ABSTRACT

Title of dissertation: Sound, precise and efficient
static race detection
for multi-threaded programs

Polyvios Pratikakis
Doctor of Philosophy, 2008

Dissertation directed by: Professor Michael Hicks
Professor Jeffrey S. Foster
Department of Computer Science

Multi-threaded programming is increasingly relevant due to the growing prevalence of multi-core processors. Unfortunately, the non-determinism in parallel processing makes it easy to make mistakes but difficult to detect them, so that writing concurrent programs is considered very difficult. A data race, which happens when two threads access the same memory location without synchronization is a common concurrency error, with potentially disastrous consequences.

This dissertation presents LOCKSMITH, a tool for automatically finding data races in multi-threaded C programs by analyzing their source code. LOCKSMITH uses a collection of static analysis techniques to reason about program properties, including a novel effect system to compute memory locations that are shared between threads, a system for inferring "guarded-by" correlations between locks and memory locations, and a novel analysis of data structures using existential types. We present the main analyses in detail and give formal proofs to support their soundness. We discuss the implementation of each analysis in LOCKSMITH, present the problems that arose when extending it to the full C programming language, and discuss some alternative solutions. We provide extensive measurements for the precision and performance of each analysis and compare alternative techniques to find the best combination.

SOUND, PRECISE AND EFFICIENT STATIC RACE DETECTION FOR MULTI-THREADED PROGRAMS

by

Polyvios Pratikakis

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2008

Advisory Committee:
Professor Michael Hicks, Co-chair/Advisor
Professor Jeffrey S. Foster, Co-chair/Co-advisor
Professor Alan Sussman
Professor Dana Nau
Professor Edgar Lopez-Escobar

to Despoina

ACKNOWLEDGMENTS

I owe my gratitude to all the people who have made this dissertation possible.

First and foremost I'd like to thank my advisors, Professors Michael Hicks and Jeffrey Foster, for giving me an invaluable opportunity to work on challenging and extremely interesting projects over the past six years. They have always dedicated their time and effort to provide me with help, advice and guidance without which this dissertation would not be possible. It has been a pleasure and an honor to work with and learn from such extraordinary individuals.

My colleagues at the University of Maryland have enriched my graduate life in many ways and deserve a special mention. Iulian Neamtiu, Nikhil Swami, Mike Furr, David Greenfieldboyce, Saurabh Srivastava and everyone else from my research group provided helpful insight, valuable suggestions and interesting and enlightening questions on numerous occasions.

My internship at Microsoft Research during the summer of 2006 was inspirational and enriching. For that I thank my mentor Chris Hawblitzel, Galen Hunt, Juan Chen, Manuel Fähndrich, and everyone else in the Singularity group for taking time to answer my questions, giving me invaluable feedback and making my overall experience fun and educational.

I owe my deepest thanks to my family—my mother and father who have always stood by me through my career, and my brothers and sister for their support during this enriching and challenging journey.

My house-mates have been a crucial factor in my finishing smoothly. I'd like to express my gratitude to Manolis Hourdakis and Thanos Chryssis for their friendship, camaraderie and support.

I'd like to thank my lifelong friends Nikos, Yannis, Yannis, Chronis, Jason, Vaggelis, Ioanna, Dimitris, Despoina and Dimitris, for their friendship and support despite long distance, in this difficult journey. It is impossible to remember all, and I apologize to those I've inadvertently left out.

Contents

List of Figures

Chapter 1

Introduction

Since the advent of transistors and digital computers, solid-state physics, materials science and computer architecture have advanced continuously, creating numerous iterations of ever smaller and faster processors. In what later became known as *Moore's Law*, Gordon Moore predicted in 1965 that the number of components in integrated circuits would double every year, and ten years later corrected this prediction to every two years. The semiconductor industry has so far kept up with that prediction, reducing the size and increasing the number of components per chip. The reduction in transistor size furthermore allows lower response times and thus higher clock frequencies, making computers faster "for free," even when using the same architecture.

Recently, however, the industry has begun reaching the physical limits of the materials used to make field-effect transistors. This has caused clock frequencies to stop following the exponential doubling pattern of the past, bounded by the physical limits for transistors made of silicon and operating in normal temperatures. Moreover, the time for doubling the number of components per chip had increased from two to three years already in 2006, with predictions of an even longer doubling cycle in the future [55]. To answer the market's ever-increasing demand for processing power, processor manufacturers are turning towards multicore or chip-multiprocessor architectures, that is, more than one processor core per silicon chip. Dual-core processors are already common among desktop users, and hardware manufacturers have announced prototype chips with as many as 80 or 96 cores [80, 115].

This shift in processor architecture cannot but affect software. To take advantage of such computers with multiple processors functioning in parallel, software must be able to perform several tasks at once. It seems inevitable that developing parallel software will become the norm even for the average programmer.

Currently, the dominant model for writing concurrent programs is by using threads, that is, processes that share memory and are executed concurrently. This abstraction makes writing concurrent programs seem very similar to writing sequential programs, aiming to help the programmer reason about each thread as if it were an independent process. Interactions between threads can be both explicit, using basic synchronization constructs such as locks or semaphores, and implicit, through accesses of many threads to the same memory.

Unfortunately, because of the implicit interactions through memory, threads are not

actually independent. That makes writing multi-threaded software difficult and error-prone [92]. In particular, the implicit interactions between threads via shared memory break the assumption that the programmer can reason about each thread in the same way they do for a sequential program. Sequential programs are deterministic and self-consistent, meaning that knowing the instructions and the initial state, one can predict the state at each point of the program's execution. On the contrary, the state of execution in the presence of many parallel threads depends not only on the program instructions, but also on the relative order in which they are executed, making execution non-deterministic.

It is the programmer's responsibility in a multi-threaded program to use explicit synchronization to remove non-determinism, so that either the program is deterministic, or whatever non-determinism is left does not cause the program to misbehave. Hence, programmers writing multi-threaded programs strive to balance *determinism* and *parallelism*. Programmers typically ensure determinism by using synchronization to restrict concurrent access to shared memory. Synchronization, on the other hand, reduces parallelism by restricting the number of possible orderings of shared accesses. Balancing this tension manually can be quite difficult, particularly since traditional uses of blocking synchronization are not modular and composable [68]. Thus, the programmer must reason about the entire program's behavior, the myriad of possible thread interactions, and may need to consider unintuitive memory models [103], making concurrent programming difficult.

This difficulty in writing parallel programs makes programming errors easier to make. Based on the balance between determinism and parallelism in a multi-threaded program, synchronization errors fall in two categories. Erring on the side of excessive synchronization might cause loss of parallelism and performance, or deadlock in the worst case. On the other hand, erring on the side of insufficient synchronization can cause unwanted non-determinism, unpredictable behavior, corrupted memory and crashes.

Data races are synchronization errors that fall in the latter category, and form a particularly important problem in multi-threaded software. A data race occurs when one thread accesses a memory location at the same time another thread writes to it [89]. Data races are infamous for causing failures of critical software systems, with disastrous consequences like loss of life [93] and failure of critical infrastructure [122]. Even in the cases when races do not cause program failures but only benign non-determinism, race-freedom is an important property in its own right, because race-free programs are easier to understand, analyze and transform [6, 136]. For example, race freedom is necessary for reasoning about code that uses locks to achieve atomicity [47, 54], and to improve the precision of dataflow analysis [22].

The inherent nondeterminism in concurrent software means that traditional testing techniques are unreliable for detecting defects and unable to guarantee race freedom, as multiple runs of the same program might produce different results, depending on the relative order of execution among the various threads, which in turn may depend on hardware factors in parallel architectures. To guarantee race freedom for a program, we need to reason about all its possible executions. This problem lends itself nicely to static analysis, i.e., analysis that reasons about program behavior by abstractly modelling all possible executions without running the program.

The summarization of all possible executions done by static analysis might however

be imprecise and include executions that can never happen. This in turn might cause a static analysis to warn about false errors in correct programs. More precise analyses perform less summarization and abstraction of the possible program executions, often at the cost of efficiency. It is undecidable to eliminate false positives altogether without sacrificing soundness, so we aim to measure this tradeoff between precision and efficiency and balance it, achieving a practical technique that can be used to find races while not producing many false positives.

1.1 Thesis

This work aims to develop tools and techniques to either automatically detect all data races in multi-threaded programs, or verify their absence. In short, this dissertation shows that

> *Data races can be automatically detected using static analysis. The static analysis can be sound, giving guarantees of race freedom; it can be precise, producing a small number of false warnings; and it can be efficient.*

In support of this thesis, we developed LOCKSMITH, a tool for automatically finding data races in C programs. This dissertation describes the set of static analyses used in LOCKSMITH. For each analysis, we describe the problem it needs to solve and how that fits into the overall LOCKSMITH architecture. We then formalize our solution and reason about its correctness. Finally, we discuss its implementation and any extensions that might be required to make it practical, and we measure its performance and precision, as well as the effect of its precision on the rest of LOCKSMITH.

1.2 Overview

We give a short overview of the main contributions of this work, following the structure of the rest of the dissertation.

1.2.1 Correlation inference

LOCKSMITH aims to soundly detect data races, and works by enforcing one of the most common techniques for race prevention: for every shared memory location ρ, there must be some lock ℓ that is held whenever ρ is accessed. When this property holds, we say that ρ is *guarded by* ℓ, or ρ and ℓ are *consistently correlated*. Motivated by the need to compute the guarded-by correlations between locations and locks in LOCKSMITH, we developed a context-sensitive analysis for inferring correlations in general. Chapter 3 first formalizes the correlation inference system for a small extension to the lambda calculus and presents its soundness property (proved in Appendix A). Chapter 6 discusses how we extended this technique for C programs in LOCKSMITH.

1.2.2 Existential label flow

Some programs store locks in data structures, making it difficult to reason about their state and their correlations statically. To allow LOCKSMITH to model such use of data structures precisely, we developed a generic system for analyzing data structures using existential quantification. In Chapter 4, we first describe our solution as a label flow analysis with support for existential context sensitivity and describe its proof of soundness (presented in full in Appendix B). We discuss how we applied this general system to add existential context sensitivity to the correlation analysis used in LOCKSMITH in Chapter 6.

1.2.3 Contextual effects

When inferring "guarded-by" correlations in LOCKSMITH, we only need to consider memory locations that are shared between two or more threads, as there can be no concurrent accesses of thread-local memory. To infer the shared memory locations in the program, we intersect at each thread creation point the the set of locations accessed by the child thread (its *standard effect*) with the set of locations accessed by the parent after that child thread is created (the *future effect* of the parent). In collaboration with Neamtiu et al [113] we generalize standard type and effect systems with *contextual effects*, which capture the effect of the computation before (*prior effect*), during (*standard effect*) and after (*future effect*) evaluating a program expression. Chapter 5 formalizes the contextual effect system and describes the interesting and unusual techniques used in its soundness proof (presented in full in Appendix C). We describe how we apply this generic solution back to our problem in LOCKSMITH to compute locations shared between threads in Chapter 6.

1.2.4 LOCKSMITH implementation

To tackle some of the issues created by the unsoundness and complexity of the C language in general and our benchmarks in particular, we employ several additional optimizations in building LOCKSMITH. Chapter 6 discusses in detail the engineering aspects of scaling the basic algorithms for race detection to the full C language, and also presents any additional optimizations we used. In many cases we tried several alternative algorithms with varying performance and precision. We perform a systematic exploration of the tradeoffs between precision and efficiency in the analysis algorithms used in LOCKSMITH both in terms of the algorithm itself, and in terms of its effects on LOCKSMITH as a whole. We performed measurements on a range of benchmarks, including C applications that use POSIX threads and Linux kernel device drives. Across more than 200,000 lines of code, we found many data races, including ones that cause potential crashes.

Put together, our results illuminate some of the key engineering challenges in building LOCKSMITH in particular, and constraint-based program analyses in general. We discovered interesting—and sometimes unexpected—conclusions about the configuration of analyses that lead to the best precision with the best performance, and we believe that our findings will prove valuable for other static analysis designers. We found that con-

text sensitivity is very important for the precision of LOCKSMITH and greatly reduces false warnings, albeit at significant cost on the running time. Also, we found the sharing analysis to be very important for precision, as erroneously inferring a memory location as thread-shared always results in a false warning. Moreover, we found that field-sensitivity for the aliasing analysis is an important factor for the overall precision of LOCKSMITH, and although it slows down the aliasing analysis, the extra precision results in overall speedup by causing less work for subsequent phases.

Chapter 2

Label flow analysis

LOCKSMITH uses static analysis to find data races. Static analysis reasons about all possible run-time behaviors of a program, by examining its code. To model the memory used at run time, LOCKSMITH uses an *abstract memory location* for any variable or expression in the program that represents memory, as is standard in static analysis. Each such abstract location possibly represents many run-time locations; for example a local variable in a function is newly allocated in memory every time the function is called at run time. When an abstract location only corresponds to a single run-time memory location it is *linear*. Conversely, two abstract locations might correspond to the same run-time memory location; for example two pointer variables that point to two abstract locations might point to the same memory location at run time. Thus, an important part of static analysis concerns reasoning about the possible aliasing between abstract memory locations or values.

