tau';\varepsilon_1 \qquad C;\Gamma \vdash_{cp} e_2 : \tau;\varepsilon_2}{C;\Gamma \vdash_{cp} e_1\ e_2 : \tau';\varepsilon \uplus \varepsilon_1 \uplus \varepsilon_2}$$

$$[\text{Pair}]\frac{C;\Gamma \vdash_{cp} e_1 : \tau_1;\varepsilon_1 \qquad C;\Gamma \vdash_{cp} e_2 : \tau_2;\varepsilon_2}{C;\Gamma \vdash_{cp} (e_1, e_2) : \tau_1 \times \tau_2;\varepsilon_1 \uplus \varepsilon_2}$$

$$[\text{Proj}]\frac{C;\Gamma \vdash_{cp} e : \tau_1 \times \tau_2;\varepsilon \qquad j = 1,2}{C;\Gamma \vdash_{cp} e.j : \tau_j;\varepsilon} \qquad [\text{Sub}]\frac{\begin{array}{c}C;\Gamma \vdash_{cp} e:\tau_1;\varepsilon \\ C \vdash \tau_1 \le \tau_2\end{array}}{C;\Gamma \vdash_{cp} e:\tau_2;\varepsilon}$$

$$[\text{Cond}]\frac{\begin{array}{c}C;\Gamma \vdash_{cp} e_1 : \mathit{int};\varepsilon_1 \\ C;\Gamma \vdash_{cp} e_2 : \tau;\varepsilon_2 \qquad C;\Gamma \vdash_{cp} e_3 : \tau;\varepsilon_3\end{array}}{C;\Gamma \vdash_{cp} \texttt{if0}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau;\varepsilon_1 \uplus (\varepsilon_2 \cup \varepsilon_3)}$$

$$[\text{Ref}]\frac{C;\Gamma \vdash_{cp} e:\tau;\varepsilon}{C;\Gamma \vdash_{cp} \texttt{ref}\ e : \mathit{ref}^{\rho}\ \tau;\varepsilon} \qquad [\text{Newlock}]\frac{}{C;\Gamma \vdash_{cp} \texttt{newlock} : \mathit{lock}\ \ell;\{\ell\}}$$

$$[\text{Deref}]\frac{C;\Gamma \vdash_{cp} e_1 : \mathit{ref}^{\rho}\ \tau;\varepsilon_1 \qquad C;\Gamma \vdash_{cp} e_2 : \mathit{lock}\ \ell;\varepsilon_2 \qquad C \vdash \rho \rhd \ell}{C;\Gamma \vdash_{cp} !^{e_2}\ e_1 : \tau;\varepsilon_1 \uplus \varepsilon_2}$$

$$[\text{Assign}]\frac{\begin{array}{c}C;\Gamma \vdash_{cp} e_1 : \mathit{ref}^{\rho}\ \tau;\varepsilon_1 \qquad C;\Gamma \vdash_{cp} e_2 : \tau;\varepsilon_2 \\ C;\Gamma \vdash_{cp} e_3 : \mathit{lock}\ \ell;\varepsilon_3 \qquad C \vdash \rho \rhd \ell\end{array}}{C;\Gamma \vdash_{cp} e_1 :=^{e_3} e_2 : \tau;\varepsilon_1 \uplus \varepsilon_2 \uplus \varepsilon_3}$$

$$[\text{Lock}]\frac{C \vdash L \le^1 \ell}{C;\Gamma \vdash_{cp} [L] : \mathit{lock}\ \ell;\emptyset} \qquad [\text{Loc}]\frac{C;\Gamma \vdash_{cp} v:\tau;\emptyset \qquad C \vdash R \le \rho}{C;\Gamma \vdash_{cp} v^R : \mathit{ref}^{\rho}\ \tau;\emptyset}$$

Figure 15: $\lambda^{cp}_{\rhd}$ Monomorphic Typing Rules

$$\text{[Let]}\ \frac{C';\Gamma \vdash_{cp} v_1 : \tau_1;\emptyset \qquad C;\Gamma, f : \forall\vec{\alpha}[C'].\tau_1 \vdash_{cp} e_2 : \tau_2;\varepsilon_2 \qquad \vec{\alpha} \subseteq (\mathit{ftv}(\tau_1) \cup \mathit{ftv}(C')) \setminus \mathit{ftv}(\Gamma)}{C;\Gamma \vdash_{cp} \texttt{let}\ f = v_1\ \texttt{in}\ e_2 : \tau_2;\varepsilon_2}$$

$$\text{[Fix]}\ \frac{C';\Gamma, f : \forall\vec{\alpha}[C'].\tau \vdash_{cp} v : \tau;\emptyset \qquad \vec{\alpha} \subseteq (\mathit{ftv}(\tau) \cup \mathit{ftv}(C')) \setminus \mathit{ftv}(\Gamma) \qquad C \vdash \phi(C') \qquad \mathit{dom}(\phi) = \vec{\alpha}}{C;\Gamma \vdash_{cp} \texttt{fix}\, f.v : \phi(\tau);\emptyset}$$

$$\text{[Inst]}\ \frac{C \vdash \phi(C') \qquad \mathit{dom}(\phi) = \vec{\alpha}}{C;\Gamma, f : \forall\vec{\alpha}[C'].\tau \vdash_{cp} f^i : \phi(\tau);\emptyset}$$

$$\text{[Down]}\ \frac{\begin{array}{c} C \cup \{\nu\vec{\alpha}[C';\varepsilon']\} \cup \mathit{strip}(\alpha^{\vec{l'}}(\nu\vec{\alpha}[C';\varepsilon'])); \Gamma \vdash_{cp} e : \tau; \varepsilon \uplus \phi_\alpha^{\vec{l'}}(\varepsilon') \\ \phi_\alpha^{\vec{l'}}(\vec{\alpha}) \cap (\mathit{ftv}(\Gamma) \cup \mathit{ftv}(\tau)) = \emptyset \\ \varepsilon' \subseteq \vec{\alpha} \qquad \vec{l'} \supseteq \mathit{ftv}(\mathit{strip}^*(C) \cup \mathit{strip}^*(\nu\vec{\alpha}[C';\varepsilon'])) \cup \varepsilon \end{array}}{C \cup \{\nu\vec{\alpha}[C';\varepsilon']\}; \Gamma \vdash_{cp} e : \tau; \varepsilon}$$

Figure 16: $\lambda_{\triangleright}^{cp}$ Polymorphic Typing Rules and [Down]

device for establishing the freshness of bound variables in encapsulated constraints, and is important for later proving that constraint sets are well-formed even if encapsulated constraints are "instantiated" many times.

Definition 8 (Alpha-converting Substitutions) *We write $\alpha^{\vec{l}}(C)$ denote the alpha-conversion of binders in the encapsulated constraints in C to labels not in $\vec{l}$. Thus we have $\mathit{dom}(\alpha^{\vec{l}}) = \mathit{bl}(C)$ and $\mathit{rng}(\alpha^{\vec{l}}) \cap (\vec{l} \cup \mathit{dom}(\alpha^{\vec{l}}) \cup \mathit{ftv}(C)) = \emptyset$ and $|\mathit{dom}(\alpha^{\vec{l}})| = |\mathit{rng}(\alpha^{\vec{l}})|$. We write $\phi_\alpha^{\vec{l}}$ as the normal, capture-avoiding version of $\alpha^{\vec{l}}$, where $\mathit{strip}^*(\alpha^{\vec{l}}(C)) = \phi_\alpha^{\vec{l}}(\mathit{strip}^*(C))$ while $\phi_\alpha^{\vec{l}}(C) = C$ (since $\mathit{dom}(\phi_\alpha^{\vec{l}})$ only contains binders in C).*

Given this definition, we can now understand rule [Down]. In the first premise, given a constraint set with some encapsulated constraints $\nu\vec{\alpha}[C';\varepsilon']$, we type e by stripping the binders off of the constraints after first alpha-converting them (where $\vec{l'}$ is defined in the last premise to avoid conflicts with existing labels). This alpha-conversion is necessary for ensuring the constraint set is well-formed, as described later. However, we can prune these stripped constraints from the conclusion because the alpha-converted binders ($\phi_\alpha^{\vec{l'}}(\vec{\alpha})$) do not appear in the environment or the final type (second premise). We can similarly remove the effect of any allocations that appear in neither the environment nor the type (as established by the second and third premises), but we note the effect of the allocation in the encapsulated constraints.

Finally, rule [Sub] in Figure 15 uses the subtyping rules in Figure 17. These rules are standard.

A.3 Consistent Correlation

The goal of $\lambda_{\triangleright}^{cp}$ is to prove that well-typed programs are consistently correlated, meaning that a given location R is always accessed with the same lock L. We establish this from the constraints

$$\text{[Sub-Int]}\frac{}{C \vdash \mathit{int} \leq \mathit{int}}$$

$$\text{[Sub-Pair]}\frac{C \vdash \tau_1 \leq \tau_2 \qquad C \vdash \tau_1' \leq \tau_2'}{C \vdash \tau_1 \times \tau_1' \leq \tau_2 \times \tau_2'}$$

$$\text{[Sub-Fun]}\frac{C \vdash \tau_2 \leq \tau_1 \qquad C \vdash \tau_1' \leq \tau_2' \qquad \varepsilon_1 \subseteq \varepsilon_2}{C \vdash \tau_1 \to^{\varepsilon_1} \tau_1' \leq \tau_2 \to^{\varepsilon_2} \tau_2'}$$

$$\text{[Sub-Lock]}\frac{C \vdash \ell_1 \leq \ell_2}{C \vdash \mathit{lock}\ \ell_1 \leq \mathit{lock}\ \ell_2}$$

$$\text{[Sub-Ref]}\frac{C \vdash \rho_1 \leq \rho_2 \qquad C \vdash \tau_1 \leq \tau_2 \qquad C \vdash \tau_2 \leq \tau_1}{C \vdash \mathit{ref}^{\rho_1}\ \tau_1 \leq \mathit{ref}^{\rho_2}\ \tau_2}$$

Figure 17: $\lambda_{\triangleright}^{cp}$ Subtyping

$$\text{[Loc-Flow]}\frac{\varphi \leq \rho \in C}{C \vdash \varphi \leq \rho} \qquad \text{[Lab-Refl]}\frac{}{C \vdash l \leq l}$$

$$\text{[Loc-Trans]}\frac{C \vdash \varphi \leq \rho' \qquad C \vdash \rho' \leq \rho}{C \vdash \varphi \leq \rho} \qquad \text{[Lock-Flow]}\frac{L \leq^1 \ell \in C}{C \vdash L \leq^1 \ell}$$

$$\text{[Encap-Flow]}\frac{\nu\vec{\alpha}[C_0; \varepsilon] \in C \qquad C_0 \vdash C'}{C \vdash \nu\vec{\alpha}[C'; \varepsilon]} \qquad \text{[Correlate]}\frac{C \vdash \varphi \leq \rho \qquad \rho \triangleright \ell \in C}{C \vdash \varphi \triangleright \ell}$$

Figure 18: $\lambda_{\triangleright}^{cp}$ Constraint Logic

C needed to type the program. In particular, we use the constraints C to establish *correlation sets* so that we can prove *consistent correlation*. We repeat Definitions 1 and 2 for clarity:

Definition 9 (Correlation Set) *Given a location ρ and a set of constraints C, we define the* correlation set *of ρ in C as*

$$S(C, \rho) = \{\ell \mid C \vdash \rho \rhd \ell\}$$

Here we write $C \vdash \rho \rhd \ell$ to say that $\rho \rhd \ell$ can be proven from the constraints in C.

Definition 10 (Consistent Correlation) *A set of constraints C is* consistently correlated *iff* $\forall \varphi.\ |S(C, \varphi)| \leq 1$.

Thus, a constraint set C is consistently correlated if all locations φ are either correlated with one lock, or are never accessed and so are correlated with no locks. We refine $S(C, \varphi)$ to refer to only concrete lock labels in its range:

$$S_g(C, \varphi) = \{L \mid C \vdash \varphi \rhd \ell \wedge C \vdash L \leq^1 \ell\}$$

We prove the facts $C \vdash c$ in these definitions (and in typing and subtyping rules presented earlier) according to the rules in Figure 18. The [Loc-Flow], [Lab-Refl], and [Loc-Trans] rules establish flow between locations as the reflexive, transitive closure of atomic flow constraints in C. The only flow permitted between locks is due to [Lab-Refl], which effectively means that each lock name in the program identifies a distinct lock, enforcing linearity. The [Correlate] rule defines correlation as transitive with respect to flow. Finally, observe that encapsulated constraints cannot be used to prove flows or correlations directly, as [Encap-Flow] can only be used to prove weaker encapsulated constraints. Instead, we "unwrap" encapsulated constraints as part of [Down], and we will show below that for well-formed constraint sets, encapsulated constraints can be duplicated arbitrarily many times while preserving consistent correlation.

Figure 19 defines a well-formedness judgment $\varepsilon \vdash_{ok} C \hookrightarrow C'; \vec{\alpha}$ on constraints, whose "inputs" are ε and C. Ignoring the "outputs" we introduce the short form of well-formedness as follows:

Definition 11 *We define $\varepsilon \vdash_{ok} C$ if there exist C', $\vec{\alpha}$ such that $\varepsilon \vdash_{ok} C \hookrightarrow C'; \vec{\alpha}$.*

The well-formedness rules establish several properties. First, bound variables appearing in encapsulated constraints within C are disjoint. Notice that [Con-Encap] includes the bound variables $\vec{\alpha}$ in the output, and that they must be disjoint from binders $\vec{\beta}$ within constraints C', as we have $\vec{\alpha} \uplus \vec{\beta}$. [Con-Encap] also strips the encapsulated constraints before checking them for well-formedness (the second premise), so that the output constraint set contains no encapsulated constraints, but keeps the names of the variables intact. The second line of premises in [Con-Union] then ensures that these variables are disjoint with any binders in "adjacent" constraints. The requirement for disjoint binder variables is the reason for the explicit alpha-conversion when stripping encapsulated constraints in the [Down] rule.

Second, the rules ensure that a given lock variable ℓ is only allocated once. The last premise of [Con-Union] ensures this fact directly, and [Con-Lock] ensures that if $\varepsilon \vdash_{ok} C$ that ε is disjoint from those ℓ for which constraint $\ell \leq^1 \ell$ appears in C, and is likewise disjoint from any ε' appearing in an encapsulated constraint. We also require no lock allocation constraints appear in encapsulated constraints, as enforced by the third premise of [Con-Encap]. This places no limit on expressive

$$\text{[Con-Union]}\ \frac{\begin{array}{c}\varepsilon \vdash_{ok} C_1 \hookrightarrow C_1'; \vec{\alpha} \qquad \varepsilon \vdash_{ok} C_2 \hookrightarrow C_2'; \vec{\beta} \\ \mathit{fl}(C_1') \cap \vec{\beta} = \emptyset \qquad \mathit{fl}(C_2') \cap \vec{\alpha} = \emptyset \\ \text{for all } \varphi.\ |S(C_1' \cup C_2', \varphi)| \leq 1 \\ C_1' \vdash L_1 \leq^1 \ell \ \wedge\ C_2' \vdash L_2 \leq^1 \ell \ \Rightarrow L_1 = L_2\end{array}}{\varepsilon \vdash_{ok} C_1 \cup C_2 \hookrightarrow C_1' \cup C_2'; \vec{\alpha} \uplus \vec{\beta}}$$

$$\text{[Con-Encap]}\ \frac{\begin{array}{c}\varepsilon' \subseteq \vec{\alpha} \qquad \varepsilon \uplus \varepsilon' \vdash_{ok} C \hookrightarrow C'; \vec{\beta} \\ \text{for all } c \in C'.\quad c \neq (L \leq^1 \ell) \\ \text{for all } \ell \in \vec{\alpha}.\quad (C \vdash \varphi \rhd \ell) \Rightarrow \varphi \in \vec{\alpha}\end{array}}{\varepsilon \vdash_{ok} \{\nu\vec{\alpha}[C; \varepsilon']\} \hookrightarrow C'; \vec{\alpha} \uplus \vec{\beta}}$$

$$\text{[Con-Other]}\ \frac{C = \emptyset \vee\ C = \{\rho \rhd \ell\} \vee\ C = \{\varphi \leq \rho\}}{\varepsilon \vdash_{ok} C \hookrightarrow C; \emptyset}$$

$$\text{[Con-Lock]}\ \frac{\ell \notin \varepsilon}{\varepsilon \vdash_{ok} \{L \leq^1 \ell\} \hookrightarrow \{L \leq^1 \ell\}; \emptyset}$$

Figure 19: $\lambda^{cp}_{\rhd}$ Constraint Set Well-Formedness

power as such constraints are not useful for source programs (which should have no occurrences of the [Lock] rule), but it establishes a useful invariant that permits duplicating encapsulated constraints as part of the preservation proof.

Finally, the third premise of [Con-Union] enforces consistent correlation of the stripped constraints, and we can prove as much for the original constraints without much trouble, as we show below. First, we can prove some useful properties.

Lemma 12 (Well-formed Constraint Properties) *If* $\varepsilon \vdash_{ok} C \hookrightarrow C'; \vec{\alpha}$ *then*

1. $C' = \mathit{strip}^*(C)$ *and* $\vec{\alpha} = \mathit{bl}(C)$.
2. $\varepsilon \vdash_{ok} \alpha^{\vec{l'}}(C) \hookrightarrow \phi^{\vec{l'}}_{\alpha}(C'); \phi^{\vec{l'}}_{\alpha}(\vec{\alpha})$ *where* $\vec{l'} \supseteq \varepsilon$.
3. $\ell \in \varepsilon$ *implies* $C \not\vdash L \leq^1 \ell$ *and* $C' \not\vdash L \leq^1 \ell$ *for all* L.
4. $C'' \subseteq C$ *implies* $\varepsilon \vdash_{ok} C''$.
5. $\varepsilon' \subseteq \varepsilon$ *implies* $\varepsilon' \vdash_{ok} C \hookrightarrow C'; \vec{\alpha}$.