This chapter presents the theoretical foundations of the static analyses used in LOCKSMITH. First, we present an example to motivate label flow analysis in general, and we explain how to generate and solve label flow constraints to answer points-to queries, using types. We motivate context-sensitive label flow, present it using constrained polymorphic types, and show how to encode it as a problem of context-free language reachability (CFLR), which can be solved more efficiently.

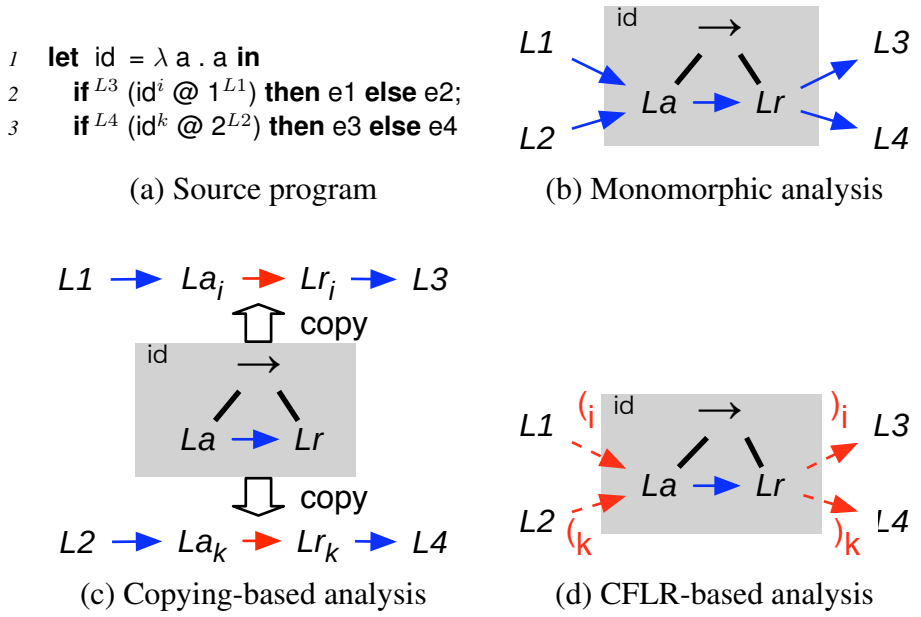
Readers familiar with type based label flow analysis can safely skip this chapter. For a very detailed presentation of type-based label flow analysis with copying context-sensitivity, we refer the reader to the dissertation by Mossin [109], and for its encoding as CFLR to Fähndrich et al [38, 132].

2.1 A simple language

Figure 2.1 presents a simple functional language. This language consists of the standard lambda calculus constructs (variables x, functions $\lambda x : t.e$, and function application $e\ e$) extended in several ways. To model conditional control flow, we add integers n and the conditional form $\mathsf{if0}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$, which evaluates to e_1 if e_0 evaluates to 0, and to e_2 otherwise. To model structured data (i.e., C `structs`) we introduce pairs (e, e) along with projection $e.j$. The latter form returns the jth element of the pair ($j \in 1, 2$).

$$
\begin{array}{rcl}
e & ::= & v \mid x \mid e_1\ e_2 \mid \mathsf{if0}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid (e_1, e_2) \mid e.j \\
 & \mid & \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 \mid \mathsf{fix}^i\ f : t.e \mid f^i \\
v & ::= & n \mid \lambda x : t.e \mid (v_1, v_2) \\
t & ::= & int \mid t \times t \mid t \to t
\end{array}
$$

Figure 2.1: A simple functional language



(a) Source program (b) Monomorphic analysis

(c) Copying-based analysis (d) CFLR-based analysis

Figure 2.2: Universal Types Example

The language also includes name binding construct let $f = e_1$ in e_2 that binds the name f to expression e_1 in the scope of expression e_2, and construct $\mathsf{fix}^i\ f : t.e_1$ that recursively binds the name f to expression e_1 in the scope of expression e_1. Finally, the syntax f^i annotates an occurrence of a let- or fix-bound name f with an index i. We use this index (also in the fix expression) to specifically refer to each occurrence of a name f in a program.

We call the expressions that create a value *constructors*, and the expressions that consume a value *destructors*. For example, expression $\lambda a.a$ constructs a function value, expression 1 constructs a number value, expression $\mathsf{if0}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ consumes a number, and expression $e_1\ e_2$ consumes a function value at run-time.

Source language types t include the integer type int, pair types $t \times t$ and function types $t \to t$. Note that our source language is monomorphically typed, and that in a function $\lambda x : t.e$, the type t of the formal argument x is supplied by the programmer.

$$
\begin{array}{rcl}
e & ::= & v \mid x \mid e_1 @^L e_2 \mid \mathsf{if0}^L\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid (e_1, e_2)^L \mid e.^L j \\
 & \mid & \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 \mid \mathsf{fix}^i\ f : t.e \mid f^i \\
v & ::= & n \mid \lambda^L x : t.e \mid (v_1, v_2)^L \\
\tau & ::= & int^l \mid \tau \times^l \tau \mid \tau \to^l \tau \\
C & ::= & \mid \ell \leq \ell \\
\ell & ::= & L \mid l
\end{array}
$$

$$
\begin{array}{rclll}
\langle\langle int \rangle\rangle & = & int^l & & l \textit{ fresh} \\
\langle\langle t \times t' \rangle\rangle & = & \langle\langle t \rangle\rangle \times^l \langle\langle t' \rangle\rangle & & l \textit{ fresh} \\
\langle\langle t \to t' \rangle\rangle & = & \langle\langle t \rangle\rangle \to^l \langle\langle t' \rangle\rangle & & l \textit{ fresh}
\end{array}
\qquad
\begin{array}{rclll}
\langle\langle int^l \rangle\rangle & = & intl' & & l' \textit{ fresh} \\
\langle\langle \tau \times^l \tau' \rangle\rangle & = & \langle\langle \tau \rangle\rangle \times^{l'} \langle\langle \tau' \rangle\rangle & & l' \textit{ fresh} \\
\langle\langle \tau \to^l \tau' \rangle\rangle & = & \langle\langle \tau \rangle\rangle \to^{l'} \langle\langle \tau' \rangle\rangle & & l' \textit{ fresh}
\end{array}
$$

Figure 2.3: The annotated language

2.2 Monomorphic label flow

Label flow analysis aims to answer queries of the form "does the value of the expression e_1 flow to expression e_2?", by annotating every program expression with a label, and using constraints among labels to model all possible run-time behaviors. For example, the program in Figure 2.2(a) defines the identity function id that simply returns its argument. It then applies it twice on values 1 and 2, and checks the results in two if expressions. In this program, to correctly infer that the values 1 and 2 reach the first and second if expressions at run time, we annotate the values with *constant labels* $L1$ and $L2$, and the if expressions with labels $L3$ and $L4$, respectively. Then, label flow analysis should show that $L1$ flows to $L3$ and $L2$ flows to $L4$.

To infer the flow of values using labels, we use a type-based, constraint-based analysis, i.e. we annotate all types with *label variables* l, and then use a type system to generate and solve *flow constraints* among constant labels and label variables. We refer to constant labels L and label variables l as *labels* ℓ of the form $\ell \leq \ell'$ (ℓ "flows to" ℓ'). The goal of the analysis is to determine which constructor labels flow to which destructor labels. For example, in the expression $(\lambda^L x.e)@^{L'} e'$, the label L flows to the label L'.

The top of Figure 2.3 presents the language, annotated with labels. Like the language in Figure 2.1, it includes integers, variables, functions, function application (written with @ to provide a position on which to write a label), conditionals, pairs, and projection, which extracts a component from a pair, and also binding constructs let and fix, which introduce simple or recursive definitions respectively. We annotate each constructor and destructor expression with a constant label L. We also extend the source types t to *labeled types* τ annotated with label variables l. During type inference, our type rules generate *constraints* C of the form $\ell \leq \ell'$ whenever a label ℓ (either constant L or variable l) *flows to* a label ℓ'. We discuss constraints further below. The bottom half of Fig. 2.3 defines a function $\langle\langle \cdot \rangle\rangle$ that annotates either a standard type t or a labeled type τ with fresh labels at each position.

Fig. 2.4 gives our type inference rules which are as in the standard λ-calculus except for the addition of labels and subtyping. These rules prove judgments of the form $C; \Gamma \vdash e : \tau$, meaning in type environment Γ (a mapping from variable names to labeled types),

$$[\textsc{Var}]\frac{}{C;\Gamma,x:\tau\vdash x:\tau} \qquad [\textsc{Int}]\frac{C\vdash L\leq l}{C;\Gamma\vdash n^L:int^l}$$

$$[\textsc{Lam}]\frac{\begin{array}{c}\tau=\langle\langle t\rangle\rangle\\ C;\Gamma,x:\tau\vdash e:\tau'\\ C\vdash L\leq l\end{array}}{C;\Gamma\vdash\lambda^L x:t.e:\tau\to^l\tau'} \qquad [\textsc{App}]\frac{\begin{array}{c}C;\Gamma\vdash e_1:\tau\to^l\tau'\\ C;\Gamma\vdash e_2:\tau\\ C\vdash l\leq L\end{array}}{C;\Gamma\vdash e_1@^L e_2:\tau'}$$

$$[\textsc{Pair}]\frac{\begin{array}{c}C;\Gamma\vdash e_1:\tau_1 \quad C;\Gamma\vdash e_2:\tau_2\\ C\vdash L\leq l\end{array}}{C;\Gamma\vdash(e_1,e_2)^L:\tau_1\times^{l'}\tau_2} \qquad [\textsc{Proj}]\frac{\begin{array}{c}C;\Gamma\vdash e:\tau_1\times^l\tau_2\\ j\in 1,2 \quad C\vdash l\leq L\end{array}}{C;\Gamma\vdash e.^L j:\tau_j}$$

$$[\textsc{Cond}]\frac{\begin{array}{c}C;\Gamma\vdash e_0:int^l \quad C\vdash l\leq L\\ C;\Gamma\vdash e_1:\tau \quad C;\Gamma\vdash e_2:\tau\end{array}}{C;\Gamma\vdash \mathsf{if0}^L\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2:\tau}$$

$$[\textsc{Let}]\frac{\begin{array}{c}C;\Gamma\vdash e_1:\tau_1\\ C;\Gamma,f:\tau_1\vdash e_2:\tau_2\end{array}}{C;\Gamma\vdash\mathsf{let}\ f=e_1\ \mathsf{in}\ e_2:\tau_2} \qquad [\textsc{Fix}]\frac{\begin{array}{c}\tau=\langle\langle t\rangle\rangle\\ C;\Gamma,f:\tau\vdash e:\tau\end{array}}{C;\Gamma\vdash\mathsf{fix}^i\ f:t.e:\tau}$$

$$[\textsc{Inst}]\frac{}{C;\Gamma,f:\tau\vdash f:\tau} \qquad [\textsc{Sub}]\frac{C;\Gamma\vdash e:\tau_1 \quad C\vdash\tau_1\leq\tau_2}{C;\Gamma\vdash e:\tau_2}$$

Figure 2.4: The monomorphic type rules

$$[\textsc{Sub-Int}]\frac{C \vdash l \leq l'}{C \vdash int^{l} \leq int^{l'}}$$

$$[\textsc{Sub-Pair}]\frac{C \vdash l \leq l' \quad C \vdash \tau_1 \leq \tau_1' \quad C \vdash \tau_2 \leq \tau_2'}{C \vdash \tau_1 \times^{l} \tau_2 \leq \tau_1' \times^{l'} \tau_2'}$$

$$[\textsc{Sub-Fun}]\frac{C \vdash l \leq l' \quad C \vdash \tau_1' \leq \tau_1 \quad C \vdash \tau_2 \leq \tau_2'}{C \vdash \tau_1 \rightarrow^{l} \tau_2 \leq \tau_1' \rightarrow^{l'} \tau_2'}$$

Figure 2.5: Subtyping

$$C \cup \{\ell_1 \leq \ell_2\} \cup \{\ell_2 \leq \ell_3\} \quad \cup \Rightarrow \quad \{\ell_1 \leq \ell_3\}$$

Figure 2.6: Monomorphic closure rule

expression e has type τ under the constraints C.

The constructor rules ([INT], [LAM] and [PAIR]) require $C \vdash L \leq l$, i.e., that the constructor label L must flow to the corresponding label variable l of the constructed type. The destructor rules ([COND], [APP] and [PROJ]) require the converse, i.e., that the label of the destructed type flows to the label of the destructor. We discuss the subsumption rule [SUB] below. In this system, the rules [LET] and [FIX] are equivalent to combining function definition and application, and [INST] is equivalent to [VAR].

Rule [SUB] in Figure 2.4 uses the subtyping relation shown in Figure 2.5. These rules are standard subtyping rules extended to labeled types. We reduce every subtyping relation to flow constraints among the corresponding labels in the two types. Although the rules are formed as a type checking system, we interpret the judgement $C \vdash \ell \leq \ell$ as *adding* the flow constraint $\ell \leq \ell$ to the set C of all constraints. Rule [SUB-INT] simply generates the constraint $l \leq l'$, rule [SUB-PAIR] similarly generates $l \leq l'$ and also descends in both elements of the pairs, and rule [SUB-FUN] generates constraint $l \leq l'$ and similarly descends the subtyping relation in the argument (contravariantly) and the return type (covariantly).

Finally, we solve the generated constraints C using the simple transitivity closure rule shown in Figure 2.6.

2.3 Copying context sensitivity

Applying these rules to our example, the function id is given the type $int^{La} \rightarrow int^{Lr}$, where La and Lr label the argument and return types, respectively. The body of id returns its argument, which is modeled by the *constraint* $La \leq Lr$. The call id^i yields constraints $L1 \leq La$ and $Lr \leq L3$, and the call id^k yields constraints $L2 \leq La$ and $Lr \leq L4$. Pictorially, constraints form the directed edges in a *flow graph*, as shown in Figure 2.2(b), and flow is determined by graph reachability. Thus the graph accurately shows that $L1$ flows to $L3$ and $L2$ flows to $L4$.

Notice, however, that the graph conflates the two calls to id and therefore suggests possible flows from $L1$ to $L4$ and from $L2$ to $L3$, which is sound but imprecise. The precision of the analysis can be improved by adding *context sensitivity*, to differentiate, that is, between the two calls to the function id, at the contexts i and j.

The standard approach [109] to adding context sensitivity, is to capture the body of function id and inline a fresh copy at each call. This is based on that the labels local to function id represent different values for each invocation of id at run-time. We do that by defining *type schemes*

$$\sigma ::= \tau \mid \forall \vec{l}[C].\tau$$

that are either annotated types τ or universally quantified types (or universal types) written as $\forall \vec{l}[C].\tau$. Universal types contain the set of quantified label variables $\vec{l}$, a constraint set C, and an annotated type τ. Note that constant labels are not quantified. For a variable f which has type $\forall \vec{l}[C].\tau$ in the environment, the set $\vec{l}$ includes all the label variables that are certain to represent different values for every occurrence of f in the program; the constraints C capture the label flow in the expression that f represents (e.g., the function body), and τ is the type of f and might contain labels from $\vec{l}$. In other words, The universal type $\forall \vec{l}[C].\tau$ stands for any type $S(\tau)$ where $S(C)$ is satisfied, for any substitution S. We redefine rules [LET] and [FIX] that introduce universal types in Γ, and rule [INST] that instantiates a universal type. Figure 2.7 shows the new rules.

Rule [LET] types expression e_1 (usually a function) under flow constraints C', yielding type τ_1. It then abstracts this type creating a universally polymorphic type $\forall \vec{l}[C'].\tau_1$. The set of abstract labels $\vec{l}$ is a subset of the labels in C' and τ_1, except all labels that are also in the environment Γ, because labels in Γ correspond to the same values for every instance of f. We then bind f to the new universal type and type e_2 under the new environment, yielding type τ_2, which is also the type of the let expression.

Rule [INST] instantiates the universal type $\forall \vec{l}[C'].\tau$ of f. We *instantiate* the universal type using a *substitution* S_i that maps all labels in $\vec{l}$ to fresh labels. The premise of [INST] adds a fresh copy of the constraints C' in f's type to C, where all labels in $\vec{l}$ have been replaced with fresh labels. Finally, the instance type of f^i is $S_i(\tau)$, i.e., the abstract type where all labels in $\vec{l}$ are replaced by fresh.

Similar to both, [FIX] types the expression e under Γ with the additional assumption that f has universal type $\forall \vec{l}[C'].\tau$. Note that the constraints C' of e are also in the universal type of f. The last premise of [FIX] *instantiates* the constraints C' in C.

Returning to the example of Figure 2.2(a), we apply these rules to generate the flow graph shown in Figure 2.2(c). Function id now has a *polymorphically constrained* universal type $\forall La, Lr[La \leq Lr].int^{La} \rightarrow int^{Lr}$, where we have annotated id's type with the flow constraints needed to type its body. Each time id is used, we *instantiate* its type and constraints, effectively "inlining" a fresh copy of id's body. At the call id^i, we instantiate the constraint with the substitution $[La \mapsto La_i, Lr \mapsto Lr_i]$, and then apply the constraints from the call site, yielding $L1 \leq La_i \leq Lr_i \leq L3$, as shown. Similarly, at the call id^k we instantiate again, this time yielding $L2 \leq La_k \leq Lr_k \leq L4$. Thus we see that $L1$ could flow to $L3$, and $L2$ could flow to $L4$, but we avoid the spurious flows from the monomorphic analysis.

$$\text{[LET]}\ \frac{C';\Gamma \vdash e_1 : \tau_1 \qquad C;\Gamma, f : \forall\vec{l}[C'].\tau_1 \vdash e_2 : \tau_2 \qquad \vec{l} \subseteq (\mathit{fl}(\tau_1) \cup \mathit{fl}(C')) \setminus \mathit{fl}(\Gamma)}{C;\Gamma \vdash \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

$$\text{[FIX]}\ \frac{\tau = \langle\langle t \rangle\rangle \qquad C';\Gamma, f : \forall\vec{l}[C'].\tau \vdash e : \tau \qquad \vec{l} \subseteq (\mathit{fl}(\tau) \cup \mathit{fl}(C')) \setminus \mathit{fl}(\Gamma) \qquad C \vdash S_i(C')}{C;\Gamma \vdash \mathsf{let}\ f^i : t = e\ \mathsf{in}\ i : \tau}$$

$$\text{[INST]}\ \frac{C \vdash S_i(C')}{C;\Gamma, f : \forall\vec{l}[C'].\tau \vdash f^i : S_i(\tau)}$$

Figure 2.7: The copying type rules

2.4 Context sensitivity as CFLR

While this technique is effective, explicit constraint copying can be difficult to implement, because it requires juggling various sets of constraints as they are duplicated and instantiated, and may require complicated constraint simplification techniques [44, 35, 41] for efficiency.

An alternative approach is to encode the problem in terms of a slightly different graph and use CFL reachability to compute flow, as suggested by Rehof et al [132]. This solution adds call and return edges to the graph and labels them with parentheses indexed by the call site. Intuitively, we add an edge annotated with $(i$ for label flow *entering* the function f at instantiation i, and an edge annotated with $)i$ for label flow *exiting* f at instantiation i. Then, label flow is restricted to paths in the flow graph with matching parentheses, and excludes mismatched paths. This in turn achieves the goal of differentiating between different instantiations of f, but without replicating all the constraints of f every time.

To encode context sensitivity as a parenthesis-matching problem, we extend constraints C to include special parenthesis-annotated edges, using *instantiation constraints*:

$$C ::= \ldots \mid \tau \preceq_p^i \tau \mid l \preceq_p^i l$$

An instantiation constraint $L \preceq_p^i L'$ indicates that the *abstract* label variable l is renamed to the *instance* label variable l' at instantiation i. The polarity p encodes the direction of flow, so that $l \preceq_+^i l'$ represents the edge $l \xrightarrow{(i} l'$ and $l \preceq_-^i l'$ represents $l' \xrightarrow{)i} l$. We also redefine universal types in type schemes σ to:

$$\sigma ::= (\forall\vec{l}.\tau, \vec{l'})$$

Here, $\vec{l}$ are the labels in τ that are quantified, and $\vec{l'}$ are the labels that are *not* quantified. We write $\vec{l}$ in the type for clarity, although it is always the case that $\vec{l} = \mathit{fl}(\tau) \setminus \vec{l'}$. Finally, Figure 2.8 shows the new rules for [LET], [FIX] and [INST].

Rules [LET] and [FIX] bind f to a universal type. As before, we cannot quantify over labels in Γ, as they do not represent different values for every instance of f, represented by $\vec{L'} = \mathit{fl}(\Gamma)$ in the type. We also limit the quantified labels to the labels in τ,

$$\text{[Let]}\ \frac{C;\Gamma \vdash e_1 : \tau_1 \quad C;\Gamma, f : (\forall \vec{l}.\tau_1, \vec{l'}) \vdash e_2 : \tau_2 \quad \vec{l} = \mathit{fl}(\tau_1) \setminus \vec{l'} \quad \vec{l'} = \mathit{fl}(\Gamma)}{C;\Gamma \vdash \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

$$\text{[Fix]}\ \frac{\tau = \langle\langle t \rangle\rangle \quad \tau' = \langle\langle t \rangle\rangle \quad C;\Gamma, f : (\forall \vec{l}.\tau, \vec{l'}) \vdash e : \tau \quad \vec{l} = \mathit{fl}(\tau_1) \setminus \vec{l'} \quad \vec{l'} = \mathit{fl}(\Gamma) \quad C \vdash \tau \preceq^i_+ \tau' \quad C \vdash \vec{l'} \preceq^i_\pm \vec{l'}}{C;\Gamma \vdash \mathsf{let}\ f^i : t = e\ \mathsf{in}\ i : \tau'}$$

$$\text{[Inst]}\ \frac{\tau' = \langle\langle \tau \rangle\rangle \quad C \vdash \tau \preceq^i_+ \tau' \quad C \vdash \vec{l'} \preceq^i_\pm \vec{l'}}{C;\Gamma, f : (\forall \vec{l}.\tau, \vec{l'}) \vdash f^i : \tau'}$$

Figure 2.8: The CFLR type rules

$$\text{[Inst-Const]}\ \frac{}{C \vdash L \preceq^i_p L}$$

$$\text{[Inst-Int]}\ \frac{C \vdash l \preceq^i_p l'}{C \vdash int^l \preceq^i_p int^{l'}}$$

$$\text{[Inst-Pair]}\ \frac{C \vdash l \preceq^i_p l' \quad C \vdash \tau_1 \preceq^i_p \tau'_1 \quad C \vdash \tau_2 \preceq^i_p \tau'_2}{C \vdash \tau_1 \times^l \tau_2 \preceq^i_p \tau'_1 \times^{l'} \tau'_2}$$

$$\text{[Inst-Fun]}\ \frac{C \vdash l \preceq^i_p l' \quad C \vdash \tau_1 \preceq^i_{\bar{p}} \tau'_1 \quad C \vdash \tau_2 \preceq^i_p \tau'_2}{C \vdash \tau_1 \to^l \tau_2 \preceq^i_p \tau'_1 \to^{l'} \tau'_2}$$

Figure 2.9: Instantiation relation

regardless of the other labels in the body of the expression bound to f. For every instance of a name f in the program, rules [Fix] and [Inst] instantiate universal type $(\forall \vec{L}.\tau, \vec{L'})$. We instantiate every label in τ to the corresponding label in τ', a freshly annotated type, with the constraint $\tau \preceq^i_+ \tau'$. Figure 2.9 defines the instantiation relation on types. Again, the judgement $C \vdash l \preceq^i_p l'$ adds the constraint $l \preceq^i_p l'$ to the set C. Each instantiation also implicitly defines a label substitution S_i that maps labels in τ to the corresponding labels in τ'. All non-quantifiable labels, i.e., all labels in $\vec{l'}$, are not renamed, which we model by instantiating each label in $\vec{l'}$ to itself with both positive and negative polarity.

To solve the constraints C in this case, we introduce two additional closure rules, shown in Figure 2.10. Rule [Constant] adds a "self-loop" that permits matching flows

$$C \cup \{\ell_1 \preceq^i_- \ell_0\} \cup \{\ell_1 \leq \ell_2\} \cup \{\ell_2 \preceq^i_+ \ell_3\} \quad \cup \Rightarrow \quad \{\ell_0 \leq \ell_3\}$$

Figure 2.10: Additional closure rule for CFLR

to or from any constant label. We generate these edges because constant labels are global names and thus context-insensitive.

We show instantiation edges in Figure 2.2(d) with dashed lines. Edges from id^i are labeled with $(_i$ for inputs and $)_i$ for outputs, and similarly for id^k. To compute flow in this graph, we find paths with no mismatched parentheses. In this case the paths from $L1$ to $L3$ and from $L2$ to $L4$ are matched, while the other paths are mismatched and hence not considered.

Chapter 3

Context-sensitive correlation inference

This chapter formalizes the *correlation inference* system used in LOCKSMITH. 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 (CFLR) [132, 135] and semi-unification [71] 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.

Although here we focus on locking, we believe that the concept of correlation may be of independent interest. For example, a program may correlate a variable containing an integer length with a array having that length [168]; it may correlate an environment structure with the closure that takes it as an argument [105]; or it may correlate a memory location with the *region* in which that location is stored [65, 72].