Proof: By easy induction on $\varepsilon \vdash_{ok} C \hookrightarrow C'; \vec{\alpha}$. □

We can show well-formed constraints are consistently correlated.

Lemma 13 (Consistent Correlation) *If* $\varepsilon \vdash_{ok} C \hookrightarrow C'; \vec{\alpha}$ *then*

1. *for all* φ, $|S(C, \varphi)| \leq 1$ *and* $|S(C', \varphi)| \leq 1$.
2. $C \vdash L_1 \leq^1 \ell \ \wedge C \vdash L_2 \leq^1 \ell \Rightarrow L_1 = L_2$ *and* $C' \vdash L_1 \leq^1 \ell \ \wedge C' \vdash L_2 \leq^1 \ell \Rightarrow L_1 = L_2$

Proof: Proof by induction on $\varepsilon \vdash_{ok} C \hookrightarrow C'; \vec{\alpha}$. To prove the properties mentioning C (rather than C'), observe by the rules in Figure 18 that encapsulated constraints cannot contribute to correlation sets. That is, let C'' be C with all encapsulated constraints removed; then $C \vdash \rho \rhd \ell$ implies $C'' \vdash \rho \rhd \ell$. It is clear that for $\varepsilon \vdash_{ok} C'' \hookrightarrow C'''; \vec{\beta}$ (by Lemma 12(1)) that $C'' = C'''$ and so the result on C''' implies the result for C'' which implies the result for C. □

Finally, we wish to prove that encapsulated constraints can be freely duplicated while still preserving well-formedness, as mentioned above. To do this, we first establish some useful properties on (well-formed) encapsulated constraints.

Lemma 14 (Encapsulated Constraint Properties) *If* $\varepsilon \vdash_{ok} C \cup \{\nu\vec{\alpha}[C_1; \varepsilon_1]\} \hookrightarrow C' \cup C'_1; \vec{\beta} \uplus \vec{\beta'}$ *where* $\alpha^{\vec{l'}}$ *is an alpha-converting substitution on* $\nu\vec{\alpha}[C_1; \varepsilon_1]$ *with* $\vec{l'} \supseteq \varepsilon \cup fl(C')$ *then*

1. for all $\ell \in bl(C_1) \cup \vec{\alpha}$. $\quad (C'_1 \vdash \varphi \rhd \ell) \Rightarrow \varphi \in bl(C_1) \uplus \vec{\alpha}$
2. *if* $C' \cup C'_1 \cup \phi^{\vec{l'}}_{\alpha}(C'_1) \vdash \varphi \leq \rho$ *then*
   - *if* $\varphi \in fl(C' \cup C'_1)$ *then*
     (1) $\rho \in fl(C' \cup C'_1)$ *implies* $C' \cup C'_1 \vdash \varphi \leq \rho$ *and*
     (2) $\rho \in fl(C' \cup \phi^{\vec{l'}}_{\alpha}(C'_1))$ *implies* $C' \cup C'_1 \vdash \varphi \leq \rho'$ *where* $\phi^{\vec{l'}}_{\alpha}(\rho') = \rho$
   - *if* $\varphi \in fl(C' \cup \phi^{\vec{l'}}_{\alpha}(C'_1))$ *then*
     (3) $\rho \in fl(C' \cup \phi^{\vec{l'}}_{\alpha}(C'_1))$ *implies* $C' \cup \phi^{\vec{l'}}_{\alpha}(C'_1) \vdash \varphi \leq \rho$ *and*
     (4) $\rho \in fl(C' \cup C'_1)$ *implies* $C' \cup \phi^{\vec{l'}}_{\alpha}(C'_1) \vdash \varphi \leq \rho'$ *where* $\phi^{\vec{l'}}_{\alpha}(\rho) = \rho'$

Proof: The first is proved by easy induction on $\varepsilon \vdash_{ok} C \cup \{\nu\vec{\alpha}[C_1; \varepsilon_1]\} \hookrightarrow C' \cup C'_1; \vec{\beta} \uplus \vec{\beta'}$. The last is proved by induction on the derivation $C' \cup C'_1 \cup \phi^{\vec{l'}}_{\alpha}(C'_1) \vdash \varphi \leq \rho$.
Case [Lab-Refl]. We have

$$\text{[Lab-Refl]}\frac{}{C' \cup C'_1 \cup \phi^{\vec{l'}}_{\alpha}(C'_1) \vdash \varphi \leq \varphi}$$

and thus $\rho = \varphi$.

- Assume $\varphi \in fl(C' \cup C'_1)$:
  (1) We have $C' \cup C'_1 \vdash \varphi \leq \varphi$ by [Lab-Refl].
  (2) Assume $\varphi \in fl(C' \cup \phi^{\vec{l'}}_{\alpha}(C'_1))$. We can show that $\varphi \notin dom(\phi^{\vec{l'}}_{\alpha})$ which implies that $\phi^{\vec{l'}}_{\alpha}(\varphi) = \varphi$ and thus $C' \cup C'_1 \vdash \varphi \leq \varphi$ by [Lab-Refl], and the result follows by taking $\rho' = \varphi$. We prove $\varphi \notin dom(\phi^{\vec{l'}}_{\alpha})$ by contradiction. Suppose $\varphi \in dom(\phi^{\vec{l'}}_{\alpha})$, and thus $\varphi \in C'_1$. Since the $dom(\phi^{\vec{l'}}_{\alpha})$ and $rng(\phi^{\vec{l'}}_{\alpha})$ must be disjoint, the fact that $\varphi \in fl(C' \cup \phi^{\vec{l'}}_{\alpha}(C'_1))$ and $\varphi \in fl(C' \cup C'_1)$ implies it must be in the part in which the two constraint sets agree. But that implies that φ appears at least twice in the constraints: bound in $\nu\vec{\alpha}[C_1; \varepsilon_1]$ and elsewhere in the $C \cup \nu\vec{\alpha}[C_1; \varepsilon_1]$, either bound separately or free. But this is impossible since $\varepsilon \vdash_{ok} C \cup \{\nu\vec{\alpha}[C_1; \varepsilon_1]\}$ forbids such duplication.
- Assume $\varphi \in fl(C' \cup \phi^{\vec{l'}}_{\alpha}(C'_1))$; proofs of (3) and (4) mirror (1) and (2), above.

Case [Loc-Flow]. We have

$$[\text{Loc-Flow}]\frac{\varphi \leq \rho \in C' \cup C'_1 \cup \phi^{\vec{l'}}_\alpha(C'_1)}{C' \cup C'_1 \cup \phi^{\vec{l'}}_\alpha(C'_1) \vdash \varphi \leq \rho}$$

From the premise at least one of the following is true: (1) $\varphi \leq \rho \in C'$; (2) $\varphi \leq \rho \in C'_1$; and/or (3) $\varphi \leq \rho \in \phi^{\vec{l'}}_\alpha(C'_1)$. We prove the desired conditions by cases:

1. Assume $\varphi \leq \rho \in C'$, which implies that $\rho, \varphi \notin \mathit{dom}(\phi^{\vec{l'}}_\alpha)$ since by $\varepsilon \vdash_{ok} C \cup \{\nu\vec{\alpha}[C_1; \varepsilon_1]\}$ binders cannot be duplicated. As a result, we easily have $C' \vdash \varphi \leq \rho$ and $C' \vdash \varphi \leq \phi^{\vec{l'}}_\alpha(\rho)$ by [Loc-Flow], and results (1)–(4) easily follow by weakening.

2. Assume $\varphi \leq \rho \in C'_1$.

   (1) $C' \cup C'_1 \vdash \varphi \leq \rho$ by [Loc-Flow]

   (2) If $\rho \in \mathit{fl}(C' \cup \phi^{\vec{l'}}_\alpha(C'_1))$, then we can prove that $\rho \notin \mathit{dom}(\phi^{\vec{l'}}_\alpha)$, so $\phi^{\vec{l'}}_\alpha(\rho) = \rho$ and thus $C'_1 \vdash \varphi \leq \phi^{\vec{l'}}_\alpha(\rho')$ where $\rho' = \rho$ by [Loc-Flow]; the result follows by weakening. To prove $\rho \notin \mathit{dom}(\phi^{\vec{l'}}_\alpha)$, there are two cases. If $\rho \in \mathit{fl}(C')$ then $\varepsilon \vdash_{ok} C \cup \{\nu\vec{\alpha}[C_1; \varepsilon_1]\}$ prevents a binder in C_1 from being duplicated. If $\rho \in \mathit{fl}(\phi^{\vec{l'}}_\alpha(C'_1))$ then we follow the argument from (2) of the [Lab-Refl] case, above.

   (3) Assume $\varphi, \rho \in \mathit{fl}(C' \cup \phi^{\vec{l'}}_\alpha(C'_1))$; we want to show that $\phi^{\vec{l'}}_\alpha(C'_1) \vdash \varphi \leq \rho$ so the result follows by weakening. By the argument for (2), above, we know that $\rho, \varphi \notin \mathit{dom}(\phi^{\vec{l'}}_\alpha)$ and thus $\varphi \leq \rho \in C'_1$ implies $\varphi \leq \rho \in \phi^{\vec{l'}}_\alpha(C'_1)$.

   (4) Assume $\varphi \in \mathit{fl}(C' \cup \phi^{\vec{l'}}_\alpha(C'_1))$ and $\rho \in \mathit{fl}(C' \cup C'_1)$; we want to show that $\phi^{\vec{l'}}_\alpha(C'_1) \vdash \varphi \leq \phi^{\vec{l'}}_\alpha(\rho)$. The fact that $\varphi \leq \rho \in C'_1$ implies $\phi^{\vec{l'}}_\alpha(\varphi) \leq \phi^{\vec{l'}}_\alpha(\rho) \in \phi^{\vec{l'}}_\alpha(C'_1)$. Since $\varphi \in \mathit{fl}(C'_1)$ and $\varphi \in \mathit{fl}(C' \cup \phi^{\vec{l'}}_\alpha(C'_1))$, we know that $\varphi \notin \mathit{dom}(\phi^{\vec{l'}}_\alpha)$ following the argument for (2), above. Therefore, $\varphi \leq \phi^{\vec{l'}}_\alpha(\rho) \in \phi^{\vec{l'}}_\alpha(C'_1)$ which gives us the desired result by [Loc-Flow].

3. Assume $\varphi \leq \rho \in \phi^{\vec{l'}}_\alpha(C'_1)$. The arguments for (1),(2) mirror case 2's arguments for (3),(4), above; likewise (3),(4) mirror (1),(2).

Case [Loc-Trans]. We have

$$[\text{Loc-Trans}]\frac{C' \cup C'_1 \cup \phi^{\vec{l'}}_\alpha(C'_1) \vdash \varphi \leq \rho' \qquad C' \cup C'_1 \cup \phi^{\vec{l'}}_\alpha(C'_1) \vdash \rho' \leq \rho}{C' \cup C'_1 \cup \phi^{\vec{l'}}_\alpha(C'_1) \vdash \varphi \leq \rho}$$

To prove (1)–(4), we consider two cases: (1) when $\rho' \in \mathit{fl}(C' \cup C'_1)$ and (2) when $\rho' \in \mathit{fl}(C' \cup \phi^{\vec{l}}_\alpha(C'_1))$. Consider the former case:

(1) Assume $\varphi \in \mathit{fl}(C' \cup C'_1)$ and $\rho \in \mathit{fl}(C' \cup C'_1)$. We have $C' \cup C'_1 \vdash \varphi \leq \rho'$ and $C' \cup C'_1 \vdash \rho' \leq \rho$ by induction, and the result follows by [Loc-Trans].

(2) Assume $\varphi \in \mathit{fl}(C' \cup C'_1)$ and $\rho \in \mathit{fl}(C' \cup \phi^{\vec{l'}}_\alpha(C'_1))$. By induction we have $C' \cup C'_1 \vdash \varphi \leq \rho'$ and $C' \cup C'_1 \vdash \rho' \leq \rho''$ where $\phi^{\vec{l'}}_\alpha(\rho'') = \rho$, and the result follows by [Loc-Trans].

(3) Assume $\varphi \in \mathit{fl}(C' \cup \phi^{\vec{l'}}_\alpha(C'_1))$ and $\rho \in \mathit{fl}(C' \cup \phi^{\vec{l'}}_\alpha(C'_1))$ so we must prove $C' \cup \phi^{\vec{l'}}_\alpha(C'_1) \vdash \varphi \leq \rho$. By induction we have $C' \cup \phi^{\vec{l'}}_\alpha(C'_1) \vdash \varphi \leq \phi^{\vec{l'}}_\alpha(\rho')$ and $C' \cup C'_1 \vdash \rho' \leq \rho''$ where $\phi^{\vec{l'}}_\alpha(\rho'') = \rho$. To get the desired result by [Loc-Trans], we need to show $C' \cup \phi^{\vec{l'}}_\alpha(C'_1) \vdash \phi^{\vec{l'}}_\alpha(\rho') \leq \phi^{\vec{l'}}_\alpha(\rho'')$. This follows from $C' \cup C'_1 \vdash \rho' \leq \rho''$ which implies $\phi^{\vec{l'}}_\alpha(C') \cup \phi^{\vec{l'}}_\alpha(C'_1) \vdash \phi^{\vec{l'}}_\alpha(\rho') \leq \phi^{\vec{l'}}_\alpha(\rho'')$, and from $\phi^{\vec{l'}}_\alpha(C') = C'$ since the binders in $\nu\vec{\alpha}[C_1; \varepsilon_1]$ must be disjoint with $\mathit{fl}(C)$ by $\varepsilon \vdash_{ok} C \cup \{\nu\vec{\alpha}[C_1; \varepsilon_1]\}$.

(4) $\varphi \in \mathit{fl}(C' \cup \phi^{\vec{l'}}_\alpha(C'_1))$ and $\rho \in \mathit{fl}(C' \cup C'_1)$ so we must prove $C' \cup \phi^{\vec{l'}}_\alpha(C'_1) \vdash \varphi \leq \phi^{\vec{l'}}_\alpha(\rho)$. By induction we have $C' \cup \phi^{\vec{l'}}_\alpha(C'_1) \vdash \varphi \leq \phi^{\vec{l'}}_\alpha(\rho')$ and $C' \cup C'_1 \vdash \rho' \leq \rho$. To get the desired result by [Loc-Trans] we need to show $C' \cup \phi^{\vec{l'}}_\alpha(C'_1) \vdash \phi^{\vec{l'}}_\alpha(\rho') \leq \phi^{\vec{l'}}_\alpha(\rho)$, but this follows by the same reasoning as case (3), above.

When assuming $\rho' \in \mathit{fl}(C' \cup \phi^{\vec{l'}}_\alpha(C'_1))$, the reasoning mirrors the cases above $\square$

Finally, we prove that encapsulated constraints can be duplicated and "stripped" while still preserving well-formedness.

Lemma 15 (Duplicated Encapsulated Constraint)
If

$$\varepsilon \vdash_{ok} C \cup \{\nu\vec{\alpha}[C_1; \varepsilon_1]\} \hookrightarrow C' \cup C'_1; \vec{\alpha} \uplus \vec{\beta}$$

then

$$\varepsilon \uplus \phi^{\vec{l'}}_\alpha(\varepsilon_1) \vdash_{ok} C \cup \{\nu\vec{\alpha}[C_1; \varepsilon_1]\} \cup \mathit{strip}(\alpha^{\vec{l'}}(\nu\vec{\alpha}[C_1; \varepsilon_1])) \hookrightarrow C' \cup C'_1 \cup \phi^{\vec{l'}}_\alpha(C'_1); \vec{\alpha} \uplus \vec{\beta} \uplus \phi^{\vec{l'}}_\alpha(\vec{\beta})$$

where $\vec{l'} \supseteq \mathit{fl}(C'_1) \cup \mathit{fl}(C') \cup \varepsilon$.

Proof: By inversion, we have

$$[\text{Con-Union}]\frac{\begin{array}{c}\varepsilon \vdash_{ok} C \hookrightarrow C'; \vec{\alpha} \quad \varepsilon \vdash_{ok} \nu\vec{\alpha}[C_1; \varepsilon_1] \hookrightarrow C'_1; \vec{\beta} \\ \mathit{fl}(C') \cap \vec{\beta} = \emptyset \qquad \mathit{fl}(C'_1) \cap \vec{\alpha} = \emptyset \\ \text{for all } \varphi.\ |S(C' \cup C'_1, \varphi)| \leq 1 \\ C' \vdash L_1 \leq^1 \ell \ \wedge \ C'_1 \vdash L_2 \leq^1 \ell \ \Rightarrow L_1 = L_2\end{array}}{\varepsilon \vdash_{ok} C \cup \{\nu\vec{\alpha}[C_1; \varepsilon_1]\} \hookrightarrow C' \cup C'_1; \vec{\alpha} \uplus \vec{\beta}}$$

We want to prove

$$[\text{Con-Union}]\frac{\begin{array}{c}\varepsilon \uplus \phi^{\vec{l'}}_\alpha(\varepsilon_1) \vdash_{ok} C \cup \{\nu\vec{\alpha}[C_1; \varepsilon_1]\} \hookrightarrow C' \cup C'_1; \vec{\alpha} \uplus \vec{\beta}^{(1)} \\ \varepsilon \uplus \phi^{\vec{l'}}_\alpha(\varepsilon_1) \vdash_{ok} \mathit{strip}(\alpha^{\vec{l'}}(\nu\vec{\alpha}[C_1; \varepsilon_1])) \hookrightarrow \phi^{\vec{l'}}_\alpha(C'_1); \phi^{\vec{l'}}_\alpha(\vec{\beta})^{(2)} \\ \mathit{fl}(C' \cup C'_1) \cap \phi^{\vec{l'}}_\alpha(\vec{\beta}) = \emptyset^{(3)} \qquad \mathit{fl}(\phi^{\vec{l'}}_\alpha(C'_1)) \cap (\vec{\alpha} \uplus \vec{\beta}) = \emptyset^{(4)} \\ \text{for all } \varphi.\ |S(C' \cup C'_1 \cup \phi^{\vec{l'}}_\alpha(C'_1), \varphi)| \leq 1^{(5)} \\ C' \cup C'_1 \vdash L_1 \leq^1 \ell \ \wedge \ \phi^{\vec{l'}}_\alpha(C'_1) \vdash L_2 \leq^1 \ell \ \Rightarrow L_1 = L_2^{(6)}\end{array}}{\begin{array}{l}\varepsilon \uplus \phi^{\vec{l'}}_\alpha(\varepsilon_1) \vdash_{ok} C \cup \{\nu\vec{\alpha}[C_1; \varepsilon_1]\} \cup \mathit{strip}(\alpha^{\vec{l'}}(\nu\vec{\alpha}[C_1; \varepsilon_1])) \hookrightarrow^{(7)} \\ C' \cup C'_1 \cup \phi^{\vec{l'}}_\alpha(C'_1); \vec{\alpha} \uplus \vec{\beta} \uplus \phi^{\vec{l'}}_\alpha(\vec{\beta})\end{array}}$$

We prove each of the seven labeled statements (6 premises and well-formedness of $\uplus$ in the conclusion) to get the desired result:

(1) $\varepsilon \uplus \phi_\alpha^{\vec{l'}}(\varepsilon_1) \vdash_{ok} C \cup \{\nu\vec{\alpha}[C_1;\varepsilon_1]\} \hookrightarrow C' \cup C_1'; \vec{\alpha} \uplus \vec{\beta}$.

Proof by easy induction on $\varepsilon \vdash_{ok} C \cup \{\nu\vec{\alpha}[C_1;\varepsilon_1]\} \hookrightarrow C' \cup C_1'; \vec{\alpha} \uplus \vec{\beta}$. The key is that adding $\phi_\alpha^{\vec{l'}}(\varepsilon_1)$ to the input effect cannot cause applications of [Con-Lock] to fail because $\varepsilon_1 \subseteq dom(\phi_\alpha^{\vec{l'}})$ and $rng(\phi_\alpha^{\vec{l'}}) \cap fl(strip^*(C \cup \{\nu\vec{\alpha}[C_1;\varepsilon_1]\})) = \emptyset$ by the definition of $\vec{l'}$.

(2) $\varepsilon \uplus \phi_\alpha^{\vec{l'}}(\varepsilon_1) \vdash_{ok} strip(\alpha^{\vec{l'}}(\nu\vec{\alpha}[C_1;\varepsilon_1])) \hookrightarrow \phi_\alpha^{\vec{l'}}(C_1'); \phi_\alpha^{\vec{l'}}(\vec{\beta})$.

We have by assumption that $\varepsilon \vdash_{ok} \nu\vec{\alpha}[C_1;\varepsilon_1] \hookrightarrow C_1'; \vec{\beta}$ and so $\varepsilon \vdash_{ok} \alpha^{\vec{l'}}(\nu\vec{\alpha}[C_1;\varepsilon_1]) \hookrightarrow \phi_\alpha^{\vec{l'}}(C_1'); \phi_\alpha^{\vec{l'}}(\vec{\beta})$ by Lemma 12(2). The final rule of this derivation must be [Con-Encap], so by inversion we have $\varepsilon \uplus \phi_\alpha^{\vec{l'}}(\varepsilon_1) \vdash_{ok} strip(\alpha^{\vec{l'}}(\nu\vec{\alpha}[C_1;\varepsilon_1])) \hookrightarrow \phi_\alpha^{\vec{l'}}(C_1'); \phi_\alpha^{\vec{l'}}(\vec{\beta})$ which is the desired result.

(3) $fl(C' \cup C_1') \cap \phi_\alpha^{\vec{l'}}(\vec{\beta}) = \emptyset$.

Follows since $\vec{\beta} = dom(\phi_\alpha^{\vec{l'}})$ by Lemma 12(1), and $rng(\phi_\alpha^{\vec{l'}}) \cap fl(strip^*(C \cup \{\nu\vec{\alpha}[C_1;\varepsilon_1]\})) = \emptyset$ by the definition of $\vec{l'}$, and $strip^*(C \cup \{\nu\vec{\alpha}[C_1;\varepsilon_1]\}) = C \cup C_1'$ by Lemma 12(1).

(4) $fl(\phi_\alpha^{\vec{l'}}(C_1')) \cap (\vec{\alpha} \uplus \vec{\beta}) = \emptyset$.

We know $bl(C) = \vec{\alpha}$ (thus $\vec{\alpha} \subseteq fl(C')$) and $bl(\nu\vec{\alpha}[C_1;\varepsilon_1]) = dom(\phi_\alpha^{\vec{l'}}) = \vec{\beta}$ (thus $\vec{\beta} \subseteq fl(C_1')$) by Lemma 12(1). Consider some $\alpha \in fl(C_1')$. If $\alpha \in dom(\phi_\alpha^{\vec{l'}})$ then $\phi_\alpha^{\vec{l'}}(\alpha) \notin (\vec{\alpha} \uplus \vec{\beta})$ because by the fact that $\phi_\alpha^{\vec{l'}}$ derives from an alpha-converting substitution we have that $rng(\phi_\alpha^{\vec{l'}}) \cap (\vec{l'} \cup dom(\phi_\alpha^{\vec{l'}})) = \emptyset$ where $\vec{l'} \supseteq (fl(C') \cup fl(C_1')) \supseteq (\vec{\alpha} \uplus \vec{\beta})$. If $\alpha \notin dom(\phi_\alpha^{\vec{l'}})$ then $\phi_\alpha^{\vec{l'}}(\alpha) = \alpha$ and $\alpha \notin \vec{\beta}$. Moreover, $\alpha \notin \vec{\alpha}$ by our assumption $\varepsilon \vdash_{ok} C \cup \{\nu\vec{\alpha}[C_1;\varepsilon_1]\} \hookrightarrow C' \cup C_1'; \vec{\alpha} \uplus \vec{\beta}$ whose last rule must be [Con-Union] by inversion, which contains the premise $fl(C_1') \cap \vec{\alpha} = \emptyset$.

(5) for all φ'. $|S(C' \cup C_1' \cup \phi_\alpha^{\vec{l'}}(C_1'), \varphi')| \leq 1$.

The proof proceeds by contradiction: assume that there exists some φ, ℓ_1 and ℓ_2 where $C' \cup C_1' \cup \phi_\alpha^{\vec{l'}}(C_1') \vdash \varphi \rhd \ell_1$ and $C' \cup C_1' \cup \phi_\alpha^{\vec{l'}}(C_1') \vdash \varphi \rhd \ell_2$. Thus we must have

$$\text{[Correlate]}\frac{C' \cup C_1' \cup \phi_\alpha^{\vec{l'}}(C_1') \vdash \varphi \leq \rho \qquad \rho \rhd \ell_1 \in C' \cup C_1' \cup \phi_\alpha^{\vec{l'}}(C_1')}{C' \cup C_1' \cup \phi_\alpha^{\vec{l'}}(C_1') \vdash \varphi \rhd \ell_1}$$

and

$$\text{[Correlate]}\frac{C' \cup C_1' \cup \phi_\alpha^{\vec{l'}}(C_1') \vdash \varphi \leq \rho' \qquad \rho' \rhd \ell_2 \in C' \cup C_1' \cup \phi_\alpha^{\vec{l'}}(C_1')}{C' \cup C_1' \cup \phi_\alpha^{\vec{l'}}(C_1') \vdash \varphi \rhd \ell_2}$$

We prove one of these derivations is impossible by showing that it would contradict that $|S(C' \cup C_1', \varphi)| \leq 1$ or $|S(C' \cup \phi_\alpha^{\vec{l'}}(C_1'), \varphi)| \leq 1$. We know the former is true by the fact that $\varepsilon \vdash_{ok} C \cup \{\nu\vec{\alpha}[C_1;\varepsilon_1]\} \hookrightarrow C' \cup C_1'; \vec{\alpha} \uplus \vec{\beta}$ (assumption) and Lemma 13. The latter is true by the fact that $\varepsilon \vdash_{ok} C \cup \{\alpha^{\vec{l'}}(\nu\vec{\alpha}[C_1;\varepsilon_1])\} \hookrightarrow C' \cup \phi_\alpha^{\vec{l'}}(C_1'); \vec{\alpha} \uplus \phi_\alpha^{\vec{l'}}(\vec{\beta})$ (by Lemma 12(2)) and Lemma 13.

The proof proceeds by cases. Consider how we might prove the premises of $C' \cup C_1' \cup \phi_\alpha^{\vec{l'}}(C_1') \vdash \varphi \rhd \ell_1$:

1. $C' \cup C'_1 \vdash \varphi \leq \rho$ and $\rho \rhd \ell_1 \in C' \cup C'_1$. From this, we know that $\varphi \in \mathit{fl}(C' \cup C'_1)$, since either $\varphi = \rho$ and $\rho \in \mathit{fl}(C' \cup C'_1)$ (since $\rho \rhd \ell_1 \in C' \cup C'_1$), or else $\varphi \neq \rho$ and so $C' \cup C'_1 \vdash \varphi \leq \rho$ implies that $\varphi \in \mathit{fl}(C' \cup C'_1)$ by inspection of the rules in Figure 18. Now consider the premises of $C' \cup C'_1 \cup \phi^{\vec{l'}}_{\alpha}(C'_1) \vdash \varphi \rhd \ell_2$:

   (a) $C' \cup C'_1 \cup \phi^{\vec{l'}}_{\alpha}(C'_1) \vdash \varphi \leq \rho'$ and $\rho' \rhd \ell_2 \in C' \cup C'_1$. Since $\varphi \in \mathit{fl}(C' \cup C'_1)$ and $\rho' \in \mathit{fl}(C' \cup C'_1)$ (since $\rho' \rhd \ell_2 \in C' \cup C'_1$), by Lemma 14(2)(1) we must have that $C' \cup C'_1 \vdash \varphi \leq \rho'$. But then this implies that $C' \cup C'_1 \vdash \varphi \rhd \ell_2$ which contradicts that $|S(C' \cup C'_1, \varphi)| \leq 1$.

   (b) $C' \cup C'_1 \cup \phi^{\vec{l'}}_{\alpha}(C'_1) \vdash \varphi \leq \rho'$ and $\rho' \rhd \ell_2 \in C' \cup \phi^{\vec{l'}}_{\alpha}(C'_1)$. Since $\varphi \in \mathit{fl}(C' \cup C'_1)$ and $\rho' \in C' \cup \phi^{\vec{l'}}_{\alpha}(C'_1)$, by Lemma 14(2)(2) we must have that $C' \cup C'_1 \vdash \varphi \leq \rho''$ where $\phi^{\vec{l'}}_{\alpha}(\rho'') = \rho'$. Also $\rho'' \rhd \ell'_2 \in C' \cup C'_1$ where $\phi^{\vec{l'}}_{\alpha}(\ell'_2) = \ell_2$ since $\rho' \rhd \ell_2 \in C' \cup \phi^{\vec{l'}}_{\alpha}(C'_1)$ by assumption. By Lemma 14(1) $\phi^{\vec{l'}}_{\alpha}(\ell'_2) = \ell_2$ and thus $\rho'' \rhd \ell_2 \in C' \cup C'_1$. But then this implies that $C' \cup C'_1 \vdash \varphi \rhd \ell_2$ which contradicts that $|S(C' \cup C'_1, \varphi)| \leq 1$.

2. $C' \cup C'_1 \vdash \varphi \leq \rho$ and $\rho \rhd \ell_1 \in C' \cup \phi^{\vec{l'}}_{\alpha}(C'_1)$. If $\varphi \neq \rho$ then $\rho \in \mathit{fl}(C' \cup C'_1)$ and since $\rho \in \mathit{fl}(C' \cup \phi^{\vec{l'}}_{\alpha}(C'_1))$, it must be that $\rho \notin \mathit{bl}(C_1) \cup \vec{\alpha}$. But then by Lemma 14(1) we must also have that $\rho \rhd \ell_1 \in C' \cup C'_1$, so the above case applies. If $\varphi = \rho$ then $C' \cup \phi^{\vec{l'}}_{\alpha}(C'_1) \vdash \varphi \leq \rho$ as well, so the case below applies.

3. $C' \cup \phi^{\vec{l'}}_{\alpha}(C'_1) \vdash \varphi \leq \rho$ and $\rho \rhd \ell_1 \in C' \cup \phi^{\vec{l'}}_{\alpha}(C'_1)$. Mirroring the argument of case 1, above, we know $\varphi \in \mathit{fl}(C_1 \cup \phi^{\vec{l'}}_{\alpha}(C'_1))$. Now consider the premises of $C' \cup C'_1 \cup \phi^{\vec{l'}}_{\alpha}(C'_1) \vdash \varphi \rhd \ell_2$:

   (a) $C' \cup C'_1 \cup \phi^{\vec{l'}}_{\alpha}(C'_1) \vdash \varphi \leq \rho'$ and $\rho' \rhd \ell_2 \in C' \cup \phi^{\vec{l'}}_{\alpha}(C'_1)$. Mirroring the argument from case 1(a) above, we can show $C' \cup \phi^{\vec{l'}}_{\alpha}(C'_1) \vdash \varphi \leq \rho'$. But then this implies that $C' \cup \phi^{\vec{l'}}_{\alpha}(C'_1) \vdash \varphi \rhd \ell_2$ which contradicts that $|S(C' \cup \phi^{\vec{l'}}_{\alpha}(C'_1), \varphi)| \leq 1$ since we already have that $C' \cup \phi^{\vec{l'}}_{\alpha}(C'_1) \vdash \varphi \rhd \ell_1$.

   (b) $C' \cup C'_1 \cup \phi^{\vec{l'}}_{\alpha}(C'_1) \vdash \varphi \leq \rho'$ and $\rho' \rhd \ell_2 \in C' \cup C'_1$. Since $\varphi \in \mathit{fl}(C' \cup \phi^{\vec{l'}}_{\alpha}(C'_1))$ and $\rho' \in C' \cup C'_1$, by Lemma 14(2)(4) we must have that $C' \cup \phi^{\vec{l'}}_{\alpha}(C'_1) \vdash \varphi \leq \phi^{\vec{l'}}_{\alpha}(\rho')$. Since $\rho' \rhd \ell_2 \in C' \cup C'_1$ we have $\phi^{\vec{l'}}_{\alpha}(\rho') \rhd \phi^{\vec{l'}}_{\alpha}(\ell_2) \in C' \cup \phi^{\vec{l'}}_{\alpha}(C'_1)$, and thus $C' \cup \phi^{\vec{l'}}_{\alpha}(C'_1) \vdash \varphi \rhd \phi^{\vec{l'}}_{\alpha}(\ell_2)$. We can show that $\phi^{\vec{l'}}_{\alpha}(\ell_2) \neq \ell_1$, but then this contradicts $|S(C' \cup \phi^{\vec{l'}}_{\alpha}(C'_1), \varphi)| \leq 1$.

   Given that $C' \cup \phi^{\vec{l'}}_{\alpha}(C'_1) \vdash \varphi \leq \phi^{\vec{l'}}_{\alpha}(\rho')$ and $\phi^{\vec{l'}}_{\alpha}(\rho') \rhd \phi^{\vec{l'}}_{\alpha}(\ell_2) \in C' \cup \phi^{\vec{l'}}_{\alpha}(C'_1)$, if $C' \cup C'_1 \cup \phi^{\vec{l'}}_{\alpha}(C'_1) \vdash \varphi \rhd \ell_1$ where $\ell_2 \neq \ell_1$, we want to show that $\phi^{\vec{l'}}_{\alpha}(\ell_2) \neq \ell_1$. Two cases. First, if $\ell_2 \notin \mathit{dom}(\phi^{\vec{l'}}_{\alpha})$ then $\phi^{\vec{l'}}_{\alpha}(\ell_2) = \ell_2$ and the result follows since $\ell_2 \neq \ell_1$ by assumption. Otherwise, $\ell_2 \in \mathit{bl}(C_1) \cup \vec{\alpha}$ and so by Lemma 14(1) we have $\varphi \in \mathit{bl}(C_1) \cup \vec{\alpha}$ and thus $\varphi \in \mathit{dom}(\phi^{\vec{l'}}_{\alpha})$. Now consider two sub-cases. First, if $\ell_1 \in \mathit{fl}(C' \cup C'_1)$ then $\phi^{\vec{l'}}_{\alpha}(\ell_2) \neq \ell_1$ since $\vec{l'} \supseteq \mathit{fl}(C' \cup C'_1)$. Otherwise, if $\ell_1 \in \mathit{rng}(\phi^{\vec{l'}}_{\alpha})$ then $\varphi \in \mathit{rng}(\phi^{\vec{l'}}_{\alpha})$ by Lemma 14(1). But this is impossible since that means $\varphi \in \mathit{dom}(\phi^{\vec{l'}}_{\alpha})$ and $\varphi \in \mathit{rng}(\phi^{\vec{l'}}_{\alpha})$ but the domain and range of $\phi^{\vec{l'}}_{\alpha}$ must be disjoint.