3.1 Race-freedom as consistent correlation

Consider the C program in Figure 3.1. 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.

```
1   pthread_mutex_t L1 = ..., L2 = ...;
2   int x, y, z;
3
4   void munge(pthread_mutex_t *l, int *p) {
5     pthread_mutex_lock(l);
6     *p = 3;
7     pthread_mutex_unlock(l);
8   }
9   ...
10  munge(&L1, &x);
11  munge(&L2, &y);
12  munge(&L2, &z);
```

Figure 3.1: Locking Example in C

$$
\begin{array}{rcl}
e & ::= & v \mid x \mid e_1\ e_2 \mid \mathsf{if0}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid (e_1, e_2) \mid e.j \\
& \mid & \mathsf{let}\ f = v\ \mathsf{in}\ e_2 \mid \mathsf{fix}^i\ f : t.\ v \mid f^i \\
& \mid & \mathsf{newlock} \mid \mathsf{ref}\ e \mid !^{e_2}\ e_1 \mid e_1 :=^{e_3} e_2 \\
v & ::= & n \mid \lambda x : t.e \mid (v_1, v_2) \\
t & ::= & \mathit{int} \mid t \times t \mid t \to t' \mid \mathit{lock} \mid \mathit{ref}(t)
\end{array}
$$

Figure 3.2: $\lambda_{\triangleright}$ Syntax

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.

3.2 The language $\lambda_{\triangleright}$

Figure 3.2 extends the simple language presented in chapter 2 with a primitive for generating mutual exclusion locks and updateable references. We have highlighted the main differences for clarity. As in chapter 2, we annotate function occurrences f^i with an *instantiation site* i. 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. Note that the addition of updateable references forces the abstracted expression in let and fix expressions to be values, to maintain type safety.

```
1  let L1 = newlock in
2  let L2 = newlock in
3  let x = ref 0 in
4  let y = ref 1 in
5  let z = ref 2 in
6
7  let munge l p =
8    p :=^l 3
9  in
10 munge^1 L1 x;
11 munge^2 L2 y;
12 munge^3 L2 z
```

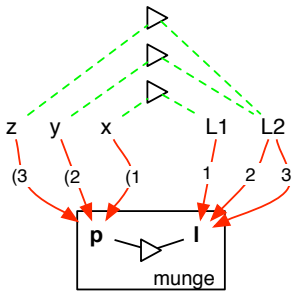


Figure 3.3: Locking example in $\lambda_{\triangleright}$ and its constraint graph

3.2.1 Example

The left side of Figure 3.3 gives the program in Figure 3.1 modeled in $\lambda_{\triangleright}$. The body of munge has been reduced to the expression $\mathsf{p} :=^{\mathsf{l}} 3$, 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 $\mathsf{p} :=^{\mathsf{l}} 3$ 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 $\{\mathtt{L1}, \mathtt{L2}\}$, 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 the correlation between l and p is not being treated *context-sensitively*. Even if we were to use a context-sensitive alias analysis [29], 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.

3.2.2 Correlation constraints

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 3.2.1 (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 3.2.2 (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 3.3 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 dataflow. 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 3.2.2.

3.3 Context-sensitivity as CFLR

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.

3.3.1 Typing

Figure 3.4 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. Again, we highlight the main differences in the type language compared to chapter 2. 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

$$
\begin{array}{llll}
\text{types} & \tau & ::= & \mathit{int} \mid \tau \times \tau \mid \tau \to^{\varepsilon} \tau' \mid \mathit{lock}^{\ell} \mid \mathit{ref}^{\rho}(\tau) \\
\text{labels} & l & ::= & \ell \mid \rho \\
\text{effects} & \varepsilon & ::= & \emptyset \mid \{\ell\} \mid \chi \mid \varepsilon \uplus \varepsilon' \mid \varepsilon \cup \varepsilon' \\
\text{schemes} & \sigma & ::= & (\forall.\tau, \vec{l}) \\
\text{constr. sets} & C & ::= & \emptyset \mid \{c\} \mid C \cup C \\
\text{constraints} & c & ::= & \tau \leq \tau' \qquad \text{(subtyping)} \\
 & & \mid & \ell = \ell' \qquad \text{(lock unification)} \\
 & & \mid & \rho \leq \rho' \qquad \text{(location flow)} \\
 & & \mid & \rho \rhd \ell \qquad \text{(correlation)} \\
 & & \mid & \varepsilon \leq \chi \qquad \text{(effect flow)} \\
 & & \mid & \varepsilon \leq_{\vec{l}} \chi \qquad \text{(effect filtering)} \\
 & & \mid & \mathit{effect}(\tau) = \emptyset \qquad \text{(effect emptiness)} \\
 & & \mid & \tau \preceq^{i}_{p} \tau' \qquad \text{(type instantiation)} \\
 & & \mid & \ell \preceq^{i} \ell' \qquad \text{(lock instantiation)} \\
 & & \mid & \rho \preceq^{i}_{p} \rho' \qquad \text{(location inst.)} \\
 & & \mid & \varepsilon \preceq^{i} \chi \qquad \text{(effect inst.)}
\end{array}
$$

Figure 3.4: Types and Constraints

quantified type, where τ is the base type and $\vec{l}$ is the set of *non*-quantified labels [71, 132]. 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.

Linearity effects Linearity 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 $x \notin \mathit{fv}(e_2)$.

```
1  let l = ref (newlock) in      // l : ref^ρ'((lock^ℓ))
2  let x = ref 0 in              // x : ref^ρ(int)
3    x :=^l 1;                   // ρ ▷ ℓ
4    l := newlock;
5    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.

Typing Rules We now turn to the monomorphic type rules for $\lambda_{\rhd}$, shown in Figure 3.5. 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.

Some other type-based systems for race detection [42, 64] and related systems for modeling dynamic memory allocation [149] 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.

Turning to the remaining rules in Figure 3.5, [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 3.7(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 3.7(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 while holding lock e_2, and generates a correlation constraint between the lock and location label, as does [ASSIGN].

Polymorphism Figure 3.6 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 repre-

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

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

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

$$[\textsc{Sub}]\frac{C;\Gamma\vdash e:\tau_1;\varepsilon \quad C\vdash\tau_1\leq\tau_2}{C;\Gamma\vdash e:\tau_2;\varepsilon} \qquad [\textsc{Newlock}]\frac{\ell\ \mathit{fresh}}{C;\Gamma\vdash\mathsf{newlock}:\mathit{lock}^{\ell};\{\ell\}}$$

$$[\textsc{Cond}]\frac{C;\Gamma\vdash e_0:\mathit{int};\varepsilon_0 \quad C;\Gamma\vdash e_1:\tau;\varepsilon_1 \quad C;\Gamma\vdash e_2:\tau;\varepsilon_2}{C;\Gamma\vdash\mathsf{if0}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2:\tau;\varepsilon_0\uplus(\varepsilon_1\cup\varepsilon_2)}$$

$$[\textsc{Ref}]\frac{C;\Gamma\vdash e:\tau;\varepsilon \quad \rho\ \mathit{fresh}}{C;\Gamma\vdash\mathsf{ref}\ e:\mathit{ref}^{\rho}(\tau);\varepsilon} \qquad [\textsc{Deref}]\frac{C;\Gamma\vdash e_1:\mathit{ref}^{\rho}(\tau);\varepsilon_1 \quad C;\Gamma\vdash e_2:\mathit{lock}^{\ell};\varepsilon_2 \quad C\vdash\rho\rhd\ell}{C;\Gamma\vdash!^{e_2}\ e_1:\tau;\varepsilon_1\uplus\varepsilon_2}$$

$$[\textsc{Assign}]\frac{C;\Gamma\vdash e_1:\mathit{ref}^{\rho}(\tau);\varepsilon_1 \quad C;\Gamma\vdash e_2:\tau;\varepsilon_2 \quad C;\Gamma\vdash e_3:\mathit{lock}^{\ell};\varepsilon_3 \quad C\vdash\rho\rhd\ell}{C;\Gamma\vdash e_1:=^{e_3}\ e_2:\tau;\varepsilon_1\uplus\varepsilon_2\uplus\varepsilon_3}$$

Figure 3.5: $\lambda_{\rhd}$ Monomorphic Rules

sent the direction of subtyping through a constraint, either covariant ($+$) or contravariant ($-$). Instantiation constraints correspond to the edges labeled with parentheses in Figure 3.3. 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 $\rho \to^{)i} \rho'$. A constraint $\rho \preceq^i_- \rho'$ corresponds to an input (i.e., a parameter), and we draw it with a directed edge $\rho' \to^{(i} \rho$. We draw a constraint $\ell \preceq^i \ell'$ as an edge $\ell' \to^i \ell$, 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 3.7(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 3.3, we generated three instantiation

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

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

Figure 3.6: $\lambda_\triangleright$ Polymorphic Rules (plus [Down])

constraints

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

corresponding to the three instantiations and calls of munge. For full details on polarities, see Rehof et al [132].

Hiding Effects [Fix] introduces polymorphic recursion, which is decidable for label flow [109, 132]. However, in our system we instantiate effects and that may cause disjoint unions to 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 to infer correlations for recursive functions that allocate locks. For example, consider the two code snippets in Figure 3.8. 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 [59]. 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

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

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 $\rho \triangleright \ell$ and ρ escapes, then ℓ escapes. This prevents l

$$[\textsc{Sub-Int}]\frac{}{C \vdash int \leq int} \qquad [\textsc{Sub-Lock}]\frac{C \vdash \ell = \ell'}{C \vdash lock^{\ell} \leq lock^{\ell'}}$$

$$[\textsc{Sub-Pair}]\frac{C \vdash \tau_1 \leq \tau_1' \quad C \vdash \tau_2 \leq \tau_2'}{C \vdash \tau_1 \times \tau_2 \leq \tau_1' \times \tau_2'} \qquad [\textsc{Sub-Ref}]\frac{C \vdash \tau \leq \tau' \quad C \vdash \tau' \leq \tau \quad C \vdash \rho \leq \rho'}{C \vdash ref^{\rho}(\tau) \leq ref^{\rho'}(\tau')}$$

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

(a) Subtyping

$$[\textsc{Inst-Int}]\frac{}{C \vdash int \preceq^{i} int} \qquad [\textsc{Inst-Lock}]\frac{C \vdash \ell \preceq^{i} \ell'}{C \vdash lock^{\ell} \preceq^{i}_{p} lock^{\ell'}}$$

$$[\textsc{Inst-Pair}]\frac{C \vdash \tau_1 \preceq^{i}_{p} \tau_1' \quad C \vdash \tau_2 \preceq^{i}_{p} \tau_2'}{C \vdash \tau_1 \times \tau_2 \preceq^{i}_{p} \tau_1' \times \tau_2'} \qquad [\textsc{Inst-Ref}]\frac{C \vdash \rho \preceq^{i}_{p} \rho' \quad C \vdash \tau \preceq^{i}_{\pm} \tau'}{C \vdash ref^{\rho}(\tau) \preceq^{i}_{p} ref^{\rho'}(\tau')}$$

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

(b) Instantiation

Figure 3.7: Subtyping and Instantiation Constraints

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 6.13) that contains the only reference to them.

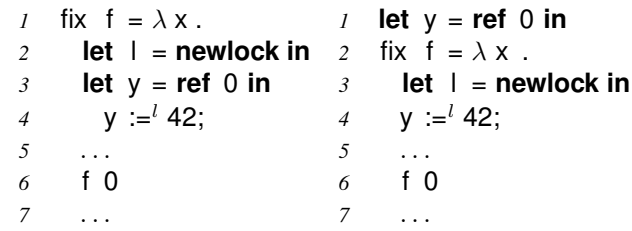


Figure 3.8: Example demonstrating the need for [DOWN]

3.3.2 Constraint resolution

After we have applied the rules in Figures 3.5, 3.6, and 3.7 to a $\lambda_{\triangleright}$ 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 3.9 gives a series of left-to-right rewrite rules that we apply exhaustively to the constraints to compute their solution. Figure 3.9(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 [71]: 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 3.6: if f has polytype $(\forall.ref^{\rho_1}(\tau_1) \rightarrow^{\emptyset} ref^{\rho_2}(\tau_1), \emptyset)$, then instantiating this polytype at site i to the type $ref^{\rho_0}(\tau_1) \rightarrow^{\emptyset} ref^{\rho_3}(\tau_1)$ 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 [132].

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 rule propagates correlations at instantiation sites. Similarly to location propagation, if we have a correlation constraint $\rho \triangleright \ell$ 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 3.3 depicts the following three constraints, among others (recall an edge $l' \rightarrow^{(i}\ l$ in the figure corresponds to a constraint $l \preceq^i_- l'$):

$$\mathsf{l} \preceq^1_- \mathsf{L1} \qquad \mathsf{p} \preceq^1_- \mathsf{x} \qquad \mathsf{p} \triangleright \mathsf{l}$$

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

(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

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

(c) Effect propagation

Figure 3.9: Constraint resolution

Using our resolution rule yields the constraint x $\rhd$ L1, shown in Figure 3.3 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:

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

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 3.9(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 3.9(a) take time $O(n^3)$ [132], as do the rules in Figure 3.9(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 [29].

There exist constraint sets C for which the rules in Figure 3.9(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.

3.4 Soundness

We have proven that a version of our type system $\lambda_{\triangleright}^{cp}$ based on polymorphically constrained types [109] 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 3.4.1 (Valid Evaluation) *We write* $C \vdash e \longrightarrow e'$ *if-f* $e \equiv \mathbb{E}[!^{[L]}\, v^R]$ *or* $e \equiv \mathbb{E}[v'^R :=^{[L]} v]$ *implies* $L \in S_g(C, R)$.

Notice that this still allows a location to be correlated with more than one lock. 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 3.4.2 (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* $C' \vdash C$*; 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' entails 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 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 [38].

Lemma 3.4.3 (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 3.9(a) and (b), ε^* is the set of locks in ε according to the rules in Figure 3.9(c), and Γ^* is a translation from $\lambda_{\triangleright}$ to $\lambda_{\triangleright}^{cp}$ type assumptions.

Full proofs can be found in Appendix A.

Chapter 4

Label flow analysis with existential context sensitivity

In programming languages, existential quantification is useful for describing relationships among members of a structured type. For example, we may have a list in which there *exists* some mutual exclusion lock l in each list element such that l protects the data stored in that element. With this information, a static analysis can reason about the relationship between locks and locations in the list even when the precise identity of the lock and/or location is unknown. To facilitate the construction of such static analyses, this chapter extends the context-sensitive label flow analysis algorithm presented in Chapter 2 with support for existential quantification. Following Chapter 2, we use context-free language reachability (CFLR) to develop an efficient $O(n^3)$ label flow inference algorithm. We prove the algorithm sound by reducing its derivations to those in a system based on polymorphically-constrained types.

4.1 Introduction

Many modern static program analyses are context-sensitive, meaning they can analyze different calls to the same function without conservatively attributing results from one call site to another, as shown in Chapter 2. While this technique is very useful, it often aids little in the analysis of data structures. In particular, a typical alias analysis, even a context-sensitive one, conflates all elements of the same data structure, resulting in a "blob" of indistinguishable pointers [28] that cannot be precisely analyzed.

One way to solve this problem is to use *existential quantification* [107] to express relations among members of each individual data structure element. For example, an element might contain a buffer and the length of that buffer [168]; a pointer to data and the lock that must be held when accessing it [125, 42]; or a closure, consisting of a function and a pointer to its environment [105]. The important idea is that such relations are sound even when the identity of individual data structure elements cannot be discerned.

This chapter extends the context-sensitive label flow analysis algorithm presented in Chapter 2 to support existential quantification. Our goal is to provide a formal foundation for augmenting static analyses with support for existential quantification. The core result

presented in this chapter is a provably sound and efficient type inference system for label flow that supports existential quantification. In summary:

- We present $\lambda_{\exists}^{cp}$, a subtyping-based label-flow system based on the copying system presented in Chapter 2. Our contribution is to show how to support existential quantification using existential types [107], applying the duality of $\forall$ and $\exists$. We prove that the resulting system is sound. (Section 4.3)
- We present $\lambda_{\exists}^{cfl}$, an alternative to $\lambda_{\exists}^{cp}$ that supports efficient inference. Our contribution is to show that existentially-quantified flow can also be expressed as a CFLR problem as in chapter 2, and to prove that $\lambda_{\exists}^{cfl}$ is sound by reducing it to $\lambda_{\exists}^{cp}$. These results are interesting because existential types are *first-class* in our system, as opposed to universal types, which in the style of Hindley-Milner only appear in type environments. To make inference tractable, we require the programmer to indicate where existential types are used, and we restrict the interaction between existentially bound labels and free labels in the program. (Section 4.4)

4.2 Existential context sensitivity

First, we demonstrate how to use existential quantification during static analysis to efficiently model properties of data structures more precisely. We begin by sketching our new technique for supporting first-class existential types, extending the label flow analysis presented in Chapter 2. Sections 4.3 and 4.4 formally develop the label flow systems introduced here.

4.2.1 Existential types and label flow

Consider the example shown in Figure 4.1(a). In this program, functions f and g add an unspecified value to their argument. As before, we wish to determine which integers flow to which $+$ operations. In the third line of this program we create existentially-quantified pairs using pack operations in which f is paired with 1 and g with 2. Using an if, we then conflate these two pairs, binding one of them to p. In the last line we use p by applying its first component to its second component. (We use pattern matching in this example for simplicity, while the language in Section 4.3 uses explicit projection.)

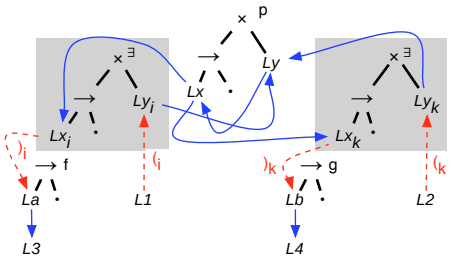
In this example, no matter which pair p is assigned, f is only ever applied to 1, and g is only ever applied to 2. However, an analysis like the one described above would conservatively conflate the types at the two pack sites, generating spurious constraints $L1 \leq L4$ and $L2 \leq L3$. To solve this problem, Section 4.3 presents $\lambda_{\exists}^{cp}$, a system that can model p precisely by giving it a polymorphically constrained existential type

$$\exists Lx, Ly[Ly \leq Lx].(int^{Lx} \to int) \times int^{Ly}$$

indicating that p contains a pair whose second element flows to the argument position of its first element. (The uninteresting labels are omitted for clarity.) At pack^i, this type is instantiated to yield $L1 \leq La$, and since $La \leq L3$ we have $L1 \leq L3$ transitively.