4. $C' \cup \phi^{\vec{l'}}_{\alpha}(C'_1) \vdash \varphi \leq \rho$ and $\rho \rhd \ell_1 \in C' \cup C'_1$. Mirrors the second case, above.

5. $C' \cup C'_1 \cup \phi^{\vec{l'}}_{\alpha}(C'_1) \vdash \varphi \leq \rho$ and $\rho \rhd \ell_1 \in C' \cup C'_1$. If $\varphi \in \mathit{fl}(C' \cup C'_1)$ then by Lemma 14(2)(1) we have $C' \cup C'_1 \vdash \varphi \leq \rho$. Since $\rho \rhd \ell_1 \in C' \cup C'_1$ the reasoning for the first case applies. If $\varphi \in \mathit{fl}(C' \cup \phi^{\vec{l'}}_{\alpha}(C'_1))$ then by Lemma 14(2)(4) we have $C' \cup \phi^{\vec{l'}}_{\alpha}(C'_1) \vdash \varphi \leq \rho'$ where $\phi^{\vec{l'}}_{\alpha}(\rho) = \rho'$. We have $\phi^{\vec{l'}}_{\alpha}(\rho) \rhd \phi^{\vec{l'}}_{\alpha}(\ell_1) \in C' \cup \phi^{\vec{l'}}_{\alpha}(C'_1)$ by the fact that $\rho \rhd \ell_1 \in C' \cup C'_1$ and we can show $\phi^{\vec{l'}}_{\alpha}(\ell_1) \neq \ell_2$ as we did in 3(b), above. So the reasoning from case 3 applies, where we have $C' \cup \phi^{\vec{l'}}_{\alpha}(C'_1) \vdash \varphi \leq \phi^{\vec{l'}}_{\alpha}(\rho)$ and $\phi^{\vec{l'}}_{\alpha}(\rho) \rhd \phi^{\vec{l'}}_{\alpha}(\ell_1) \in C' \cup \phi^{\vec{l'}}_{\alpha}(C'_1)$ from the first derivation and $C' \cup C'_1 \cup \phi^{\vec{l'}}_{\alpha}(C'_1) \vdash \varphi \leq \ell_2$ as the second.

6. $C' \cup C'_1 \cup \phi^{\vec{l'}}_{\alpha}(C'_1) \vdash \varphi \leq \rho$ and $\rho \rhd \ell_1 \in C' \cup \phi^{\vec{l'}}_{\alpha}(C'_1)$. Mirrors the argument in the above case.

(6) $C' \cup C'_1 \vdash L_1 \leq^1 \ell \ \wedge \phi^{\vec{l'}}_{\alpha}(C'_1) \vdash L_2 \leq^1 \ell \Rightarrow L_1 = L_2$.

We must have that $\ell \notin \mathit{bl}(C_1) \cup \vec{\alpha}$ since it appears in both $C' \cup C'_1$ and $\phi^{\vec{l'}}_{\alpha}(C_1)'$. But in this case we have both $C' \cup C'_1 \vdash L_1 \leq^1 \ell$ and $C' \cup C'_1 \vdash L_2 \leq^1 \ell$ and $C' \cup \phi^{\vec{l'}}_{\alpha}(C'_1) \vdash L_1 \leq^1 \ell$ and $C' \cup \phi^{\vec{l'}}_{\alpha}(C'_1) \vdash L_2 \leq^1 \ell$ which by Lemma 13 implies that $L_1 = L_2$.

(7) $\vec{\alpha} \uplus \vec{\beta} \uplus \phi^{\vec{l'}}_{\alpha}(\vec{\beta})$ and $\varepsilon \uplus \phi^{\vec{l'}}_{\alpha}(\varepsilon_1)$ are well-defined.

For the first we can conclude that $(\vec{\alpha} \uplus \vec{\beta}) \cap \phi^{\vec{l'}}_{\alpha}(\vec{\beta}) = \emptyset$ by $\mathit{fl}(C' \cup C'_1) \cap \phi^{\vec{l'}}_{\alpha}(\vec{\beta}) = \emptyset$ from case (3) above, since $\mathit{fl}(C' \cup C'_1) \supseteq \vec{\alpha} \uplus \vec{\beta}$. The second follows from the fact that $\vec{l'} \supseteq \varepsilon$ and thus $\mathit{rng}(\phi^{\vec{l'}}_{\alpha}) \cap \varepsilon = \emptyset$. □

A.4 Soundness

Proving soundness involves proving the standard substitution lemmas and preservation (a.k.a. subject reduction). We present some weakening lemmas first, then the substitution lemmas, and finally the proof of preservation.

A.4.1 Weakening Lemmas

Definition 16 (Constraint Entailment) *$C' \vdash C$ iff $\forall c \in C.\ C' \vdash c$.*

Lemma 17 (Entailment Implication)

1. *If $C' \supseteq C$ then $C' \vdash C$.*
2. *If $C' \vdash C$ then $C \vdash c$ implies $C' \vdash c$.*

Proof: Proof by induction on $C \vdash c$. □

Lemma 18 (Constraint weakening in subtyping) *If $C \vdash \tau \leq \tau'$ then for any C' such that $C' \vdash C$ it holds that $C' \vdash \tau \leq \tau'$.*

Proof: By induction on $C \vdash \tau \leq \tau'$. □

Lemma 19 (Constraint weakening in typing) *If $C; \Gamma \vdash e : \tau; \varepsilon$ and $C' \vdash C$ then $C'; \Gamma \vdash e : \tau; \varepsilon$.*

Proof: By induction on $C; \Gamma \vdash e : \tau; \varepsilon$. Most cases follow either trivially (e.g., [Int],[Unit], and [Id]) or by applying induction on the subderivations along with Lemma 17 to prove $C' \vdash L \leq^1 \ell$ or $C' \vdash \varphi \leq \rho$ or $C' \vdash \rho \rhd \ell$, as appropriate. For [Sub] we appeal to Lemma 18. Here are the more interesting polymorphic cases.

Case [Let]. We have

$$[\text{Let}]\frac{C''; \Gamma \vdash_{cp} v_1 : \tau_1; \emptyset \qquad C; \Gamma, f : \forall \vec{\alpha}[C''].\tau_1 \vdash_{cp} e_2 : \tau_2; \varepsilon_2 \qquad \vec{\alpha} \subseteq (\mathit{ftl}(\tau_1) \cup \mathit{ftl}(C'')) \setminus \mathit{ftl}(\Gamma)}{C; \Gamma \vdash_{cp} \texttt{let}\ f = v_1\ \texttt{in}\ e_2 : \tau_2; \varepsilon_2}$$

By induction, $C'; \Gamma, f : \forall \vec{\alpha}[C''].\tau_1 \vdash_{cp} e_2 : \tau_2; \varepsilon_2$. Thus we can apply [Let] to show $C'; \Gamma \vdash_{cp} \texttt{let}\ f = e_1\ \texttt{in}\ e_2 : \tau_2; \varepsilon_2$.

Case [Fix]. We have

$$[\text{Fix}]\frac{C''; \Gamma, f : \forall \vec{\alpha}[C''].\tau \vdash_{cp} v : \tau; \emptyset \qquad \vec{\alpha} \subseteq (\mathit{ftl}(\tau) \cup \mathit{ftl}(C'')) \setminus \mathit{ftl}(\Gamma) \qquad C \vdash \phi(C'') \qquad \mathit{dom}(\phi) = \vec{\alpha}}{C; \Gamma \vdash_{cp} \texttt{fix}\, f.v : \phi(\tau); \emptyset}$$

Then since $C' \vdash C$ and $C \vdash \phi(C'')$, we have $C' \vdash \phi(C'')$ by Lemma 17. Thus we can apply [Fix] to yield $C'; \Gamma \vdash_{cp} \texttt{fix}\, f.v : \phi(\tau_1); \varepsilon$.

Case [Inst]. Similar to [Fix].

Case [Down]. We have

$$[\text{Down}]\frac{\begin{array}{c} C \cup \{\nu\vec{\alpha}[C_1; \varepsilon_1]\} \cup \mathit{strip}(\alpha^{\vec{l'}}(\nu\vec{\alpha}[C_1; \varepsilon_1])); \Gamma \vdash_{cp} e : \tau; \varepsilon \uplus \phi_\alpha^{\vec{l'}}(\varepsilon_1) \\ \phi_\alpha^{\vec{l'}}(\vec{\alpha}) \cap (\mathit{ftl}(\Gamma) \cup \mathit{ftl}(\tau)) = \emptyset \\ \varepsilon_1 \subseteq \vec{\alpha} \qquad \vec{l'} \supseteq \mathit{ftl}(\mathit{strip}^*(C) \cup \mathit{strip}^*(\nu\vec{\alpha}[C_1; \varepsilon_1])) \cup \varepsilon \end{array}}{C \cup \{\nu\vec{\alpha}[C_1; \varepsilon_1]\}; \Gamma \vdash_{cp} e : \tau; \varepsilon}$$

Since $C' \vdash C \cup \{\nu\vec{\alpha}[C_1; \varepsilon_1]\}$, we must have

$$[\text{Encap-Flow}]\frac{\nu\vec{\alpha}[C_0; \varepsilon_1] \in C' \qquad C_0 \vdash C_1}{C' \vdash \nu\vec{\alpha}[C_1; \varepsilon_1]}$$

And thus $C' \equiv (C'' \cup \{\nu\vec{\alpha}[C_0; \varepsilon_1]\})$. It follows that $\alpha^{\vec{l'}}(\nu\vec{\alpha}[C_0; \varepsilon_1]) \vdash \alpha^{\vec{l'}}(\nu\vec{\alpha}[C_1; \varepsilon_1])$, and thus $\mathit{strip}(\alpha^{\vec{l'}}(\nu\vec{\alpha}[C_0; \varepsilon_1])) \vdash \mathit{strip}(\alpha^{\vec{l'}}(\nu\vec{\alpha}[C_1; \varepsilon_1]))$ by inversion. With this we have $C'' \cup \{\nu\vec{\alpha}[C_0; \varepsilon_1]\} \cup \mathit{strip}(\alpha^{\vec{l'}}(\nu\vec{\alpha}[C_0; \varepsilon_1])) \vdash C \cup \{\nu\vec{\alpha}[C_1; \varepsilon_1]\} \cup \mathit{strip}(\alpha^{\vec{l'}}(\nu\vec{\alpha}[C_1; \varepsilon_1]))$ and so by induction it follows that $C'' \cup \nu\vec{\alpha}[C_0; \varepsilon_1] \cup \mathit{strip}(\alpha^{\vec{l'}}(\nu\vec{\alpha}[C_0; \varepsilon_1])); \Gamma \vdash_{cp} e : \tau; \varepsilon \uplus \phi_\alpha^{\vec{l'}}(\varepsilon_1)$.

We wish to apply [Down] to achieve the final result, where the above forms the first premise, so now we must establish the rest. Assume without loss of generality that

$$\vec{l'} \supseteq \mathit{ftl}(\mathit{strip}^*(C'') \cup \mathit{strip}^*(C) \cup \mathit{strip}^*(\nu\vec{\alpha}[C_1; \varepsilon_1])) \cup \mathit{strip}^*(\nu\vec{\alpha}[C_0; \varepsilon_1])) \cup \mathit{ftl}(\Gamma) \cup \mathit{ftl}(\tau) \cup \varepsilon$$

which satisfies our assumptions that

$$\begin{array}{c} \phi_\alpha^{\vec{l'}}(\vec{\alpha}) \cap (\mathit{ftl}(\Gamma) \cup \mathit{ftl}(\tau)) = \emptyset \text{ and} \\ \vec{l'} \supseteq \mathit{ftl}(\mathit{strip}^*(C) \cup \mathit{strip}^*(\nu\vec{\alpha}[C_1; \varepsilon_1])) \cup \varepsilon \end{array}$$

But then we also have that $\vec{l'} \supseteq \mathit{fl}(\mathit{strip}^*(C) \cup \mathit{strip}^*(\nu\vec{\alpha}[C_0; \varepsilon_1])) \cup \varepsilon$, and the other two premises hold by assumption, so we can apply [Down] to achieve the final result. $\square$

Lemma 20 (Polymorphic constraint weakening in typing) *If* $C; \Gamma, f : \forall\vec{\alpha}[C''].\tau \vdash e : \tau; \varepsilon$ *then* $C; \Gamma, f : \forall\vec{\alpha}[C'' \cup C'].\tau \vdash e : \tau; \varepsilon$ *where* $C \vdash C'$ *and* $\mathit{fl}(C') \cap \vec{\alpha} = \emptyset$.

Proof: Proof by induction on $C; \Gamma, f : \forall\vec{\alpha}[C''].\tau \vdash e : \tau; \varepsilon$. Most cases are trivial or by induction; the interesting cases are [Inst] and [Fix].
Case [Inst]. If $f \neq g$ then we have

$$[\text{Inst}]\frac{C \vdash \phi(C''') \qquad \mathit{dom}(\phi) = \vec{\beta}}{C; \Gamma, f : \forall\vec{\alpha}[C''].\tau, g : \forall\vec{\beta}[C'''].\tau \vdash_{cp} g^i : \phi(\tau); \emptyset}$$

The result follows trivially. Otherwise, we have

$$[\text{Inst}]\frac{C \vdash \phi(C'') \qquad \mathit{dom}(\phi) = \vec{\alpha}}{C; \Gamma, f : \forall\vec{\alpha}[C''].\tau \vdash_{cp} f^i : \phi(\tau); \emptyset}$$

So awe must prove for some ϕ' that $C \vdash \phi'(C'' \cup C')$. Let $\phi' = \phi$. By alpha-renaming we have $\mathit{fl}(C') \cap \vec{\alpha} = \emptyset$, so $\phi'(C') = C'$. Thus $C \vdash \phi'(C'') \cup C'$ since $C \vdash \phi'(C'')$ and $C \vdash C'$ by assumption.

Case [Fix]. We can assume $f \neq g$ by alpha-renaming, with

$$[\text{Fix}]\frac{\begin{array}{c} C'; \Gamma, f : \forall\vec{\alpha}[C''].\tau, g : \forall\vec{\beta}[C'''].\tau \vdash_{cp} v : \tau; \emptyset \\ \vec{\beta} \subseteq (\mathit{fl}(\tau) \cup \mathit{fl}(C''')) \setminus \mathit{fl}(\Gamma) \qquad C \vdash \phi(C''') \qquad \mathit{dom}(\phi) = \vec{\beta} \end{array}}{C; \Gamma, f : \forall\vec{\alpha}[C''].\tau \vdash_{cp} \mathtt{fix}\, g.v : \phi(\tau); \emptyset}$$

The result follows trivially by induction. $\square$

A.4.2 Substitution

Lemma 21 (Substitution lemma) *If* $C; \Gamma, x : \tau' \vdash_{cp} e : \tau; \varepsilon$ *and* $C \vdash C'$ *and* $C'; \Gamma \vdash_{cp} e' : \tau'; \emptyset$, *then* $C; \Gamma \vdash_{cp} e[x \mapsto e'] : \tau; \varepsilon$.

Proof: Notice that we only allow substituting with expressions e' that have no effect. The proof proceeds by induction on $C; \Gamma, x : \tau' \vdash_{cp} e : \tau; \varepsilon$.
Case [Id]. There are two cases. First, if $e = x$, we have

$$[\text{Id}]\frac{}{C; \Gamma, x : \tau' \vdash_{cp} x : \tau'; \emptyset}$$

Then $\tau = \tau'$, and since $x[x \mapsto e'] = e'$, by our assumption $C'; \Gamma \vdash_{cp} e' : \tau'; \emptyset$ and Lemma 19 we have $C; \Gamma \vdash_{cp} e' : \tau'; \emptyset$. Otherwise, we have

$$[\text{Id}]\frac{}{C; \Gamma, x : \tau \vdash_{cp} y : \tau; \emptyset}$$

where $y \neq x$. Since $y[x \mapsto e'] = y$, we have the result by assumption and a trivial strengthening of Γ.

Case [Int]. Trivial.

Case [Lam]. We have

$$[\text{Lam}]\frac{C; \Gamma, x : \tau', y : \tau_1 \vdash_{cp} e_2 : \tau_2; \varepsilon}{C; \Gamma, x : \tau' \vdash_{cp} \lambda y.e_2 : \tau_1 \to^{\varepsilon} \tau_2; \emptyset}$$

Using alpha renaming we can assume $y \neq x$, and hence $C; \Gamma, y : \tau_1, x : \tau' \vdash_{cp} e_2 : \tau_2; \varepsilon$. Then by induction we have $C; \Gamma, y : \tau_1 \vdash_{cp} e_2[x \mapsto e'] : \tau_2; \varepsilon$. Thus we can apply [Lam] to yield $C; \Gamma \vdash_{cp} (\lambda y.e_2)[x \mapsto e'] : \tau_1 \to^{\varepsilon} \tau_2; \emptyset$.

Case [App]. We have

$$[\text{App}]\frac{\begin{array}{c} C; \Gamma, x : \tau' \vdash_{cp} e_1 : \tau_2 \to^{\varepsilon} \tau; \varepsilon_1 \\ C; \Gamma, x : \tau' \vdash_{cp} e_2 : \tau_2; \varepsilon_2 \end{array}}{C; \Gamma, x : \tau' \vdash_{cp} e_1\ e_2 : \tau; \varepsilon_1 \uplus \varepsilon_2 \uplus \varepsilon}$$

Then by induction, we have $C; \Gamma \vdash_{cp} e_1[x \mapsto e'] : \tau_2 \to^{\varepsilon} \tau; \varepsilon_1$ and $C; \Gamma \vdash_{cp} e_2[x \mapsto e'] : \tau_2; \varepsilon_2$. Therefore we can apply [App] to yield $C; \Gamma \vdash_{cp} (e_1\ e_2)[x \mapsto e'] : \tau; \varepsilon_1 \uplus \varepsilon_2 \uplus \varepsilon$.

Case [Pair], [Proj], [Cond], [Sub], [Ref], [Newlock], [Deref], [Assign], [Loc], [Lock]. By induction (similar to [App]).

Case [Let]. We have

$$[\text{Let}]\frac{\begin{array}{c} C''; \Gamma, x : \tau' \vdash_{cp} v_1 : \tau_1; \emptyset \\ C; \Gamma, x : \tau', f : \forall \vec{\alpha}[C''].\tau_1 \vdash_{cp} e_2 : \tau_2; \varepsilon_2 \\ \vec{\alpha} \subseteq (\mathit{fl}(\tau_1) \cup \mathit{fl}(C'')) \setminus \mathit{fl}(\Gamma, x : \tau') \end{array}}{C; \Gamma, x : \tau' \vdash_{cp} \texttt{let}\ f = v_1\ \texttt{in}\ e_2 : \tau_2; \varepsilon_2}$$

By Lemma 19 and induction, we have $C' \cup C''; \Gamma \vdash_{cp} v_1[x \mapsto e'] : \tau_1; \emptyset$. By Lemma 20 we have $C; \Gamma, x : \tau', f : \forall \vec{\alpha}[C'' \cup C'].\tau_1 \vdash_{cp} e_2 : \tau_2; \varepsilon_2$, and by alpha-conversion (since $f \neq x$) and induction we have $C; \Gamma, f : \forall \vec{\alpha}[C' \cup C''].\tau_1 \vdash e_2[x \mapsto e'] : \tau_2; \varepsilon_2$. We have

$$\begin{array}{rcl} \vec{\alpha} & \subseteq & (fl(\tau_1) \cup fl(C'')) \setminus (fl(\Gamma) \cup fl(\tau')) \\ & \subseteq & (fl(\tau_1) \cup fl(C'')) \setminus fl(\Gamma) \\ & \subseteq & (fl(\tau_1) \cup fl(C'') \cup fl(C')) \setminus fl(\Gamma) \end{array}$$

So the result follows by [Let].

Case [Fix]. We have

$$[\text{Fix}]\frac{\begin{array}{c} C''; \Gamma, x : \tau', f : \forall \vec{\alpha}[C''].\tau \vdash_{cp} v : \tau; \emptyset \\ \vec{\alpha} \subseteq (\mathit{fl}(\tau) \cup \mathit{fl}(C'')) \setminus \mathit{fl}(\Gamma, x : \tau') \\ C \vdash \phi(C'') \qquad dom(\phi) = \vec{\alpha} \end{array}}{C; \Gamma, x : \tau' \vdash_{cp} \texttt{fix}\ f.v : \phi(\tau); \emptyset}$$

By Lemma 19 and Lemma 20 we have $C'' \cup C'; \Gamma, x : \tau', f : \forall\vec{\alpha}[C'' \cup C'].\tau \vdash_{cp} v : \tau; \emptyset$, so by alpha-conversion (since $f \neq x$) and induction we have $C'' \cup C'; \Gamma, f : \forall\vec{\alpha}[C'' \cup C'].\tau \vdash_{cp} v[x \mapsto e'] : \tau; \emptyset$. As per the reasoning in [Let], $\vec{\alpha} \subseteq (\mathit{fl}(\tau_1) \cup \mathit{fl}(C'' \cup C')) \setminus \mathit{fl}(\Gamma)$. By our alpha-renaming convention, we have $\mathit{fl}(C') \cap \vec{\alpha} = \emptyset$, so $\phi(C') = C'$ and thus $C \vdash \phi(C'' \cup C')$ since $C \vdash C'$ and $C \vdash \phi(C'')$. The result follows from [Fix].

Case [Inst]. We have

$$[\text{Inst}]\frac{C \vdash \phi(C'') \qquad dom(\phi) = \vec{\alpha}}{C; \Gamma, x : \tau', f : \forall\vec{\alpha}[C''].\tau \vdash_{cp} f^i : \phi(\tau); \emptyset}$$

Since $f_i[x \mapsto e'] = f_i$ (x and f are different syntactic forms) the result follows by assumption and a trivial strengthening of Γ.

Case [Down]. We have

$$[\text{Down}]\frac{\begin{array}{c} C \cup \{\nu\vec{\alpha}[C_1; \varepsilon_1]\} \cup \mathit{strip}(\alpha^{\vec{l'}}(\nu\vec{\alpha}[C_1; \varepsilon_1])); \Gamma, x : \tau' \vdash_{cp} e : \tau; \varepsilon \uplus \phi_\alpha^{\vec{l'}}(\varepsilon_1) \\ \phi_\alpha^{\vec{l'}}(\vec{\alpha}) \cap (\mathit{fl}(\Gamma, x : \tau') \cup \mathit{fl}(\tau)) = \emptyset \\ \varepsilon_1 \subseteq \vec{\alpha} \qquad \vec{l'} \supseteq \mathit{fl}(\mathit{strip}^*(C) \cup \mathit{strip}^*(\nu\vec{\alpha}[C_1; \varepsilon_1])) \cup \varepsilon \end{array}}{C \cup \{\nu\vec{\alpha}[C_1; \varepsilon_1]\}; \Gamma, x : \tau' \vdash_{cp} e : \tau; \varepsilon}$$

Since $C \cup \{\nu\vec{\alpha}[C_1; \varepsilon_1]\} \vdash C'$ by assumption, it follows that $C \cup \{\nu\vec{\alpha}[C_1; \varepsilon_1]\} \cup \mathit{strip}(\alpha^{\vec{l'}}(\nu\vec{\alpha}[C_1; \varepsilon_1])) \vdash C'$. By induction $C \cup \{\nu\vec{\alpha}[C_1; \varepsilon_1]\} \cup \mathit{strip}(\alpha^{\vec{l'}}(\nu\vec{\alpha}[C_1; \varepsilon_1])); \Gamma \vdash_{cp} e[x \mapsto e'] : \tau; \varepsilon \uplus \phi_\alpha^{\vec{l'}}(\varepsilon_1)$.

We wish to show the result by [Down], where the first premise is the above, so we must establish the remaining premises. Since $\phi_\alpha^{\vec{l'}}(\vec{\alpha}) \cap (\mathit{fl}(\Gamma, x : \tau') \cup \mathit{fl}(\tau)) = \emptyset$ we have $\phi_\alpha^{\vec{l'}}(\vec{\alpha}) \cap (\mathit{fl}(\Gamma) \cup \mathit{fl}(\tau)) = \emptyset$, and the remaining premises follow by assumption. □

Lemma 22 (Polymorphic substitution lemma) *If* $C; \Gamma, f : \forall\vec{\alpha}[C'].\tau' \vdash_{cp} e : \tau; \varepsilon$ *and* $C'; \Gamma \vdash_{cp} e' : \tau'; \emptyset$ *where* $\vec{\alpha} \cap fl(\Gamma) = \emptyset$ *then* $C; \Gamma \vdash_{cp} e[f \mapsto e'] : \tau; \varepsilon$.

Proof: The proof proceeds by induction on $C; \Gamma, f : \forall\vec{\alpha}[C'].\tau' \vdash_{cp} e : \tau; \varepsilon$.
Case [Id]. Trivial, since $x[f \mapsto e'] = x$ (f and x are different syntactic forms).

Case [Int]. Trivial.

Case [Lam]. We have

$$[\text{Lam}]\frac{C; \Gamma, f : \forall\vec{\alpha}[C'].\tau', x : \tau_1 \vdash_{cp} e : \tau_2; \varepsilon}{C; \Gamma, f : \forall\vec{\alpha}[C'].\tau' \vdash_{cp} \lambda x.e : \tau_1 \to^{\varepsilon} \tau_2; \emptyset}$$

By alpha conversion, we can assume $\vec{\alpha} \cap fl(\tau_1) = \emptyset$ and $C'; \Gamma, x : \tau_1 \vdash_{cp} e' : \tau'; \emptyset$. Since $x \neq f$, by induction we have $C; \Gamma, x : \tau_1 \vdash_{cp} e[f \mapsto e'] : \tau_2; \varepsilon$. Then applying [Lam] we have $C; \Gamma \vdash_{cp} (\lambda x.e)[f \mapsto e'] : \tau_1 \to^{\varepsilon} \tau_2; \emptyset$.

Case [App]. We have

$$[\text{App}]\frac{\begin{array}{c} C; \Gamma, f : \forall\vec{\alpha}[C'].\tau' \vdash_{cp} e_1 : \tau_2 \to^{\varepsilon} \tau_1; \varepsilon_1 \\ C; \Gamma, f : \forall\vec{\alpha}[C'].\tau' \vdash_{cp} e_2 : \tau_2; \varepsilon_2 \end{array}}{C; \Gamma, f : \forall\vec{\alpha}[C'].\tau' \vdash_{cp} e_1\ e_2 : \tau_1; \varepsilon_1 \uplus \varepsilon_2 \uplus \varepsilon}$$

By induction, we have $C;\Gamma \vdash_{cp} e_1[f \mapsto e'] : \tau_2 \to^{\varepsilon} \tau_1;\varepsilon_1$ and $C;\Gamma \vdash_{cp} e_2[f \mapsto e'] : \tau_2;\varepsilon_2$. Then applying [App] yields $C;\Gamma \vdash_{cp} (e_1\ e_2)[f \mapsto e'] : \tau_1;\varepsilon_1 \uplus \varepsilon_2 \uplus \varepsilon$.

Case [Pair], [Proj], [Cond], [Sub], [Ref], [Newlock], [Loc], [Lock], [Deref], [Assign]. By induction (similar to [App]).

Case [Let]. We have

$$[\text{Let}]\frac{\begin{array}{c} C'';\Gamma, f : \forall\vec{\alpha}[C'].\tau' \vdash_{cp} v_1 : \tau_1;\emptyset \\ C;\Gamma, f : \forall\vec{\alpha}[C'].\tau', g : \forall\vec{\beta}[C''].\tau_1 \vdash_{cp} e_2 : \tau_2;\varepsilon_2 \\ \vec{\beta} \subseteq (fl(\tau_1) \cup fl(C'')) \setminus (fl(\Gamma) \cup fl(\forall\vec{\alpha}[C'].\tau')) \end{array}}{C;\Gamma, f : \forall\vec{\alpha}[C'].\tau' \vdash_{cp} \texttt{let}\ g = v_1\ \texttt{in}\ e_2 : \tau_2;\varepsilon_2}$$

By induction, $C'';\Gamma \vdash_{cp} v_1[f \mapsto e'] : \tau_1;\emptyset$. Assuming by alpha renaming that $f \neq g$, by induction we also have $C;\Gamma, g : \forall\vec{\beta}[C''].\tau_1 \vdash_{cp} e_2[f \mapsto e'] : \tau_2;\varepsilon_2$. Finally,

$$\begin{array}{rcl} \vec{\beta} & \subseteq & (fl(\tau_1) \cup fl(C'')) \setminus (fl(\Gamma) \cup fl(\forall\vec{\alpha}[C'].\tau')) \\ & \subseteq & (fl(\tau_1) \cup fl(C'')) \setminus fl(\Gamma) \end{array}$$

so we can apply [Let] to show $C;\Gamma \vdash_{cp} (\texttt{let}\ g = v_1\ \texttt{in}\ e_2)[f \mapsto e'] : \tau_2;\varepsilon_2$.

Case [Fix]. Similar to [Let].

Case [Inst]. Thus $e = g$ for some variable g. There are two cases. If $g \neq f$, then the conclusion holds trivially, since $g[f \mapsto e'] = g$. Otherwise, we have

$$[\text{Inst}]\frac{C \vdash \phi(C') \qquad dom(\phi) = \vec{\alpha}}{C;\Gamma, f : \forall\vec{\alpha}[C'].\tau \vdash_{cp} f^i : \phi(\tau);\emptyset}$$

By assumption, $C';\Gamma \vdash_{cp} e' : \tau';\emptyset$ so we know $\phi(C');\phi(\Gamma) \vdash_{cp} e' : \phi(\tau);\emptyset$. Since $\vec{\alpha} \cap fl(\Gamma) = \emptyset$, we then have $\phi(C');\Gamma \vdash_{cp} e' : \phi(\tau);\emptyset$. But $C \vdash \phi(C')$, and so by Lemma 19, $C;\Gamma \vdash_{cp} e' : \phi(\tau);\emptyset$, and so we have shown the conclusion, since $f^i[f \mapsto e'] = e'$.

Case [Down]. We have

$$[\text{Down}]\frac{\begin{array}{c} C \cup \{\nu\vec{\alpha}[C'';\varepsilon'']\} \cup strip(\alpha^{\vec{l'}}(\nu\vec{\alpha}[C'';\varepsilon''])); \Gamma, f : \forall\vec{\alpha}[C'].\tau' \vdash_{cp} e : \tau;\varepsilon \uplus \phi_\alpha^{\vec{l'}}(\varepsilon') \\ \phi_\alpha^{\vec{l'}}(\vec{\alpha}) \cap (fl(\Gamma, f : \forall\vec{\alpha}[C'].\tau') \cup fl(\tau)) = \emptyset \\ \varepsilon' \subseteq \vec{\alpha} \qquad \vec{l'} \supseteq fl(strip^*(C) \cup strip^*(\nu\vec{\alpha}[C'';\varepsilon''])) \cup \varepsilon \end{array}}{C \cup \{\nu\vec{\alpha}[C'';\varepsilon'']\}; \Gamma, f : \forall\vec{\alpha}[C'].\tau' \vdash_{cp} e : \tau;\varepsilon}$$

By induction $C \cup \{\nu\vec{\alpha}[C'';\varepsilon'']\} \cup strip(\alpha^{\vec{l'}}(\nu\vec{\alpha}[C'';\varepsilon'']));\Gamma \vdash_{cp} e[f \mapsto e'] : \tau;\varepsilon \uplus \phi_\alpha^{\vec{l'}}(\varepsilon')$. Since $\phi_\alpha^{\vec{l'}}(\vec{\alpha}) \cap (fl(\Gamma, f : \forall\vec{\alpha}[C'].\tau') \cup fl(\tau)) = \emptyset$ we have $\phi_\alpha^{\vec{l'}}(\vec{\alpha}) \cap (fl(\Gamma) \cup fl(\tau)) = \emptyset$; with the other premises by assumption, the result follows by [Down]. $\square$

A.4.3 Preservation

The preservation lemma establishes that if a program is well typed using a constraint set that is well-formed then its entire evaluation will exhibit consistent correlation. Note that the preservation property establishes a new constraint set C' for each evaluation step, where $C' \supseteq C$ (and thus $C' \vdash C$ by Lemma 17). This ensures that correlations are consistent—each R is correlated to a single, unchanging lock L—across the entire evaluation derivation.

Definition 23 (Valid Evaluation) *We write $C \vdash e \longrightarrow e'$ iff $e \equiv \mathbb{E}[!^{[L]}\, v^R]$ or $e \equiv \mathbb{E}[v'^R :=^{[L]} v]$ implies $S_g(C, R) = \{L\}$.*

Lemma 24 (Preservation) *If $C; \Gamma \vdash_{cp} e : \tau; \varepsilon$ where $\varepsilon \vdash_{ok} C$ and $e \longrightarrow e'$, then there exists some C', ε', s.t.*

1. $(\varepsilon' - \varepsilon) \cap \mathit{fl}(C) = \emptyset$
2. $C' \supseteq C$
3. $L \leq^1 \ell \in (C' - C) \Rightarrow \ell \in (\varepsilon - \varepsilon')$
4. $C' \vdash e \longrightarrow e'$
5. $\varepsilon' \vdash_{ok} C'$
6. $C'; \Gamma \vdash_{cp} e' : \tau; \varepsilon'$.

Proof: The proof is by induction on $C; \Gamma \vdash_{cp} e : \tau; \varepsilon$.

Case [Id], [Int], [Lam], [Lock], [Loc], [Inst]. These cases cannot happen, because we assume $e \longrightarrow e'$.

Case [Ref]. In this case, the term is $\texttt{ref}\ e$, and there are two possible reductions. In the first case, we have $\texttt{ref}\ e \longrightarrow \texttt{ref}\ e'$. By assumption, we have

$$[\text{Ref}]\frac{C; \Gamma \vdash_{cp} e : \tau; \varepsilon}{C; \Gamma \vdash_{cp} \texttt{ref}\ e : \mathit{ref}^{\rho}\, \tau; \varepsilon}$$

By induction, there exists C_i, ε_i s.t. $C_i; \Gamma \vdash_{cp} e' : \tau'; \varepsilon_i$; and $C_i \vdash e \longrightarrow e'$; and $\varepsilon_i \vdash_{ok} C_i$. Let $C' = C_i$ and $\varepsilon' = \varepsilon_i$. Then applying [Ref] yields $C'; \Gamma \vdash_{cp} \texttt{ref}\ e' : \mathit{ref}^{\rho}\, \tau; \varepsilon'$, and we also have $C' \vdash \texttt{ref}\ e \longrightarrow \texttt{ref}\ e'$ by applying the $\mathbb{E}[e] \longrightarrow \mathbb{E}[e']$ evaluation rule.