```
1  let f = λ a . a +^L3 ··· in
2  let g = λ b . b +^L4 ··· in
3  let p = if ··· then
4    pack^i (f, 1^L 1)
5  else
6    pack^j (g, 2^L 2)
7  in
8  unpack (p1, p2) = p in
9  p1 @ p2
```

(a) Source program



(b) Flow graph

Figure 4.1: Existential types example

Instantiating at pack^k yields $L2 \leq Lb \leq L4$. Thus we precisely model that 1^{L1} only flows to $+^{L3}$ and 2^{L2} only flows to $+^{L4}$.

To support existential types, we have extrapolated on the duality of universal and existential quantification. Intuitively, we give a universal type to id in Figure 2.2 because id is polymorphic in the label it is called with—whatever it is called with, it returns. Conversely, in Figure 4.1 we give an existential type to p because the *rest of the program* is polymorphic in the pairs—no matter which pair is used, the first element is always applied to the second.

Section 4.4 shows in detail how to perform inference with existential types efficiently using CFLR. Figure 4.1(b) shows the flow graph generated for our example program. When packing the pair $(f, 1^{L_1})$, instead of normal flow edges we generate edges labeled by i-parentheses, and we generate edges labeled by k-parentheses when packing $(g, 2^{L_2})$. Flow for this graph again corresponds to paths with no mismatched parentheses. For example, in this graph there is a matched path from $L2$ to $L4$, indicating that the value 2^{L2} may flow to $+^{L4}$, and there is similarly a path from $L1$ to $L3$. Notice that restricting flow to matched paths again suppresses spurious flows from $L2$ to $L3$ and from $L1$ to $L4$. Thus, the two existential packages can be conflated without losing the flow relationships of their members.

4.2.2 Existential quantification and race detection

Our interest in studying existential label flow arose during the development of LOCKSMITH. LOCKSMITH uses label flow analysis to determine what locations ρ may flow to each assignment or dereference in the program, and we use a combination of label flow analysis and linearity checking to determine which locks ℓ are definitely held at that point. Here ρ and ℓ are just like any other flow labels, and we use different symbols only to emphasize the quantities they label.

Each time a location ρ is accessed with lock ℓ held, LOCKSMITH generates a *correlation constraint* $\rho \triangleright \ell$. After analyzing the whole program, LOCKSMITH ensures that, for each location ρ, there is one lock consistently held for all accesses. Correlation constraints can be easily integrated into flow graphs, and we use a variant of the CFLR closure rules