In the second case we have $\texttt{ref}\ v \longrightarrow v^R$. Let $\varepsilon' = \varepsilon = \emptyset$ and $C' = C \cup \{R \leq \rho\}$. Clearly $(\varepsilon' - \varepsilon) \cap \mathit{fl}(C) = \emptyset$ and $C' \supseteq C$ and $L \leq^1 \ell \in (C' - C = \{R \leq \rho\}) \Rightarrow \ell \in (\varepsilon - \varepsilon' = \emptyset)$. And $C' \vdash \texttt{ref}\ v \longrightarrow v^R$ follows trivially since no constructors were consumed. We can prove $C'; \Gamma \vdash_{cp} v^R : \mathit{ref}^{\rho}\, \tau; \emptyset$ as follows:

$$[\text{Loc}]\frac{C'; \Gamma \vdash_{cp} v : \tau; \emptyset \qquad [\text{Loc-Flow}]\ \dfrac{R \leq \rho \in C'}{C' \vdash R \leq \rho}}{C'; \Gamma \vdash_{cp} v^R : \mathit{ref}^{\rho}\, \tau; \emptyset}$$

where $C'; \Gamma \vdash_{cp} v : \tau; \emptyset$ follows by Lemma 19. Finally, we can prove $\varepsilon' \vdash_{ok} C'$ as follows:

$$\text{[Con-Union]}\frac{\begin{array}{c}\varepsilon' \vdash_{ok} C \hookrightarrow C''; \vec{\alpha} \qquad \text{[Con-Other]}\ \dfrac{}{\varepsilon' \vdash_{ok} \{R \le \rho\} \hookrightarrow \{R \le \rho\}; \emptyset} \\ \mathit{fl}(C'') \cap \emptyset = \emptyset \qquad \mathit{fl}(\{R \le \rho\}) \cap \vec{\alpha} = \emptyset \\ \text{for all } \varphi'.\ |S(C'' \cup \{R \le \rho\}, \varphi')| \le 1 \\ C'' \vdash L_1 \le^1 \ell \ \wedge\ \{R \le \rho\} \vdash L_2 \le^1 \ell \ \Rightarrow L_1 = L_2\end{array}}{\varepsilon' \vdash_{ok} C \cup \{R \le \rho\} \hookrightarrow C'' \cup \{R \le \rho\}; \vec{\alpha}}$$

Most of the premises follow trivially.

To prove $\mathit{fl}(\{R \le \rho\}) \cap \vec{\alpha} = \emptyset$, we observe that if $\rho \notin \mathit{fl}(C)$ then there are no conditions on its flow, so $\vec{\alpha}$ (where $\vec{\alpha} = \mathit{bl}(C)$) can be safely alpha-converted. Otherwise (if $\rho \in \mathit{fl}(C)$) ρ must not appear in $\vec{\alpha}$ or it would violate the assumption $\varepsilon \vdash_{ok} C$.

Finally, we must show for all φ'. $|S(C'' \cup \{R \le \rho\}, \varphi')| \le 1$. We have $S(C'', \varphi') \le 1$ by Lemma 13. Since $R \notin C''$ (by the fact that it was fresh), we have for all $\varphi \ne R$ that $C'' \cup \{R \le \rho\} \vdash \varphi \rhd \ell$ implies $C'' \vdash \varphi \rhd \ell$, and thus $|S(C'' \cup \{R \le \rho\})| \le 1$. Because $C'' \cup \{R \le \rho\} \vdash R \le \rho$, and $|S(C'', \rho)| \le 1$, then $|S(C'' \cup \{R \le \rho\}, R)| \le 1$ follows easily.

Case [App]. In this case, the term is $e_1\ e_2$, and there are three possible reductions. In the first case, when $e_1\ e_2 \longrightarrow e_1'\ e_2$, we have

$$\text{[App]}\frac{\begin{array}{c}C; \Gamma \vdash_{cp} e_1 : \tau_2 \to^{\varepsilon} \tau_1; \varepsilon_1 \\ C; \Gamma \vdash_{cp} e_2 : \tau_2; \varepsilon_2\end{array}}{C; \Gamma \vdash_{cp} e_1\ e_2 : \tau_1; \varepsilon_1 \uplus \varepsilon_2 \uplus \varepsilon}$$

Then by induction, there exists C_i, ε_i s.t. $(\varepsilon_i - \varepsilon_1) \cap \mathit{fl}(C) = \emptyset$; and $C_i \supseteq C$; and $L \le^1 \ell \in (C_i - C) \Rightarrow \ell \in (\varepsilon - \varepsilon_i)$; and $C_i \vdash e_1 \longrightarrow e_1'$; and $\varepsilon_i \vdash_{ok} C_i$ and $C_i; \Gamma \vdash_{cp} e_1' : \tau_2 \to^{\varepsilon} \tau_1; \varepsilon_i$.

Let $C' = C_i$ and $\varepsilon' = \varepsilon_i \uplus \varepsilon_2 \uplus \varepsilon$. We prove the latter is well-formed as follows. Consider some $\ell \in \varepsilon_i$. If $\ell \in \varepsilon_1$ then $\ell \notin (\varepsilon_2 \uplus \varepsilon)$ by assumption. If $\ell \notin \varepsilon_1$ then $\ell \notin \mathit{fl}(C)$ by induction. Thus, if $\ell \in (\varepsilon_2 \uplus \varepsilon)$, we can safely alpha-convert ℓ in ε_i and C_i.

We must show that $L \le^1 \ell \in (C_i - C) \Rightarrow \ell \in (\varepsilon_1 \uplus \varepsilon_2 \uplus \varepsilon) - (\varepsilon_i \uplus \varepsilon_2 \uplus \varepsilon)$. But since $(\varepsilon_1 \uplus \varepsilon_2 \uplus \varepsilon) - (\varepsilon_i \uplus \varepsilon_2 \uplus \varepsilon) = (\varepsilon_1 - \varepsilon_i)$ we have this by induction.

We have $(\varepsilon_i \uplus \varepsilon_2 \uplus \varepsilon) - (\varepsilon_1 \uplus \varepsilon_2 \uplus \varepsilon) = (\varepsilon_i - \varepsilon_1)$, and $(\varepsilon_i - \varepsilon_1) \cap \mathit{fl}(C) = \emptyset$ by induction. Since $C' \supseteq C$ by induction, by Lemmas 17 and 19 we have $C'; \Gamma \vdash_{cp} e_2 : \tau_2; \varepsilon_2$. Thus, by [App] we have $C'; \Gamma \vdash_{cp} e_1'\ e_2 : \tau_1; \varepsilon'$. Since $C' \vdash e_1 \longrightarrow e_1'$, we have $C' \vdash e_1\ e_2 \longrightarrow e_1'\ e_2$ by congruence.

Finally, we must show $\varepsilon' \vdash_{ok} C'$; that is, that $\varepsilon_i \uplus \varepsilon_2 \uplus \varepsilon \vdash_{ok} C_i$. If $C_i = C$ then $\varepsilon_1 \uplus \varepsilon_2 \uplus \varepsilon \vdash_{ok} C$ by assumption and $\varepsilon_i \vdash_{ok} C$ by induction, so we easily have $(\varepsilon_1 \cup \varepsilon_i) \uplus \varepsilon_2 \uplus \varepsilon \vdash_{ok} C$ and thus $\varepsilon_i \uplus \varepsilon_2 \uplus \varepsilon \vdash_{ok} C$ by Lemma 12(5). Otherwise $C_i = C \cup C''$ for some C'', so by induction and inversion we have

$$\text{[Con-Union]}\frac{\begin{array}{c}\varepsilon_i \vdash_{ok} C \hookrightarrow C'''; \vec{\alpha} \qquad \varepsilon_i \vdash_{ok} C'' \hookrightarrow C''''; \vec{\beta} \\ \mathit{fl}(C''') \cap \vec{\beta} = \emptyset \qquad \mathit{fl}(C'''') \cap \vec{\alpha} = \emptyset \\ \text{for all } \varphi'.\ |S(C''' \cup C'''', \varphi')| \le 1 \\ C''' \vdash L_1 \le^1 \ell \ \wedge\ C'''' \vdash L_2 \le^1 \ell \ \Rightarrow L_1 = L_2\end{array}}{\varepsilon_i \vdash_{ok} C \cup C'' \hookrightarrow C''' \cup C''''; \vec{\alpha} \uplus \vec{\beta}}$$

As argued above, we can show $\varepsilon_i \uplus \varepsilon_2 \uplus \varepsilon \vdash_{ok} C \hookrightarrow C'''; \vec{\alpha}$, so we must show $\varepsilon_i \uplus \varepsilon_2 \uplus \varepsilon \vdash_{ok} C'' \hookrightarrow C''''; \vec{\beta}$ and the rest follows by [Con-Union]. This follows because we have $\varepsilon_i \vdash_{ok} C'' \hookrightarrow C''; \vec{\beta}$ by

assumption, and we know by induction that if $L \leq^1 \ell \in C''$ then $\ell \in (\varepsilon_1 - \varepsilon_i)$. In other words $\ell \notin (\varepsilon_2 \uplus \varepsilon)$, so we can safely strengthen the effect and get $\varepsilon_i \uplus \varepsilon_2 \uplus \varepsilon \vdash_{ok} C'' \hookrightarrow C'''; \vec{\beta}$.

The second case, when $e_1\ e_2 \longrightarrow e_1\ e_2'$, is similar.

In the last case, we have $(\lambda x.e_1)\ v \longrightarrow e_1[x \mapsto v]$. In this case, we have

$$\text{[Sub]}\ \frac{\text{[Lam]}\ \dfrac{C; \Gamma, x : \tau_1' \vdash e_1 : \tau_2'; \varepsilon'}{C; \Gamma \vdash_{cp} \lambda x.e_1 : \tau_1' \to^{\varepsilon'} \tau_2'; \emptyset} \qquad \dfrac{\begin{array}{c} C \vdash \tau_1 \leq \tau_1' \\ C \vdash \tau_2' \leq \tau_2 \\ \varepsilon' \subseteq \varepsilon \end{array}}{C \vdash \tau_1' \to^{\varepsilon'} \tau_2' \leq \tau_1 \to^{\varepsilon} \tau_2}}{C; \Gamma \vdash_{cp} \lambda x.e_1 : \tau_1 \to^{\varepsilon} \tau_2; \emptyset} \qquad C; \Gamma \vdash_{cp} v : \tau_1; \emptyset$$
$$\frac{}{C; \Gamma \vdash (\lambda x.e_1)\ v : \tau_2; \varepsilon}$$

Choose $C' = C$ and $\varepsilon' = \varepsilon$. By Lemma 21, $C'; \Gamma \vdash_{cp} e_1[x \mapsto v] : \tau_2; \varepsilon'$. The remainder of the postconditions follow by trivially or by assumption.

Case [Pair,Proj,Cond]. Follows [App].

Case [Deref]. In this case, the term is $!^{e_2}\ e_1$, and the reasoning follows that of [App] for the inductive cases.

For the case that $!^{[L]}\ v^R \longrightarrow v$, we have

$$\text{[Deref]}\ \frac{\text{[Sub]}\ \dfrac{\text{[Loc]}\ \dfrac{\begin{array}{c} C; \Gamma \vdash_{cp} v : \tau'; \emptyset \\ C \vdash R \leq \rho' \end{array}}{C; \Gamma \vdash_{cp} v^R : ref^{\rho'}\ \tau'; \emptyset} \qquad \dfrac{\begin{array}{c} C \vdash \rho' \leq \rho \\ C \vdash \tau' \leq \tau \\ C \vdash \tau \leq \tau' \end{array}}{C \vdash ref^{\rho'}\ \tau' \leq ref^{\rho}\ \tau}}{C; \Gamma \vdash_{cp} v^R : ref^{\rho}\ \tau; \emptyset} \qquad \text{[Lock]}\ \frac{C \vdash L \leq^1 \ell}{C; \Gamma \vdash_{cp} [L] : lock\ \ell; \emptyset}}{C; \Gamma \vdash_{cp} !^{[L]}\ v^R : \tau; \emptyset}\ C \vdash \rho \rhd \ell$$

Let $C' = C$ and $\varepsilon' = \varepsilon$. Thus $C'; \Gamma \vdash_{cp} v : \tau; \emptyset$ follows by assumption and [Sub], and $\varepsilon' \vdash_{ok} C'$ and $C' \supseteq C$ and $(\varepsilon' - \varepsilon) \cap \mathit{fl}(C) = \emptyset$ and $L \leq^1 \ell \in (C \cap C') \Rightarrow \ell \in (\varepsilon - \varepsilon')$ follow trivially or by assumption.

To prove $C' \vdash e \longrightarrow e'$ we must prove $S_g(C', R) = \{L\}$. Since $\varepsilon' \vdash_{ok} C'$ we have $|S(C', \varphi)| \leq 1$ for all φ by Lemma 13(1). This implies $S(C', R) = \{\ell\}$ since $C' \vdash R \rhd \ell$:

$$\text{[Correlate]}\ \frac{C' \vdash \rho \rhd \ell \qquad C' \vdash R \leq \rho}{C' \vdash R \rhd \ell}$$

We have $C' \vdash L \leq^1 \ell$ by assumption, and by Lemma 13(2), we know that if $C' \vdash L' \leq^1 \ell$ then $L = L'$ and thus $S_g(C', R) = \{L\}$.

Case [Assign]. Similar to [Deref].

Case [Newlock]. In this case, $e \equiv \texttt{newlock}$ and so $\texttt{newlock} \longrightarrow [L]$ where $L \notin C$ since it's fresh. We have

$$\text{[Newlock]}\ \frac{}{C; \Gamma \vdash_{cp} \texttt{newlock} : lock\ \ell; \{\ell\}}$$

Let $C' = C \cup \{L \leq^1 \ell\}$ and $\varepsilon' = \emptyset$. Clearly $C' \supseteq C$ and since $((\varepsilon' = \emptyset) - (\varepsilon = \{\ell\})) = \emptyset$ we have $(\varepsilon' - \varepsilon) \cap \mathit{fl}(C) = \emptyset$. Moreover, $C' \cap C = \{L \leq \ell\}$ and $\varepsilon - \varepsilon' = \{\ell\}$ which proves $L \leq^1 \ell \in (C' - C) \Rightarrow \ell \in (\varepsilon - \varepsilon')$. We can prove

$$[\text{Lock}]\frac{C' \vdash L \leq^1 \ell}{C'; \Gamma \vdash_{cp} [L] : \mathit{lock}\ \ell; \varepsilon'}$$

We have $C' \vdash \texttt{newlock} \longrightarrow [L]$ trivially since no constructors are consumed. Finally, we prove $\varepsilon' \vdash_{ok} C'$ by applying [Con-Union] as follows:

$$[\text{Con-Union}]\frac{\begin{array}{c}\emptyset \vdash_{ok} C \hookrightarrow C''; \vec{\alpha} \qquad [\text{Con-Lock}]\ \dfrac{\ell \notin \emptyset}{\emptyset \vdash_{ok} \{L \leq^1 \ell\} \hookrightarrow \{L \leq^1 \ell\}; \emptyset} \\ \mathit{fl}(C'') \cap \emptyset = \emptyset \qquad \mathit{fl}(\{L \leq^1 \ell\}) \cap \vec{\alpha} = \emptyset \\ \text{for all } \varphi'.\ |S(C'' \cup \{L \leq^1 \ell\}, \varphi')| \leq 1 \\ C'' \vdash L_1 \leq^1 \ell' \ \wedge\ \{L \leq^1 \ell\} \vdash L_2 \leq^1 \ell' \ \Rightarrow L_1 = L_2\end{array}}{\emptyset \vdash_{ok} C \cup \{L \leq^1 \ell\} \hookrightarrow C'' \cup \{L \leq^1 \ell\}; \vec{\alpha}}$$

We prove $\emptyset \vdash_{ok} C \hookrightarrow C''; \vec{\alpha}$ by assumption and weakening (Lemma 12(5)). We prove $\mathit{fl}(\{L \leq^1 \ell\}) \cap \vec{\alpha} = \emptyset$ following the argument in [Ref], above. The premise for consistent correlation follows trivially, because the addition of constraints $L \leq^1 \ell$ does not affect which correlations one can prove. Finally, since $\varepsilon = \{\ell\}$, by $\varepsilon \vdash_{ok} C \hookrightarrow C''; \vec{\alpha}$ and Lemma 12(3) we have $C \not\vdash L \leq^1 \ell$ for all L, so the last premise follows by assumption for all $\ell' \neq \ell$ and vacuously for ℓ.

Case [Let]. In this case, $\texttt{let}\ f = v_1\ \texttt{in}\ e_2 \longrightarrow (e_2[f \mapsto v_1])$, and

$$[\text{Let}]\frac{\begin{array}{c}C''; \Gamma \vdash_{cp} v_1 : \tau_1; \emptyset \qquad C; \Gamma, f : \forall \vec{\alpha}[C''].\tau_1 \vdash_{cp} e_2 : \tau_2; \varepsilon \\ \vec{\alpha} \subseteq (\mathit{fl}(\tau_1) \cup \mathit{fl}(C'')) \setminus \mathit{fl}(\Gamma)\end{array}}{C; \Gamma \vdash_{cp} \texttt{let}\ f = e_1\ \texttt{in}\ e_2 : \tau_2; \varepsilon}$$

Let $C' = C$ and $\varepsilon' = \varepsilon$. Thus $(\varepsilon' - \varepsilon) \cap \mathit{fl}(C) = \emptyset$ and $C' \supseteq C$ and $L \leq^1 \ell \in (C' - C) \Rightarrow \ell \in (\varepsilon - \varepsilon')$ and $\varepsilon' \vdash_{ok} C'$ and $C' \vdash e \longrightarrow e'$ follow trivially or by assumption. Since we can assume that $\vec{\alpha} \cap \mathit{fl}(\Gamma) = \emptyset$ by alpha-renaming, by Lemma 22 we have $C'; \Gamma \vdash e_2[f \mapsto e_1] : \tau_2; \varepsilon'$.