```
1  struct cache_entry {
2    int refs;
3    pthread_mutex_t refs_mutex;
4     ...
5  };
6
7  void cache_entry_addref(cache_entry *entry) {
8     ...
9    pthread_mutex_lock(&entry->refs_mutex);
10   entry->refs++;
11   pthread_mutex_unlock(&entry->refs_mutex);
12    ...
13 }
```

Figure 4.2: Example code with a per-element lock

to solve for correlations context-sensitively, as shown in chapter 3.

During our experiments we found several examples of code similar to Figure 4.2, which is taken from the *knot* multi-threaded web server [160]. Here cache_entry is a linked list with a per-node lock refs_mutex that guards accesses to the refs field. Without existential quantification, LOCKSMITH conflates all the locks and locations in the data structure. As a result, it does not know exactly which lock is held at the write to entry->refs, and reports that entry->refs may not always be accessed with the same lock held, falsely indicating a potential data race.

With existential quantification, however, LOCKSMITH is able to model this idiom precisely. We add annotations to specify that in type cache_entry, the fields refs and refs_mutex should be given existentially quantified labels. Then we add pack annotations when cache_entry is created and unpack annotations wherever it is used, e.g., within cache_entry_addref. The result is that, in terms of polymorphically constrained types, the entry parameter of cache_entry_addref is given the type

$$\exists \ell, \rho[\rho \rhd \ell].\{\mathsf{refs} : ref^{\rho}()int, \mathsf{refs\_mutex} : lock^{\ell}, \ldots\}$$

and thus LOCKSMITH can verify that the lock refs_mutex always guards the refs field in a given node. The remainder of this chapter focuses exclusively on existential types for label flow in general, not its particular application used in LOCKSMITH.

4.3 $\lambda^{cp}_{\exists}$: Context sensitivity with constraint copying

We begin our formal presentation by studying label flow in the context of a polymorphically-constrained type system $\lambda^{cp}_{\exists}$, which is essentially the copying system presented in chapter 2, extended to include existential types. Note that $\lambda^{cp}_{\exists}$ supports label polymorphism but not polymorphism in the type structure.

$$
\begin{array}{rcl}
e & ::= & v \mid x \mid e_1 @^L e_2 \mid \mathsf{if0}^L\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid (e_1, e_2)^L \mid e.^L j \\
& \mid & \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 \mid \mathsf{fix}^i\ f : t.e \mid f^i \\
& \mid & \mathsf{pack}^{L,i}\ e \mid \mathsf{unpack}^L\ x = e_1\ \mathsf{in}\ e_2 \\
v & ::= & n \mid \lambda^L x : t.e \mid (v_1, v_2)^L \\
\tau & ::= & int^l \mid \tau \times^l \tau \mid \tau \to^l \tau \mid \exists^l \vec{l}[C].\tau \\
C & ::= & \mid \ell \leq \ell \\
\ell & ::= & L \mid l
\end{array}
$$

Figure 4.3: A language for label flow analysis with existentials

4.3.1 A language with existential packages

Figure 4.3 extends the (annotated) source language from Figure 2.3 with constructs for creating and using existential packages. Again, we highlight the differences to improve readability.

We annotate the new constructor pack and destructor unpack expressions with constant labels. As in chapter 2, the goal of our type system is to determine which constructor labels flow to which destructor labels. Expressions include existential packages, which are created with $\mathsf{pack}^{L,i}$ and consumed with unpack. Here L labels the package itself, since existentials are first-class and can be passed around the program just like any other value, and i identifies this pack as an instantiation site. Instantiation sites are ignored in this section, but are used in Section 4.4.

The types and environments used by $\lambda^{cp}_{\exists}$ extend the types in Figure 2.3 to add existential types:

$$\tau \quad ::= \quad \ldots \mid \exists^l \vec{l}[C].\tau$$

The new type $\exists^l \vec{l}[C].\tau$ stands for the type $S(\tau)$ where constraints $S(C)$ are satisfied for *some* substitution S. Note that universal types may only appear in type environments while existential types may appear arbitrarily.

4.3.2 Typing

The expression typing rules are presented in Figures 4.4 and 4.5. Judgments have the form $C; \Gamma \vdash_{cp} e : \tau$, meaning in type environment Γ with flow constraints C, expression e has type τ. As in chapter 2, in these type rules $C \vdash l \leq l'$ means that the constraint $l \leq l'$ is in the transitive closure of the constraints in C, and $C \vdash C'$ means that all constraints in C' are in the transitive closure of C.

Figure 4.4 contains the monomorphic typing rules, which are as in chapter 2, repeated here for clarity.

Figure 4.5 contains the polymorphic typing rules. Universal types are as in chapter 2, introduced by [LET] and [FIX]. In both these rules, the constraints C' used to type e_1 become the bound constraints in the polymorphic type. Whenever a variable f with a universal type is used in the program text, written f^i where i identifies this occurrence of f, it is type checked by [INST]. This rule instantiates the type of f, and the premise

$$[\textsc{Id}]\frac{}{C;\Gamma,x:\tau\vdash_{cp} x:\tau} \qquad [\textsc{Int}]\frac{C\vdash L\leq l}{C;\Gamma\vdash_{cp} n^L : int^l}$$

$$[\textsc{Lam}]\frac{\begin{array}{c}C;\Gamma,x:\tau\vdash_{cp} e:\tau' \\ C\vdash L\leq l\end{array}}{C;\Gamma\vdash_{cp}\lambda^L x.e:\tau\to^l\tau'} \qquad [\textsc{App}]\frac{\begin{array}{c}C;\Gamma\vdash_{cp} e_1:\tau\to^l\tau' \\ C;\Gamma\vdash_{cp} e_2:\tau \qquad C\vdash l\leq L\end{array}}{C;\Gamma\vdash_{cp} e_1@^L e_2:\tau'}$$

$$[\textsc{Pair}]\frac{\begin{array}{c}C;\Gamma\vdash_{cp} e_1:\tau_1 \qquad C;\Gamma\vdash_{cp} e_2:\tau_2 \\ C\vdash L\leq l\end{array}}{C;\Gamma\vdash_{cp}(e_1,e_2)^L:\tau_1\times^l\tau_2} \qquad [\textsc{Proj}]\frac{\begin{array}{c}C;\Gamma\vdash_{cp} e:\tau_1\times^l\tau_2 \\ C\vdash l\leq L \qquad j\in\{1,2\}\end{array}}{C;\Gamma\vdash_{cp} e.^L j:\tau_j}$$

$$[\textsc{Cond}]\frac{\begin{array}{c}C;\Gamma\vdash_{cp} e_0:int^l \qquad C\vdash l\leq L \\ C;\Gamma\vdash_{cp} e_1:\tau \qquad C;\Gamma\vdash_{cp} e_2:\tau\end{array}}{C;\Gamma\vdash_{cp}\mathsf{if0}^L\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2:\tau} \qquad [\textsc{Sub}]\frac{\begin{array}{c}C;\Gamma\vdash_{cp} e:\tau_1 \\ C;\emptyset\vdash\tau_1\leq\tau_2\end{array}}{C;\Gamma\vdash_{cp} e:\tau_2}$$

Figure 4.4: $\lambda^{cp}_{\exists}$ Monomorphic Rules

$C\vdash S(C')$ effectively inlines the constraints of f function into the caller's context.

Existential types are manipulated using pack and unpack. To understand [PACK] and [UNPACK], recall that $\forall$ and $\exists$ are dual notions. Notice that $\forall$ introduction ([LET]) restricts what can be universally quantified, and instantiation occurs at $\forall$ elimination ([INST]). Thus $\exists$ introduction ([PACK]) should perform instantiation, and $\exists$ elimination ([UNPACK]) should restrict what can be existentially quantified.

In [PACK], an expression e with a concrete type $S(\tau)$ is abstracted to an existential type $\exists^l\vec{l}[C'].\tau$. Notice that the substitution maps abstract τ and C' to concrete $S(\tau)$ and $S(C')$—creating an existential corresponds to passing an argument to "the rest of the program," as if that were universally quantified in $\vec{l}$, and the constraints C' are determined by how the existential package is used after it is unpacked. Similarly to [INST], the [PACK] premise $C\vdash S(C')$ inlines the abstract constraints $S(C')$ into the current constraints.

Rule [UNPACK] binds the contents of the type to x in the scope of e_2. This rule places two restrictions on the existential package. First, e_2 must type check with the constraints $C\cup C'$.[1] Thus, any constraints among the existentially bound labels $\vec{l}$ needed to check e_2 must be in C'. Second, the labels $\vec{l}$ must not escape the scope of the unpack (as is standard [107]), which is ensured by the subset constraint.

The [SUB] rule in Figure 4.4 uses the subtyping relation shown in Figure 4.6. These rules are standard structural subtyping rules extended to labeled types. We use a simple approach to decide whether one existential is a subtype of another. Rule [SUB-$\exists$] requires $C_1\vdash C_2$, since an existential type can be used in any position inducing the same or fewer flows between labels. We allow subtyping among existentials of a "similar shape." That

[1]Note that we could have chosen this hypothesis to be $C';\Gamma,x:\tau\vdash_{cp} e_2:\tau'$ and still had a sound system, but this choice simplifies the reduction from $\lambda^{cfl}_{\exists}$ to $\lambda^{cp}_{\exists}$ discussed in Section 4.4.

$$\text{[LET]}\ \frac{C';\Gamma \vdash_{cp} e_1 : \tau_1 \qquad C;\Gamma, f : \forall \vec{l}[C'].\tau_1 \vdash_{cp} e_2 : \tau_2 \qquad \vec{l} \subseteq (fl(\tau_1) \cup fl(C')) \setminus fl(\Gamma)}{C;\Gamma \vdash_{cp} \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

$$\text{[FIX]}\ \frac{C';\Gamma, f : \forall \vec{l}[C'].\tau \vdash_{cp} e : \tau \qquad C \vdash S(C') \qquad \vec{l} \subseteq (fl(\tau) \cup fl(C')) \setminus fl(\Gamma)}{C;\Gamma \vdash_{cp} \mathsf{fix}^{i}\ f.eS(\tau)}$$

$$\text{[INST]}\ \frac{C \vdash S(C')}{C;\Gamma, f : \forall \vec{l}[C'].\tau \vdash_{cp} f^i : S(\tau)}$$

$$\text{[PACK]}\ \frac{C;\Gamma \vdash_{cp} e : S(\tau) \qquad C \vdash S(C') \qquad C \vdash L \leq l}{C;\Gamma \vdash_{cp} \mathsf{pack}^{L,i}\ e\ : \exists^{l} \vec{l}[C'].\tau}$$

$$\text{[UNPACK]}\ \frac{C;\Gamma \vdash_{cp} e_1 : \exists^{l} \vec{l}[C'].\tau \qquad C \cup C';\Gamma, x : \tau \vdash_{cp} e_2 : \tau' \qquad \vec{l} \subseteq (fl(\tau) \cup fl(C')) \setminus (fl(\Gamma) \cup fl(C) \cup fl(\tau')) \qquad C \vdash l \leq L}{C;\Gamma \vdash_{cp} \mathsf{unpack}^{L}\ x = e_1\ \mathsf{in}\ e_2 : \tau'}$$

Figure 4.5: $\lambda^{cp}_{\exists}$ Polymorphic Rules

is, they must have exactly the same (alpha-convertible) bound variables, and there must be no constraints between variables bound in one type and free in the other. We use a set D to track the set of bound variables, updated in [SUB-$\exists$].[2] Rule [SUB-LABEL-2] permits subtyping between identical bound labels ($l \in D$), whereas rule [SUB-LABEL-1] allows subtyping among non-identical labels only if neither is bound.

These restrictions on existentials forbid some clearly erroneous judgments such as $C \vdash \exists l[\emptyset].int^{l} \leq \exists l[\emptyset].int^{l'}$. The two existential types in this example quantify over the same label; however, the subtyping is invalid because it would create a constraint between a bound label and an unbound label. However, these restrictions also forbid some valid existential subtyping, such as $C \vdash (\exists l, l'[].l \leq l' int^{l} \rightarrow int^{l'}) \leq (\exists l, l'[].\emptyset int^{l} \rightarrow int^{l})$, which is permissible because l' is a bound variable with no other lower bounds except l, hence it can be set to l without losing information. However, our typing rules do not allow this. In our experience with LOCKSMITH we have not found this restriction to be an issue, and we leave it as an open question whether it can be relaxed while still maintaining efficient CFLR-based inference.

4.3.3 Soundness

We prove soundness for $\lambda^{cp}_{\exists}$ using subject reduction. Using a standard small-step operational semantics $e \longrightarrow e'$, we define a *flow-preserving evaluation step* as one whose flow is allowed by some constraint set C. Then we prove that if a program is well-typed

[2]The appendix B uses an equivalent version of D that makes the reduction proof easier.

$$[\text{SUB-LABEL-1}]\frac{l, l' \notin D \quad C \vdash l \leq l'}{C; D \vdash l \leq l'} \qquad [\text{SUB-LABEL-2}]\frac{l \in D}{C; D \vdash l \leq l}$$

$$[\text{SUB-PAIR}]\frac{C; D \vdash l \leq l' \quad C; D \vdash \tau_1 \leq \tau_1' \quad C; D \vdash \tau_2 \leq \tau_2'}{C; D \vdash \tau_1 \times^l \tau_2 \leq \tau_1' \times^{l'} \tau_2'}$$

$$[\text{SUB-FUN}]\frac{C; D \vdash l \leq l' \quad C; D \vdash \tau_1' \leq \tau_1 \quad C; D \vdash \tau_2 \leq \tau_2'}{C; D \vdash \tau_1 \to^l \tau_2 \leq \tau_1' \to^{l'} \tau_2'}$$

$$[\text{SUB-INT}]\frac{C; D \vdash l \leq l'}{C; D \vdash int^l \leq int^{l'}} \qquad [\text{SUB-}\exists]\frac{\begin{array}{cc} C_1 \vdash C_2 & D' = D \cup \vec{l} \\ C; D' \vdash \tau_1 \leq \tau_2 & C; D \vdash l_1 \leq l_2 \end{array}}{C; D \vdash \exists^{l_1}\vec{l}[C_1].\tau_1 \leq \exists^{l_2}\vec{l}[C_2].\tau_2}$$

Figure 4.6: $\lambda_\exists^{cp}$ Subtyping

according to C then it always preserves flow.

Definition 4.3.1 (Flow-preserving Evaluation Step) *Suppose $e \longrightarrow e'$ and in this reduction a destructor* (if0, @, $.j$, unpack) *labeled L' consumes a constructor* (n, λ, $(\cdot,\cdot)$, pack, *respectively*) *labeled L. Then we write $C \vdash e \longrightarrow e'$ if $C \vdash L \leq L'$. We also write $C \vdash e \longrightarrow e'$ if no value is consumed during reduction (for* let *or* fix*).*

Theorem 4.3.2 (Soundness) *If $C; \Gamma \vdash_{cp} e : \tau$ and $e \longrightarrow^* e'$, then $C \vdash e \longrightarrow^* e'$.*

Here, $\longrightarrow^*$ denotes the reflexive and transitive closure of the $\longrightarrow$ relation. The proof is by induction on $C; \Gamma \vdash_{cp} e : \tau$ and is presented in Appendix B.

4.4 $\lambda_\exists^{cfl}$: Context sensitivity as CFLR

The $\lambda_\exists^{cp}$ type system is relatively easy to understand and convenient for proving soundness, but experience suggests it is awkward to implement directly as an inference system. This section presents a label flow inference system $\lambda_\exists^{cfl}$ based on CFLR, in the style of Rehof et al [132, 38]. This system uses a single, global set of constraints, which correspond to flow graphs like those shown in Figures 2.2(d) and 4.1. Given a flow graph, we can answer queries "Does any value labeled l_1 flow to a destructor labeled l_2?", written $l_1 \leq l_2$, by using CFLR. We first present type checking rules for $\lambda_\exists^{cfl}$ and then explain how they are used to interpret the flow graph in Figure 4.1. Then we explain how the rules can be interpreted to yield an efficient inference algorithm. Finally, we prove that $\lambda_\exists^{cfl}$ reduces to $\lambda_\exists^{cp}$ and thus is sound.

4.4.1 Typing

Types in $\lambda_{\exists}^{cfl}$ are similar to the CFLR system in chapter 2 with the addition of existential types, and are defined as follows:

$$\tau \quad ::= \quad int^l \mid \tau \rightarrow^l \tau \mid \tau \times^l \tau \mid \exists^l \vec{l}.\tau$$

In contrast to $\lambda_{\exists}^{cp}$, universal types $(\forall \vec{l}.\tau, \vec{l})$ and existential types $\exists^l \vec{l}.\tau$ do not include a constraint set, since we generate a single, global flow graph. Universal types contain a set $\vec{l}$ of labels that are *not* quantified. For clarity universal types also include $\vec{l}$, the set of labels that are quantified, but it is always the case that $\vec{l} = \mathit{fl}(\tau) \setminus \vec{l}$, as in chapter 2. Existential types do not include a set $\vec{l}$, because we assume that the programmer has specified which labels are existentially quantified. We check that the specification is correct when existentials are unpacked (more on this below).

Typing judgments in $\lambda_{\exists}^{cfl}$ have the form $C; \Gamma \vdash e : \tau$, where C describes the edges in the flow graph. The constraint set C has the same form as in $\lambda_{\exists}^{cp}$. It contains subtyping constraints $l \leq l'$ (shown as unlabeled directed edges in Figures 2.2 and 4.1) and it also contains *instantiation constraints* [132] of the form $l \preceq_p^i l'$. Such a constraint indicates that l is renamed to l' at instantiation site i. (Recall that each instantiation site corresponds to a pack or a use of a universally quantified type.) As in chapter 2, the p indicates a *polarity*, which describes the flow of data. When p is $+$ then l flows to l', and so in our examples we draw the constraint $l \preceq_+^i l'$ as an edge $l \xrightarrow{)i} l'$. When p is $-$ the reverse holds, and so we draw the constraint $l \preceq_-^i l'$ as an edge $l' \xrightarrow{(i} l$. Instantiation constraints correspond to substitutions in $\lambda_{\exists}^{cp}$, and they enable context-sensitivity without the need to copy constraint sets.

The monomorphic rules for $\lambda_{\exists}^{cfl}$ are presented in Figure 4.7. With the exception of [SUB], these are identical to the rules in Figure 4.4. Figure 4.8 presents the polymorphic $\lambda_{\exists}^{cfl}$ rules. We define $\mathit{fl}(\tau)$ to be the free labels of a type as usual, except $\mathit{fl}(\forall \vec{l}.\tau, \vec{l}) = (\mathit{fl}(\tau) \setminus \vec{l}) \cup \vec{l}$. Rules [LET] and [FIX] bind f to a universal type as in chapter 2. As is standard we cannot quantify label variables that are free in the environment Γ, which we represent by setting $\vec{l} = \mathit{fl}(\Gamma)$ in type $(\forall \vec{l}.\tau_1, \vec{l})$. The [INST] rule instantiates the type τ of f to τ' using an instantiation constraint $C \vdash \tau \preceq_+^i \tau' : S$. This constraint represents a renaming S, analogous to that in $\lambda_{\exists}^{cp}$'s [INST] rule, such that $S(\tau) = \tau'$. All non-quantifiable labels, i.e., all labels in $\vec{l}$, should not be instantiated, which we model by requiring that any such label instantiate to itself, both positively and negatively.

Rule [PACK] constructs an existential type by abstracting a concrete type τ' to abstract type τ. In $\lambda_{\exists}^{cp}$'s [PACK], there is a substitution such that $\tau' = S(\tau)$, and thus $\lambda_{\exists}^{cfl}$'s [PACK] has a corresponding instantiation constraint $\tau \preceq_-^i \tau'$. The instantiation constraint has negative polarity because although the substitution is from abstract τ to concrete τ', the direction of flow is the reverse, since the packed expression e flows to the packed value. In [PACK] the choice of $\vec{l}$ is not specified. As in other systems for inferring first-class existential and universal types [13, 90, 134, 146], we expect the programmer to choose this set. In contrast to [INST], we do not generate any self-instantiations in [PACK], because we enforce a stronger restriction for escaping variables in [UNPACK].

$$[\text{ID}]\frac{}{C;\Gamma,x:\tau\vdash_{cfl} x:\tau} \qquad [\text{INT}]\frac{C\vdash L\leq l}{C;\Gamma\vdash_{cfl} n^L:int^l}$$

$$[\text{LAM}]\frac{\begin{array}{c}C;\Gamma,x:\tau\vdash_{cfl} e:\tau'\\ C\vdash L\leq l\end{array}}{C;\Gamma\vdash_{cfl}\lambda^L x.e:\tau\to^l\tau'} \qquad [\text{APP}]\frac{\begin{array}{c}C;\Gamma\vdash_{cfl} e_1:\tau\to^l\tau'\\ C;\Gamma\vdash_{cfl} e_2:\tau \quad C\vdash l\leq L\end{array}}{C;\Gamma\vdash_{cfl} e_1@^L e_2:\tau'}$$

$$[\text{PAIR}]\frac{\begin{array}{c}C;\Gamma\vdash_{cfl} e_1:\tau_1 \quad C;\Gamma\vdash_{cfl} e_2:\tau_2\\ C\vdash L\leq l\end{array}}{C;\Gamma\vdash_{cfl}(e_1,e_2)^L:\tau_1\times^l\tau_2} \qquad [\text{PROJ}]\frac{\begin{array}{c}C;\Gamma\vdash_{cfl} e:\tau_1\times^l\tau_2\\ C\vdash l\leq L \quad j\in\{1,2\}\end{array}}{C;\Gamma\vdash_{cfl} e.^L j:\tau_j}$$

$$[\text{COND}]\frac{\begin{array}{c}C;\Gamma\vdash_{cfl} e_0:int^l \quad C\vdash l\leq L\\ C;\Gamma\vdash_{cfl} e_1:\tau \quad C;\Gamma\vdash_{cfl} e_2:\tau\end{array}}{C;\Gamma\vdash_{cfl}\mathsf{if0}^L\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2:\tau} \qquad [\text{SUB}]\frac{\begin{array}{c}C;\Gamma\vdash_{cfl} e:\tau_1\\ C;\emptyset;\emptyset\vdash\tau_1\leq\tau_2\end{array}}{C;\Gamma\vdash_{cfl} e:\tau_2}$$

Figure 4.7: $\lambda^{cfl}_{\exists}$ Monomorphic Rules

Rule [UNPACK] treats the abstract existential type as a concrete type within e_2, and thus any uses of the unpacked value place constraints on its existential type. The last premise of [UNPACK] ensures that abstract labels do not escape, and moreover abstract labels may not constrain any escaping labels in any way. Specifically, we require that there are no flows after solving the constraints (see below) between any labels in $\vec{l}$ and any labels in $\vec{l}$, which is the set of labels that could escape. If this condition is violated, then the existentially quantified labels $\vec{l}$ chosen by the programmer are invalid and the program is rejected. The [UNPACK] rule in $\lambda^{cp}_{\exists}$ does not forbid interaction between free and bound labels, and therefore $\lambda^{cfl}_{\exists}$ is strictly weaker than $\lambda^{cp}_{\exists}$. However, without this restriction we can produce cases where mixing existentials and universals produces flow paths that should be valid but have mismatched parentheses. Section 4.4.4 contains one such example. In practice we believe the restriction is acceptable, as we have not found it to be an issue with LOCKSMITH. We leave it as an open question whether the restriction can be relaxed while still maintaining efficient CFLR-based inference.

Figure 4.9 defines the subtyping relation used in [SUB]. The only interesting difference with $\lambda^{cp}_{\exists}$ arises because of alpha-conversion. In $\lambda^{cp}_{\exists}$ alpha-conversion is implicit, and only trivial constraints are allowed between bound labels (by [SUB-LABEL-2] of Figure 4.6). We cannot use implicit alpha-conversions in $\lambda^{cfl}_{\exists}$, however, because we are producing a single, global set of constraints. Thus instead of the single D used in $\lambda^{cp}_{\exists}$'s[SUB] rule, $\lambda^{cfl}_{\exists}$ uses two Δ_i, which are sequences of ordered vectors of existentially-bound labels, updated in [SUB-$\exists$]. In the rules, the syntax $\Delta\oplus\{l_1,...,l_n\}$ means to append vector $\{l_1,...,l_n\}$ to sequence Δ. Rule [SUB-IND-2] in Figure 4.9, which corresponds to [SUB-LABEL-2] in Figure 4.6, does allow subtyping between bound labels l_j and l'_j—but only if they occur in exactly the same quantification position. Thus these subtyping edges actually correspond to alpha-conversion. We could also allow this in the $\lambda^{cp}_{\exists}$ system, but it

$$\text{[LET]}\ \frac{C;\Gamma \vdash_{cfl} e_1 : \tau_1 \quad C;\Gamma, f : (\forall \vec{l}.\tau_1, \vec{l}) \vdash_{cfl} e_2 : \tau_2 \quad \vec{l} = fl(\tau_1) \setminus \vec{l} \quad \vec{l} = fl(\Gamma)}{C;\Gamma \vdash_{cfl} \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

$$\text{[FIX]}\ \frac{C;\Gamma, f : (\forall \vec{l}.\tau, \vec{l}) \vdash_{cfl} e : \tau \quad \vec{l} = fl(\tau) \setminus fl(\Gamma) \quad \vec{l} = fl(\Gamma) \quad C \vdash \tau \preceq^i_+ \tau' : S \quad C \vdash \vec{l} \preceq^i_+ \vec{l} \quad C \vdash \vec{l} \preceq^i_- \vec{l}}{C;\Gamma \vdash_{cfl} \mathsf{fix}^i\ f.e\tau'}$$

$$\text{[INST]}\ \frac{C \vdash \tau \preceq^i_+ \tau' : S \quad C \vdash \vec{l} \preceq^i_+ \vec{l} \quad C \vdash \vec{l} \preceq^i_- \vec{l}}{C;\Gamma, f : (\forall \vec{l}.\tau, \vec{l}) \vdash_{cfl} f^i : \tau'}$$

$$\text{[PACK]}\ \frac{C;\Gamma \vdash_{cfl} e : \tau' \quad C \vdash \tau \preceq^i_- \tau' : S \quad dom(S) = \vec{l} \quad C \vdash L \leq l}{C;\Gamma \vdash_{cfl} \mathsf{pack}^{L,i}\ e : \exists^l \vec{l}.\tau}$$

$$\text{[UNPACK]}\ \frac{C;\Gamma \vdash_{cfl} e_1 : \exists^l \vec{l}.\tau \quad C;\Gamma, x : \tau \vdash_{cfl} e_2 : \tau' \quad \vec{l} = fl(\Gamma) \cup fl(\exists^l \vec{l}.\tau) \cup fl(\tau') \cup L \quad \vec{l} \subseteq fl(\tau) \setminus \vec{l} \quad C \vdash l \leq L \quad \forall l \in \vec{l}, l' \in \vec{l}.(C \not\vdash l \leq l' \text{ and } C \not\vdash l' \leq l)}{C;\Gamma \vdash_{cfl} \mathsf{unpack}^L\ x = e_1\ \mathsf{in}\ e_2 : \tau'}$$

Figure 4.8: $\lambda^{cfl}_{\exists}$ Polymorphic Rules

adds no expressive power and complicates proving soundness.

Figure 4.10 defines instantiation constraints on types in terms of instantiation constraints on labels. Judgments have the form $C; D \vdash \tau \preceq^i_p \tau' : S_i$, where S_i is the renaming defined by the instantiation and D is the same as in Figure 4.6—we do not need to allow alpha-conversion here, because we can always apply [SUB] if we wish to alpha-rename. Thus [INST-IND-1] permits instantiation of unbound labels, and [INST-IND-2] forbids renaming bound labels. For example, if we have an $\exists$ type nested inside a $\forall$ type, instantiating the $\forall$ type should not rename any of the bound variables of the $\exists$ type. Aside from this the rules in Figure 4.10 are standard.

Given a flow graph described by constraints C, we use the closure rules presented in Figures 2.6 and 2.10 in chapter 2 to compute the closure of the relation $l_1 \leq l_2$, which means label l_1 flows to label l_2.

Our flow relation $\leq$ in the closure of C, corresponds to the $\rightsquigarrow_m$ relation from Rehof et al [132], where m stands for "matched paths." The Rehof et al system also includes so-called *PN paths*, which allow extra parentheses that are not matched by anything, e.g., extra open parentheses at the beginning of the path, or extra closed parentheses at the end. In our system we concern ourselves only with constants, which by [CONSTANT] have all possible self-loops (this rule is not included in the Rehof et al system). These self-loops mean that any flow from one constant to another via a PN path is also captured by a matched path between the constants. Thus for purposes of showing soundness, matched

$$[\textsc{Sub-Ind-1}]\frac{C \vdash l \leq l'}{C; \emptyset; \emptyset \vdash l \leq l'} \qquad [\textsc{Sub-Int}]\frac{C; \Delta_1; \Delta_2 \vdash l \leq l'}{C; \Delta_1; \Delta_2 \vdash int^l \leq int^{l'}}$$

$$[\textsc{Sub-Ind-2}]\frac{C \vdash l_j \leq l'_j}{C; \Delta_1 \oplus \{l_1, \ldots, l_n\}; \Delta_2 \oplus \{l'_1, \ldots, l'_n\} \vdash l_j \leq l'_j}$$

$$[\textsc{Sub-Ind-3}]\frac{C; \Delta_1; \Delta_2 \vdash l \leq l' \quad l \neq l_i \quad l' \neq l'_j \quad \forall i, j \in [1..n]}{C; \Delta_1 \oplus \{l_1, \ldots, l_n\}; \Delta_2 \oplus \{l'_1, \ldots, l'_n\} \vdash l \leq l'}$$

$$[\textsc{Sub-Pair}]\frac{C; \Delta_1; \Delta_2 \vdash l \leq l' \quad C; \Delta_1; \Delta_2 \vdash \tau_1 \leq \tau'_1 \quad C; \Delta_1; \Delta_2 \vdash \tau_2 \leq \tau'_2}{C; \Delta_1; \Delta_2 \vdash \tau_1 \times^l \tau_2 \leq \tau'_1 \times^{l'} \tau'_2}$$

$$[\textsc{Sub-Fun}]\frac{C; \Delta_1; \Delta_2 \vdash l \leq l' \quad C; \Delta_1; \Delta_2 \vdash \tau'_1 \leq \tau_1 \quad C; \Delta_1; \Delta_2 \vdash \tau_2 \leq \tau'_2}{C; \Delta_1; \Delta_2 \vdash \tau_1 \to^l \tau_2 \leq \tau'_1 \to^{l'} \tau'_2}$$

$$[\textsc{Sub-}\exists]\frac{\begin{array}{c}\Delta'_1 = \Delta_1 \oplus \vec{l_1} \quad \Delta'_2 = \Delta_2 \oplus \vec{l_2} \quad S(\vec{l_2}) = \vec{l_1} \\ C; \Delta'_1; \Delta'_2 \vdash \tau_1 \leq \tau_2 \quad C; \Delta_1; \Delta_2 \vdash l_1 \leq l_2\end{array}}{C; \Delta_1; \Delta_2 \vdash \exists^{l_1}\vec{l_1}.\tau_1 \leq \exists^{l_2}\vec{l_2}.\tau_2}$$

Figure 4.9: $\lambda^{cfl}_{\exists}$ Subtyping

paths suffice. We could add PN paths to our system with no difficulty to allow queries on intermediate flows, but have not done so for simplicity.

4.4.2 Example

Consider again the example in Figure 4.1. The expression $\mathsf{pack}^i(f, 1^{L1})$ is given the type

$$\exists Lx_i, Ly_i.(int^{Lx_i} \to int) \times int^{Ly_i}$$

by the [PACK] rule. [PACK] also instantiates the pair's abstract type to its concrete type using the judgment

$$C \vdash (int^{Lx_i} \to int) \times int^{Ly_i} \preceq^i_- (int^{La} \to int) \times int^{L1}$$

Proving this judgment requires appealing in several places to [INST-IND-1], whose premise $C \vdash l \preceq^i_p l'$ requires that C contain constraints $Ly_i \preceq^i_- L1$ and $Lx_i \preceq^i_+ La$, among others. These are shown as dashed, labeled edges in the figure. Notice that the direction of the renaming is opposite the direction of flow: The concrete labels flow to the abstract labels, but the abstract type is instantiated to the concrete type. Hence the instantiation has negative polarity. This instantiated existential type flows via subtyping to the type of p shown at the center of the figure. The directed edges between the type components are induced by subtyping (applying [SUB-$\exists$] at the top level).

$$\text{[INST-IND-1]}\frac{l, l' \notin D \qquad C \vdash l \preceq^i_p l'}{C; D \vdash l \preceq^i_p l' : \emptyset} \qquad \text{[INST-IND-2]}\frac{l \in D}{C; D \vdash l \preceq^i_p l : S}$$

$$\text{[INST-INT]}\frac{C; D \vdash l \preceq^i_p l' : S}{C; D \vdash int^l \preceq^i_p int^{l'} : S}$$

$$\text{[INST-PAIR]}\frac{C; D \vdash l \preceq^i_p l' : S \qquad C; D \vdash \tau_1 \preceq^i_p \tau_1' : S \qquad C; D \vdash \tau_2 \preceq^i_p \tau_2' : S}{C; D \vdash \tau_1 \times^l \tau_2 \preceq^i_p \tau_1' \times^{l'} \tau_2' : S}$$

$$\text{[INST-FUN]}\frac{C; D \vdash l \preceq^i_p l' : S \qquad C; D \vdash \tau_1 \preceq^i_{\bar{p}} \tau_1' : S \qquad C; D \vdash \tau_2 \preceq^i_p \tau_2' : S}{C; D \vdash \tau_1 \to^l \tau_2 \preceq^i_p \tau_1' \to^{l'} \tau_2' : S}$$

$$\text{[INST-}\exists\text{]}\frac{D' = D \cup \vec{l} \qquad C; D' \vdash \tau_1 \preceq^i_p \tau_2 : S \qquad C; D \vdash l_1 \preceq^i_p l_2 : S}{C; D \vdash \exists^{l_1}\vec{l}.\tau_1 \preceq^i_p \exists^{l_2}\vec{l}.\tau_2 : S}$$

Figure 4.10: $\lambda^{cfl}_{\exists}$ Instantiation

The unpack of p is typed by the [UNPACK] rule. Within the body of the unpack, we apply the second part of the pair ($p2$) to the first part ($p1$). Here, $p2$ has type int^{Ly} while $p1$ has type $int^{Lx} \to int$, and thus to apply the [APP] rule, we must first prove (among other things) that $C; \emptyset; \emptyset \vdash int^{Ly} \leq int^{Lx}$. This requires that $Ly \leq Lx$ be in C according to [SUB-IND-1], and is shown as an unlabeled edge in the figure. With this edge we have $C \vdash L1 \leq L3$ and $C \vdash L2 \leq L4$ (but $C \not\vdash L1 \leq L4$). The final premises of [UNPACK] are satisfied because the bound labels Ly and Lx only flow among themselves or to variables bound in existential types, which are not free.

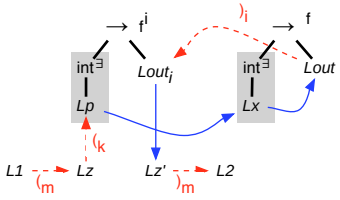
4.4.3 An inference algorithm

$\lambda^{cfl}_{\exists}$ has been presented thus far as a checking system in which the flow graph, described by C, is assumed to be known. To infer this flow graph automatically requires a simple reinterpretation of the rules, as in chapter 2. The algorithm has three stages and runs in time $O(n^3)$, where n is the size of the type-annotated program.

First, we type the program according to the rules in Figures 4.7-4.10. As usual the non-syntactic rule [SUB] can be incorporated into the remaining rules to produce a syntax-directed system [106]. During typing, we interpret a premise $C \vdash l \leq l'$ or $C \vdash \vec{l} \preceq^i_p \vec{l}$ as *generating* a constraint; i.e., we add $l \leq l'$ (or $\vec{l} \preceq^i_p \vec{l}$) to the set of global constraints C. Free occurrences of l in the rules are interpreted as fresh label variables. For example, in [INT] we interpret l as a fresh variable l and add $L \leq l$ to C. When choosing types (e.g., τ in [LAM] or τ' in [INST]) we pick a type τ of the correct shape with fresh label variables in every position. After typing we have a flow graph defined by constraint sets