Case [Fix]. In this case $\texttt{fix}\ f.v \longrightarrow v[f \mapsto \texttt{fix}\ f.v]$ and

$$[\text{Fix}]\frac{\begin{array}{c}C''; \Gamma, f : \forall \vec{\alpha}[C''].\tau \vdash_{cp} v : \tau; \emptyset \\ \vec{\alpha} \subseteq (\mathit{fl}(\tau) \cup \mathit{fl}(C'')) \setminus \mathit{fl}(\Gamma) \qquad C \vdash \phi(C'') \qquad \mathit{dom}(\phi) = \vec{\alpha}\end{array}}{C; \Gamma \vdash_{cp} \texttt{fix}\ f.v : \phi(\tau); \emptyset}$$

Let $C' = C$ and $\varepsilon' = \varepsilon = \emptyset$. Thus $(\varepsilon' - \varepsilon) \cap \mathit{fl}(C) = \emptyset$ and $C' \supseteq C$ and $L \leq^1 \ell \in (C' - C) \Rightarrow \ell \in (\varepsilon - \varepsilon')$ and $\varepsilon' \vdash_{ok} C'$ and $C' \vdash e \longrightarrow e'$ follow trivially or by assumption. For the substitution ϕ that maps all labels in $\vec{\alpha}$ to themselves, we can apply [Fix] to show

$$[\text{Fix}]\frac{\begin{array}{c}C''; \Gamma, f : \forall \vec{\alpha}[C''].\tau \vdash_{cp} v : \tau; \emptyset \\ \vec{\alpha} \subseteq (\mathit{fl}(\tau) \cup \mathit{fl}(C'')) \setminus \mathit{fl}(\Gamma) \qquad C'' \vdash C''\end{array}}{C''; \Gamma \vdash_{cp} \texttt{fix}\ f.v : \tau; \emptyset}$$

Finally, from these facts, and since we can assume that $\vec{\alpha} \cap \mathit{fl}(\Gamma) = \emptyset$ by alpha-renaming, by Lemma 22 we have $C'; \Gamma \vdash v[f \mapsto \mathtt{fix}\, f.v] : \tau; \emptyset$.

Case [Down]. In this case we have $e \longrightarrow e'$ and

$$\text{[Down]}\frac{\begin{array}{c} C \cup \{\nu\vec{\alpha}[C_1;\varepsilon_1]\} \cup \mathit{strip}(\alpha^{\vec{l'}}(\nu\vec{\alpha}[C_1;\varepsilon_1])); \Gamma \vdash_{cp} e : \tau; \varepsilon \uplus \phi_\alpha^{\vec{l'}}(\varepsilon_1) \\ \phi_\alpha^{\vec{l'}}(\vec{\alpha}) \cap (\mathit{fl}(\Gamma) \cup \mathit{fl}(\tau)) = \emptyset \\ \varepsilon_1 \subseteq \vec{\alpha} \qquad \vec{l'} \supseteq \mathit{fl}(\mathit{strip}^*(C) \cup \mathit{strip}^*(\nu\vec{\alpha}[C_1;\varepsilon_1])) \cup \varepsilon \end{array}}{C \cup \{\nu\vec{\alpha}[C_1;\varepsilon_1]\}; \Gamma \vdash_{cp} e : \tau; \varepsilon}$$

Since $\varepsilon \vdash_{ok} C \cup \{\nu\vec{\alpha}[C_1;\varepsilon_1]\} \hookrightarrow C' \cup C'_1; \vec{\alpha} \uplus \vec{\beta}$ by assumption (and inversion via [Con-Union] and [Con-Encap]), we have $\varepsilon \uplus \phi_\alpha^{\vec{l'}}(\varepsilon_1) \vdash_{ok} C \cup \{\nu\vec{\alpha}[C_1;\varepsilon_1]\} \cup \mathit{strip}(\alpha^{\vec{l'}}(\nu\vec{\alpha}[C_1;\varepsilon_1])) \hookrightarrow C' \cup C'_1 \cup \phi_\alpha^{\vec{l'}}(C'_1); \vec{\alpha} \uplus \vec{\beta} \uplus \phi_\alpha^{\vec{l'}}(\vec{\beta})$ by Lemma 15. Thus by induction there exists some C_i, ε_i s.t. $C_i \supseteq C \cup \{\nu\vec{\alpha}[C_1;\varepsilon_1]\} \cup \mathit{strip}(\alpha^{\vec{l'}}(\nu\vec{\alpha}[C_1;\varepsilon_1]))$; and $(\varepsilon_i - (\varepsilon \uplus \phi_\alpha^{\vec{l'}}(\varepsilon_1))) \cap \mathit{fl}(C \cup \{\nu\vec{\alpha}[C_1;\varepsilon_1]\} \cup \mathit{strip}(\alpha^{\vec{l'}}(\nu\vec{\alpha}[C_1;\varepsilon_1]))) = \emptyset$ and $L \leq^1 \ell \in (C_i - (C \cup \{\nu\vec{\alpha}[C_1;\varepsilon_1]\} \cup \mathit{strip}(\alpha^{\vec{l'}}(\nu\vec{\alpha}[C_1;\varepsilon_1])))) \Rightarrow \ell \in ((\varepsilon \uplus \phi_\alpha^{\vec{l'}}(\varepsilon_1)) - \varepsilon_i)$ and $C_i \vdash e \longrightarrow e'$; and $\varepsilon_i \vdash_{ok} C_i$; and $C_i; \Gamma \vdash_{cp} e' : \tau; \varepsilon_i$.

Let $C' = C_i$ and $\varepsilon' = \varepsilon_i$, so that $C' \supseteq C \cup \{\nu\vec{\alpha}[C_1;\varepsilon_1]\}$ and $\varepsilon' \vdash_{ok} C'$ and $C' \vdash e \longrightarrow e'$ and $C'; \Gamma \vdash_{cp} e' : \tau; \varepsilon'$ follow trivially.

We must show $L \leq^1 \ell \in (C_i - (C \cup \{\nu\vec{\alpha}[C_1;\varepsilon_1]\})) \Rightarrow \ell \in \varepsilon_i$. We have by induction that this property holds for constraints $(C_i - (C \cup \{\nu\vec{\alpha}[C_1;\varepsilon_1]\} \cup \mathit{strip}(\alpha^{\vec{l'}}(\nu\vec{\alpha}[C_1;\varepsilon_1])))) = C''$. Since, by the fact that $C_i \supseteq C \cup \{\nu\vec{\alpha}[C_1;\varepsilon_1]\} \cup \mathit{strip}(\alpha^{\vec{l'}}(\nu\vec{\alpha}[C_1;\varepsilon_1]))$ we have $C_i - (C \cup \{\nu\vec{\alpha}[C_1;\varepsilon_1]\}) = C'' \cup \mathit{strip}(\alpha^{\vec{l'}}(\nu\vec{\alpha}[C_1;\varepsilon_1]))$, but we know that $\nu\vec{\alpha}[C_1;\varepsilon_1]$ must not contain any lock allocation constraints since it was deemed well-formed by assumption.

Finally, we must show that $(\varepsilon_i - \varepsilon) \cap \mathit{fl}(C \cup \{\nu\vec{\alpha}[C_1;\varepsilon_1]\}) = \emptyset$. For some $\ell \in (\varepsilon_i - \varepsilon)$ there are two possibilities:

1. Assume $\ell \in (\varepsilon_i - (\varepsilon \uplus \phi_\alpha^{\vec{l'}}(\varepsilon_1)))$. By induction (as stated above) $\ell \cap \mathit{fl}(C \cup \{\nu\vec{\alpha}[C_1;\varepsilon_1]\} \cup \mathit{strip}(\alpha^{\vec{l'}}(\nu\vec{\alpha}[C_1;\varepsilon_1]))) = \emptyset$, and thus $\ell' \cap \mathit{fl}(C \cup \{\nu\vec{\alpha}[C_1;\varepsilon_1]\}) = \emptyset$ trivially.

2. Assume $\ell \in (\phi_\alpha^{\vec{l'}}(\varepsilon_1) \cap \varepsilon_i)$; i.e. there is some $\ell' \in \varepsilon_1$ s.t. $\phi_\alpha^{\vec{l'}}(\ell') = \ell \in \varepsilon_i$. But then we have $\ell \notin \mathit{fl}(C \cup \{\nu\vec{\alpha}[C_1;\varepsilon_1]\})$ since (1) $\ell' \in \mathit{dom}(\phi_\alpha^{\vec{l'}})$ by the fact that $\mathit{dom}(\phi_\alpha^{\vec{l'}}) = \vec{\alpha}$ and $\ell' \in \varepsilon_1 \subseteq \vec{\alpha}$; (2) $\ell \notin \vec{l'}$ by the fact that $\mathit{rng}(\phi_\alpha^{\vec{l'}}) \cap \vec{l'} = \emptyset$ by the definition of $\phi_\alpha^{\vec{l'}}$; and (3) since $\vec{l'} \supseteq (\mathit{fl}(\mathit{strip}^*(C)) \cup \mathit{fl}(\mathit{strip}^*(\nu\vec{\alpha}[C_1;\varepsilon_1])) \cup \varepsilon)) \supseteq \mathit{fl}(C \cup \{\nu\vec{\alpha}[C_1;\varepsilon_1]\})$.

□

Thus, if $C; \Gamma \vdash_{cp} e : \tau; \varepsilon$ and $\varepsilon \vdash_{ok} C$ then there exists a (possibly infinite) list of pairs C_i, ε_i for which $C_i \supseteq C_{i-1}$. If $e \longrightarrow e_1 \longrightarrow e_2 \ldots$ then $C \vdash e \longrightarrow e_1$, and $C_1 \vdash e_1 \longrightarrow e_2$, and $C_2 \vdash e_2 \longrightarrow e_3$ and so on, which means that each dereference or assignment to R is valid in C_i, being correlated with a single lock L in S_g. Moreover, by $\varepsilon_i \vdash_{ok} C_i$ it follows from Lemma 13 that $|S_g(C_i, R)| \leq 1$ for all R. Since $C_i \supseteq C_{i-1}$, we know $S_g(C_i, R) = \{L\}$ implies $S_g(C_j, R) = \{L\}$ for all $j \geq i$, and thus each R that is dereferenced is correlated with the same single lock for the entire evaluation of e.

$$[\text{Loc-Trans}]\frac{C \vdash \rho_0 \leq \rho_l \quad C \vdash \rho_1 \leq \rho_2}{C \vdash \rho_0 \leq \rho_2}$$

$$[\text{Loc-Match}]\frac{C \vdash \rho_1 \preceq^i_- \rho_0 \quad C \vdash \rho_1 \leq \rho_2 \quad C \vdash \rho_2 \preceq^i_+ \rho_3}{C \vdash \rho_0 \leq \rho_3}$$

(a) Location and Lock Flow

$$[\text{Corr-Trans}]\frac{C \vdash \rho \leq \rho' \quad C \vdash \rho' \rhd \ell}{C \vdash \rho \rhd \ell}$$

$$[\text{Corr-Match}]\frac{C \vdash \rho \preceq^i_p \rho' \quad C \vdash \rho \rhd \ell \quad C \vdash \ell \preceq^i \ell'}{C \vdash \rho' \rhd \ell'}$$

(b) Correlation Flow

Figure 20: Constraint Flow

```
let l' = newlock in
let x = ref 0 in
let f l = (x :=^l 0) in
  f l';
  x :=^l' 0
```

Figure 21: Program showing a difference between $\lambda^{cp}_{\rhd}$ and $\lambda_{\rhd}$

B Reduction from $\lambda_{\rhd}$ to $\lambda^{cp}_{\rhd}$

Next we prove the soundness of $\lambda_{\rhd}$ by showing that all $\lambda_{\rhd}$ derivations can be reduced to $\lambda^{cp}_{\rhd}$ derivations. Recall that the type rules for $\lambda_{\rhd}$ are shown in Figures 5–7. In order to reason about the lock and location resolution rules in Figure 8, we reformulate them as inference rules, as shown in Figure 20. Recall that our constraint resolution rules use $C \vdash \mathit{escapes}(\ell, \vec{l})$ (defined on page 10). In words, we have $\mathit{escapes}(\ell, \vec{l})$ if ℓ is connected in any way to $\vec{l}$, either via a semiunification constraint or via correlation with a location ρ that is connected in some way to $\vec{l}$.

Recall that after applying the inference rules, there are three conditions we need to check. First, we need to ensure that all disjoint unions formed during type inference and constraint resolution are truly disjoint. We define $\mathit{occurs}(\ell, \varepsilon)$ to be the number of times label ℓ occurs disjointly in ε, as defined on page 12. We require for every effect ε created during type inference (including constraint resolution), and for all ℓ, that $\mathit{occurs}(\ell, \varepsilon) \leq 1$. We enforce the constraint $\mathit{effect}(\tau) = \emptyset$ by extracting the effect ε from the function type τ and ensuring that $\mathit{occurs}(\ell, \varepsilon) = 0$ for all ℓ. Finally, we ensure that locations are consistently correlated with locks. We compute $S(C, \rho)$ (from Definition 1) for all locations ρ and check that it has size ≤ 1. This computation is easy now that we have the constraints in solved form; we simply walk through all the correlation constraints generated by the flow rules to count how many different lock labels appear correlated with each location ρ.

Note that the definition of consistent correlation in $\lambda_\triangleright$ is slightly stronger than the definition from $\lambda^{cp}_\triangleright$. In particular, consider the program shown in Figure 21. In $\lambda_\triangleright$, this program will not type check, because $C \vdash l' \triangleright x$ but also $C \vdash l \triangleright x$, because of [Corr-Match] and the self-loop on x because it is global at the definition of f. However, this program will type check in $\lambda^{cp}_\triangleright$, because in [Let] in Figure 16 the constraint system C' containing $l \triangleright x$ is abstracted and instantiated in [Inst], hence the correlation with l never appears in the outermost constraint system.

Now we prove that derivations in $\lambda_\triangleright$ reduce to derivations in $\lambda^{cp}_\triangleright$. Our approach closely follows Rehof et al [43], and we omit details where they are the same.

Definition 25 *Every application of [Inst]*

$$[Inst]\frac{C \vdash \tau \preceq^i_+ \tau' \qquad C \vdash \vec{l} \preceq^i_\pm \vec{l}}{C; \Gamma, f : (\forall.\tau, \vec{l}) \vdash_{CFL} f^i : \tau'; \emptyset}$$

defines an *instantiation context* $\langle C, \vec{l}, \tau, \phi \rangle$, where ϕ is the substitution given by instantiation i.

Definition 26 (Closure) *Let C be a set of $\lambda_\triangleright$ constraints. Then we define $C^* = \{\rho \leq \rho' \mid C \vdash \rho \leq \rho'\} \cup \{\rho \triangleright \ell \mid C \vdash \rho \triangleright \ell\}$, i.e., C^* is the closure of C with respect to the rules in Figure 20.*

Note that we omit effects from the above definition; those are handled by the following definition:

Definition 27 (Effect Closure) *Let C be a set of $\lambda_\triangleright$ constraints. Then we define ε^* to be the solution of ε as computed by the rules in Figure **??** with respect to C:*

$$\begin{array}{rcll} \emptyset^* & = & \emptyset & \\ \{\ell\}^* & = & \{\ell\} & \\ \chi^* & = & \bigcup_{\varepsilon \leq \chi} \varepsilon^* & \\ (\varepsilon_0 \uplus \varepsilon_1)^* & = & \varepsilon_0^* \uplus \varepsilon_1^* & \text{if } \varepsilon_0^* \cap \varepsilon_1^* = \emptyset \\ (\varepsilon_0 \cup \varepsilon_1)^* & = & \varepsilon_0^* \cup \varepsilon_1^* & \end{array}$$

Thus effects are just sets of locks, the same as in $\lambda^{cp}_\triangleright$.

Lemma 28 *If $C \vdash \varepsilon \leq \varepsilon'$, then $\varepsilon^* \subseteq \varepsilon'^*$.*

Lemma 29 *If $C \vdash \varepsilon \leq_{\vec{l}} \varepsilon'$, then $\varepsilon^* \cap \{\ell \mid escapes(\ell, \vec{l})\} \subseteq \varepsilon'^*$.*

Lemma 30 *If $C \vdash \varepsilon \preceq^i \varepsilon'$, then there is a substitution ϕ_i such that $\phi_i(\varepsilon) \subseteq \varepsilon'$.*

Proof: These three lemmas can be proven by observing that the rules in Figure **??** compute a valid solution to the effect constraints. □

Lemma 31 *Let $\langle C, \vec{l}, \tau, \phi \rangle$ be an instantiation context (i.e., an occurrence on [Inst] or [Fix]). Then $C^* \vdash \hat{\phi}(C^*)$, where*

$$\hat{\phi}(\rho) = \begin{cases} \phi(\rho) & \rho \in fl(\tau) - \vec{l} \\ \rho & \rho \in \vec{l} \\ \bigsqcup \hat{\phi}(\left\{\rho' \in (fl(\tau) \cup \vec{l}) \mid C^* \vdash \rho' \leq \rho\right\}) & otherwise \end{cases}$$

and

$$\hat{\phi}(\ell) = \begin{cases} \phi(\ell) & \ell \in \mathit{fl}(\tau) - \vec{l} \\ \ell & \ell \in \vec{l} \\ \emptyset & \mathit{otherwise} \end{cases}$$

Here $\emptyset$ is a special lock indicating no correlation, i.e., constraints of the form $\rho \triangleright \emptyset$ place no constraint on ρ, and $C \vdash \rho \triangleright \emptyset$ for any C, ρ.

Proof: The standard proof [13, 41] of this lemma holds. We show some of the cases for correlation constraints. Suppose $\hat{\phi}(C^*) \vdash \rho' \triangleright \ell'$. Then let ρ, ℓ be such that $C^* \vdash \rho \triangleright \ell$. We need to show that $C^* \vdash \rho' \triangleright \ell'$. There are a total of nine cases, depending on ρ and ℓ.

1. Suppose $\rho \in \vec{l}$. Then $\rho' = \hat{\phi}(\rho) = \rho$.
   (a) Suppose $\ell \in \vec{l}$. Then $\ell' = \hat{\phi}(\ell) = \ell$. Thus by assumption $C^* \vdash \rho' \triangleright \ell'$.
   (b) Suppose $\ell \in \mathit{fl}(\tau) - \vec{l}$. Then $\ell' = \hat{\phi}(\ell) = \phi(\ell)$, and $C \vdash \ell \preceq^i \ell'$. Then since $\rho \in \vec{l}$, by [Inst] we have $C \vdash \rho \preceq^i_{\pm} \rho$ and $\rho = \rho'$. Then by [Corr-Match] we have $C^* \vdash \rho' \triangleright \ell'$.
   (c) Otherwise, $\hat{\phi}(\ell) = \emptyset$, so there is nothing to show.
2. Suppose $\rho \in \mathit{fl}(\tau) - \vec{l}$. Then $\rho' = \phi(\rho)$ where $C \vdash \rho \preceq^i_p \rho'$ for some p.
   (a) Suppose $\ell \in \vec{l}$. Then $\ell' = \hat{\phi}(\ell) = \ell$ and by [Inst] $C \vdash \ell \preceq^i \ell'$. But then by [Corr-Match] we have $C^* \vdash \rho' \triangleright \ell'$.
   (b) Suppose $\ell \in \mathit{fl}(\tau) - \vec{l}$. Then $\ell' = \hat{\phi}(\ell) = \phi(\ell)$, and $C \vdash \ell \preceq^i \ell'$. Then by [Corr-Match] we have $C^* \vdash \rho' \triangleright \ell'$.
   (c) Otherwise, $\hat{\phi}(\ell) = \emptyset$, so there is nothing to show.
3. The last cases follow by the reasoning similar to above plus the standard reasoning about intermediate locations [41].

□

Definition 32 *For a $\lambda_{\triangleright}$ derivation $\mathcal{D}$, let the ith occurrence of [Down] be*

$$[\mathit{Down}]\frac{C; \Gamma_i \vdash e : \tau_i; \varepsilon_i \qquad \vec{l_i} = \mathit{fl}(\Gamma) \cup \mathit{fl}(\tau) \qquad C \vdash \varepsilon_i \leq_{\vec{l_i}} \chi_i}{C; \Gamma_i \vdash e : \tau_i; \chi_i}$$

Let

$$\begin{array}{rcl} \vec{\ell_i} & = & \varepsilon_i^* - \chi_i^* \\ \vec{\alpha_i} & = & \{\alpha \mid \neg(C^* \vdash \mathit{escapes}(\alpha, \mathit{fl}(\Gamma_i) \cup \mathit{fl}(\tau_i)))\} \end{array}$$

Here $\vec{\alpha_i}$ are all the non-escaping locks and locations from [Down]. Notice that by definition of $\leq_{\vec{l_i}}$ we have $\ell_i \subseteq \alpha_i$. Then define $C_i = \nu\vec{\alpha}[C'; \varepsilon']$ to be an alpha-renaming of $\nu\vec{\alpha_i}[C^*|_{\vec{\alpha_i}}; \vec{\ell_i}]$ such that $\vec{\alpha}$ is chosen to be distinct from all free and bound variables in C^* and any other renaming for an occurrence of [Down]. Here $C^*|_{\vec{\alpha_i}}$ are the constraints in C^* that only contain variables in $\vec{\alpha_i}$. Notice that by construction of escapes, it must be the case that in C^*, there are no constraints between a variable in $\vec{\alpha_i}$ and a variable not in $\vec{\alpha_i}$.

Finally, define

$$C^{**} = C^* \cup \bigcup_i C_i$$

Lemma 33 *Let $\langle C, \vec{l}, \tau, \phi\rangle$ be an instantiation context. Then $C^{**} \vdash \hat{\phi}(C^{**})$.*

Proof: By Lemma 31 we have $C^* \vdash \hat{\phi}(C^*)$, and all other constraint systems in C^{**} contain no free variables. □

Definition 34 *Define $(\forall.\tau, \vec{l})^* = \forall\vec{\alpha}[C^{**}].\tau$ where $\vec{\alpha} = (fl(\tau) \cup fl(C^{**})) - \vec{l}$, i.e., we generalize all variables in τ and C^{**} that we can. Define $(\Gamma, x : \sigma)^*$ to be $\Gamma^*, x : \sigma^*$ and $\cdot^* = \cdot$, where $\cdot$ is the empty environment.*

Lemma 35 *If $C \vdash \rho \leq \rho'$ then $C^* \vdash_{cp} \rho \leq \rho'$.*

Lemma 36 *If $C \vdash \rho \rhd \ell$ then $C^* \vdash_{cp} \rho \rhd \ell$.*

Lemma 37 *If $C \vdash \tau \leq \tau'$ then $C^* \vdash_{cp} \tau \leq \tau'$.*

Proof: The proofs of all three statements are trivial. The proof of the last statement uses Lemma 28 to show that the effect constraints from [Sub-Fun] in Figure 7 can be translated to $\subseteq$ conditions for [Sub-Fun] in Figure 17. □

Lemma 38 *Given a normal $C; \Gamma \vdash e : \tau; \varepsilon$, we have $\varepsilon^* \vdash_{ok} C^{**}$*

Proof: We show that the rules in Figure 19 apply. First, we can ignore [Con-Lock] and the last hypothesis of [Con-Union], because constraints of the form $\{L \leq^1 \ell\}$ never appear in C^{**}. Also, by [Con-Other] there is nothing to show for individual constraints.

To show that the disjoint unions in [Con-Encap] and [Con-Union], and the free label restrictions in [Con-Union] hold, observe that in Definition 32 we alpha-renamed all the bindings to be distinct from all other bindings, and thus these hold by construction.

For [Con-Encap], we need to show that in encapsulated constraint systems we bind all ρ's that are correlated with bound ℓ's, but that holds again by construction in Definition 32. And we need to show that $\varepsilon' \subseteq \vec{\alpha}$, but that holds again by construction in Definition 32.

Thus in essence, the only thing to show is consistent correlation according to [Con-Union]. Since all of the bindings are alpha renamed, we need to show consistent correlation of $strip^*(C^{**})$, i.e., that

$$\text{for all } \rho.\ |S(strip^*(C^{**}), \rho)| \leq 1$$

But by assumption $|S(C^*, \rho)| \leq 1$ for all ρ. Therefore for any i we have $|S(C^*|_{\vec{\alpha_i}})| \leq 1$ also. And since all variables in C_i are bound, there will be no overlapping ρ when we apply $strip^*$ to C^{**} from different C_i, and hence the union is consistently correlated. □

Lemma 39 (Reduction) *If $\mathcal{D}$ is a normal derivation of $C; \Gamma \vdash e : \tau; \varepsilon$, then $C^{**}; \Gamma^* \vdash e : \tau; \varepsilon^*$.*

Proof: By induction on the structure of e. The cases for the monomorphic rules follow by induction and Lemmas 35, 36, and 37.

Case [Let]. We have

$$[\text{Let}]\frac{C;\Gamma \vdash v_1 : \tau_1;\emptyset \quad \vec{l} = \mathit{fl}(\Gamma) \quad C;\Gamma, f : (\forall.\tau_1, \vec{l}) \vdash e_2 : \tau_2;\varepsilon}{C;\Gamma \vdash \mathtt{let}\ f = v_1\ \mathtt{in}\ e_2 : \tau_2;\varepsilon}$$

By induction we have $C^{**};\Gamma^* \vdash_{cp} v_1 : \tau_1;\emptyset$ and $C^{**};\Gamma^*, f : \forall\vec{\alpha}[C^{**}].\tau_1 \vdash_{cp} e_2 : \tau_2;\varepsilon^*$, where by construction $\vec{\alpha} = (\mathit{fl}(\tau_1) \cup \mathit{fl}(C^{**})) - \vec{l}$. But then we can apply [Let] from $\lambda^{cp}_{\triangleright}$ to yield

$$[\text{Let}]\frac{\begin{array}{c}C^{**};\Gamma^* \vdash_{cp} v_1 : \tau_1;\emptyset \quad C^{**};\Gamma^*, f : \forall\vec{\alpha}[C^{**}].\tau_1 \vdash_{cp} e_2 : \tau_2;\varepsilon^* \\ \vec{\alpha} \subseteq (\mathit{fl}(\tau_1) \cup \mathit{fl}(C^{**})) \setminus \mathit{fl}(\Gamma^*)\end{array}}{C^{**};\Gamma^* \vdash_{cp} \mathtt{let}\ f = v_1\ \mathtt{in}\ e_2 : \tau_2;\varepsilon^*}$$

Case [Inst]. We have

$$[\text{Inst}]\frac{I \vdash \tau \preceq^i_+ \tau' : \phi \quad I \vdash \vec{l} \preceq^i_\pm \vec{l}}{I;C;\Gamma, f : (\forall.\tau, \vec{l}) \vdash_{CFL} f^i : \tau';\emptyset}$$

We want to show

$$[\text{Inst}]\frac{C^{**} \vdash \hat{\phi}(C^{**}) \quad \mathit{dom}(\hat{\phi}) = \vec{\alpha}}{C^{**};\Gamma^*, f : \forall\vec{\alpha}[C^{**}].\tau \vdash_{cp} f^i : \hat{\phi}(\tau);\emptyset}$$

We apply Lemma 33 to show that $C^{**} \vdash \hat{\phi}(C^{**})$. And $\hat{\phi}(\tau) = \phi(\tau)$, by definition, so the type of f^i is what we expect.

Case [Fix]. We have

$$[\text{Fix}]\frac{\begin{array}{c}C;\Gamma, f : (\forall.\tau, \vec{l}) \vdash v : \tau';\emptyset \quad \vec{l} = \mathit{fl}(\Gamma) \quad C \vdash \tau' \leq \tau \quad C \vdash \tau \preceq^i_+ \tau'' \\ C \vdash \vec{l} \preceq^i_\pm \vec{l} \qquad\qquad C \vdash \mathit{effect}(\tau) = \emptyset\end{array}}{C;\Gamma \vdash \mathtt{fix}\, f.v : \tau'';\emptyset}$$

By induction, we have $C^{**};\Gamma^*, f : \forall\vec{\alpha}[C^{**}].\tau \vdash_{cp} v : \tau';\emptyset$, where by construction $\vec{\alpha} = (\mathit{fl}(\tau) \cup \mathit{fl}(C^{**})) - \vec{l}$. Since $C \vdash \tau' \leq \tau$, by [Sub] and Lemma 37 we have $C^{**};\Gamma^*, f : \forall\vec{\alpha}[C^{**}].\tau \vdash_{cp} v : \tau;\emptyset$. Finally, by Lemma 33 we have $C^{**} \vdash \hat{\phi}(C^{**})$. Putting this together, we get

$$[\text{Fix}]\frac{\begin{array}{c}C^{**};\Gamma^*, f : f : \forall\vec{\alpha}[C^{**}].\tau \vdash_{cp} v : \tau;\emptyset \\ \vec{\alpha} \subseteq (\mathit{fl}(\tau) \cup \mathit{fl}(C^{**})) \setminus \mathit{fl}(\Gamma) \quad C \vdash \phi(C') \quad \mathit{dom}(\hat{\phi}) = \vec{\alpha}\end{array}}{C;\Gamma \vdash_{cp} \mathtt{fix}\, f.v : \hat{\phi}(\tau);\emptyset}$$

And $\hat{\phi}(\tau) = \phi(\tau)$, by definition.

Case [Down]. Let this be the ith occurrence of [Down]. Our derivation looks like the following:

$$[\text{Down}]\frac{C;\Gamma_i \vdash e : \tau_i;\varepsilon_i \quad \vec{l} = \mathit{fl}(\Gamma_i) \cup \mathit{fl}(\tau_i) \quad C \vdash \varepsilon_i \leq_{\vec{l_i}} \chi_i}{C;\Gamma_i \vdash e : \tau_i;\chi_i}$$

By induction, we have

$$C^{**}; \Gamma_i^* \vdash_{cp} e : \tau_i; \varepsilon_i^*$$

Let $\vec{\ell_i}$, $\vec{\alpha_i}$, and $\nu\vec{\alpha_i}[C^*|_{\vec{\alpha_i}}; \vec{\ell_i}]$ be as in Definition 32. Let ϕ be the alpha-renaming such that $\phi(\vec{\alpha_i}) = \vec{\alpha}$, where $\nu\vec{\alpha}[C'; \varepsilon']$ is the constraint in C^{**}.

First, by definition $\varepsilon_i^* = \vec{\ell_i} \uplus \chi_i^*$. Also notice that since $\vec{\ell_i} \subseteq \vec{\alpha_i}$ by construction, we have

$$\varepsilon' = \phi(\vec{\ell_i}) \subseteq \phi(\vec{\alpha_i}) = \vec{\alpha} \tag{1}$$

Also we claim that $\phi(\Gamma_i) = \Gamma_i$ and $\phi(\tau_i) = \tau_i$, since any locks or locations in Γ_i or τ_i are not in $\vec{\alpha_i}$, by definition of *escapes*. Additionally, $\phi(\chi_i^*) = \chi_i^*$, since any lock in χ_i^* escapes and hence is not in $\vec{\alpha_i}$. Then applying ϕ to the induction hypothesis, we get

$$\phi(C^{**}); \Gamma_i^* \vdash_{cp} e : \tau_i; \chi_i^* \uplus \phi(\vec{\ell_i})$$

or

$$\phi(C^*) \cup \bigcup_i C_i; \Gamma_i \vdash_{cp} e : \tau_i; \chi_i^* \uplus \varepsilon'$$

since all variables in the C_i are bound. And by definition of *escapes*, there are no constraints between variables in $\vec{\alpha}_i$ and variables not in $\vec{\alpha}_i$. Therefore we have

$$C^*|_{\neg\vec{\alpha_i}} \cup \phi(C^*|_{\vec{\alpha_i}}) \cup \bigcup_i C_i; \Gamma_i \vdash_{cp} e : \tau_i; \chi_i^* \uplus \varepsilon'$$

Then by Lemma 19 and the definition of C' we have

$$C^* \cup (\bigcup_i C_i) \cup C'; \Gamma_i \vdash_{cp} e : \tau_i; \chi_i^* \uplus \varepsilon'$$

Next let $\vec{l'} = \mathit{fl}(\mathit{strip}^*(C^{**})) \cup \chi_i^*$, and let $\alpha^{\vec{l'}}$ and $\phi_\alpha^{\vec{l'}}$ be alpha-conversions according to Definition 8. Also construct $\phi_\alpha^{\vec{l'}}$ such that $\phi_\alpha^{\vec{l'}}(\vec{\alpha}) \cap (\mathit{fl}(\Gamma_i) \cup \mathit{fl}(\tau_i)) = \emptyset$. Applying $\phi_\alpha^{\vec{l'}}$ to our alpha-renamed inductive hypothesis yields

$$C^* \cup (\bigcup_i C_i) \cup \phi_\alpha^{\vec{l'}}(C'); \Gamma_i \vdash_{cp} e : \tau_i; \chi_i^* \uplus \phi_\alpha^{\vec{l'}}(\varepsilon')$$

since again $\phi_\alpha^{\vec{l'}}$ only renames elements in $\phi(\vec{\alpha})$, which do not appear in Γ_i, τ_i, or χ_i^* by choice of the alpha renaming ϕ in Definition 32. By applying appropriate alpha conversions to the bound constraint systems in C', we get

$$\phi_\alpha^{\vec{l'}}(C') = \mathit{strip}^*(\alpha^{\vec{l'}}(\nu\vec{\alpha}[C'; \varepsilon']))$$

(Note that C' contains no nested ν constraints, by construction, and hence $\mathit{strip}^*(C') = C'$.) Thus we have

$$C^{**} \cup \mathit{strip}(\alpha^{\vec{l'}}(\nu\vec{\alpha}[C'; \varepsilon'])); \Gamma_i \vdash_{cp} e : \tau_i; \chi_i^* \uplus \phi_\alpha^{\vec{l'}}(\varepsilon') \tag{2}$$

Then putting (2) together with (1), the construction of $\vec{l'}$, and the construction of $\phi_\alpha^{\vec{l'}}$, we can apply [Down] from $\lambda_{\triangleright}^{cp}$ to yield:

$$\text{[Down]}\ \frac{\begin{array}{c} C^{**} \cup \mathit{strip}(\alpha^{\vec{l'}}(\nu\vec{\alpha}[C';\varepsilon'])); \Gamma_i \vdash_{cp} e : \tau_i; \chi^* \uplus \phi_\alpha^{\vec{l'}}(\varepsilon') \\ \phi_\alpha^{\vec{l'}}(\vec{\alpha}) \cap (\mathit{fl}(\Gamma_i) \cup \mathit{fl}(\tau_i)) = \emptyset \\ \varepsilon' \subseteq \vec{\alpha} \qquad\qquad \vec{l'} \supseteq \mathit{fl}(C^{**}) \cup \chi_i^* \end{array}}{C^{**}; \Gamma_i \vdash_{cp} e : \tau_i; \chi_i^*}$$

□