```
1  let g = λ z .
2    let f = λ x . unpack y = x in y in
3    let p = pack^k z in
4    f^i p
5  in
6  g^m 1^L1 +^L2 ...
```

(a) Source program



(b) Flow graph

Figure 4.11: Example with Mismatched Flow

C.

Next, we compute all flows according to the rules in Figures 2.6 and 2.10. Excluding the final premise of [UNPACK] and the D's in [SUB] and [INST], performing typing and computing all flows takes time $O(n^3)$ [132]. To implement [SUB-IND-i] efficiently, rather than maintain D sets explicitly and repeatedly traverse them, we temporarily mark each variable with a pair (i, j) indicating its position in D and its position in $\vec{l}$ as we traverse an existential type. We can assume without loss of generality that $|\vec{l}| \leq |\mathit{fl}(\tau)|$ in an existential type, so traversing $\vec{l}$ does not increase the complexity. Then we can select among [SUB-IND-1] and [SUB-IND-2] in constant time for each constraint $C; \Delta_1; \Delta_2 \vdash l \leq l'$, so this does not affect the running time, and similarly for [INST-IND-i].

Finally, we check the last reachability condition of [UNPACK] to ensure the programmer chose a valid specification of existential quantification. Given that we have computed all flows, we can easily traverse the labels in $\vec{l}$ and check for paths to $\vec{l}$ and vice-versa. Since each set is of size $O(n)$, this takes $O(n^2)$ time, and since there are $O(n)$ uses of [UNPACK], in total this takes $O(n^3)$ time. Thus the algorithm as a whole is $O(n^3) + O(n^3) = O(n^3)$.

4.4.4 Differences between $\lambda_\exists^{cp}$ and $\lambda_\exists^{cfl}$

As mentioned in Section 4.4, if we weaken $\lambda_\exists^{cfl}$'s [UNPACK] rule to permit existentially bound labels to interact with free labels, then we can construct examples with mismatched flow. Figure 4.11(a) shows one such example. Here the function g takes an argument z, packs it, and then returns the result of calling function f with the package. Function f unpacks the existential and returns its contents. Thus g is the identity function, but with complicated dataflow. On the last line, the function g is applied to 1^{L1}, and the result is added using $+^{L2}$. Thus $L1$ flows to $L2$. Let us assume that at pack^k, the programmer wishes to quantify the type of the packed integer, and then compare $\lambda_\exists^{cp}$ and $\lambda_\exists^{cfl}$ as applied to the program.

The $\lambda_\exists^{cp}$ types rules assign f the type scheme

$$f : \forall \mathit{Lout}[\emptyset].\ \left(\exists \mathit{Lx}[\mathit{Lx} \leq \mathit{Lout}].\mathit{int}^{\mathit{Lx}}\right) \rightarrow \mathit{int}^{\mathit{Lout}}$$

Notice that since f unpacks its argument and returns the contents, there is a constraint between Lx, the label of the packed integer, and Lout, the label on f's result type. The

interesting thing here is that Lx is existentially bound and $Lout$ is not, which is acceptable in $\lambda^{cp}_{\exists}$ (technically, we need an application of [SUB] to achieve this), but not allowed in $\lambda^{cfl}_{\exists}$. At the call to f, we instantiate f's type as

$$f^i : \left(\exists Lx[Lx \leq Lout_i].int^{Lx}\right) \rightarrow int^{Lout_i}$$

Let Lz be the label on g's parameter, and let Lz' be the label on g's return type. Then when we pack z and bind the result to p, we instantiate the abstract Lx to concrete Lz and thus generate the constraint $Lz \leq Lout_i$. Then g returns the result of f^i, and hence we have $Lout_i \leq Lz'$. Putting these together and generalizing g's type, we get

$$g : \forall Lz, Lz', Lout_i[Lz \leq Lout_i, Lout_i \leq Lz'].int^{Lz} \rightarrow int^{Lz'}$$

Finally, we instantiate this type at g^m, and we get $L1 \leq Lz_m \leq Lout_{im} \leq Lz'_m \leq L2$, and thus we have flow from $L1$ to $L2$.

Now consider applying $\lambda^{cfl}_{\exists}$ to the same program. Figure 4.11(b) shows the resulting flow graph. The type of f, shown at the right of the figure, is $(\forall Lout.(\exists Lx.int^{Lx}) \rightarrow int^{Lout}, \emptyset)$ where in the global flow graph there is a constraint $Lx \leq Lout$. As before, this is a constraint between an existentially bound and free variable, which is forbidden by the strong non-escaping condition in $\lambda^{cfl}_{\exists}$'s [UNPACK] rule. However, assume for the moment that we ignore this condition. Then the type of f^i, shown in the left of the figure, is $\left(\exists Lp.int^{Lp}\right) \rightarrow int^{Lout_i}$ where we have an instantiation constraint $Lout \preceq^i_+ Lout_i$, drawn as a dashed edge labeled $)_i$ in the figure. (Note that we have also applied an extra step of subtyping to make the figure easier to read and drawn an edge $Lp \leq Lx$, although we could also set $Lp = Lx$.) Since the result of calling f^i is returned, we have $Lout_i \leq Lz'$, where again Lz' is the label on the return type of g. Moreover, at pack^k, we instantiate the abstract type of p to its concrete type, resulting in the constraint $Lp \preceq^k_- Lz$, where Lz is the label on g's parameter. Finally, at the instantiation of g we generate constraints $Lz \preceq^m_- L1$ and $Lz' \preceq^m_+ L2$.

Notice that there is no path from $L1$ to $L2$, because $(_k$ does not match $)_i$. The problem is that instantiation i must not rename Lp, and instantiation k must not rename $Lout_i$. In CFLR, we prevent instantiations from renaming labels by adding "self-loops," as in [INST] in Figure 4.8. In this case, we should have $Lp \preceq^i_\pm Lp$ and $Lout_i \preceq^k_\pm Lout_i$. We expended significant effort trying to discover a system that would add exactly these self-loops, but we were unable to find a solution that would work in all cases. For example, adding a self-loop on $Lout_i$ seems particularly problematic, since $Lout_i$ is created only after f^i is instantiated, and not at the pack or the unpack points. Moreover, because we have $(_m$ and $)_m$ at the beginning and end of the mismatched path, the self-loops on $L1$ and $L2$ do not help. Thus in [UNPACK] in Figure 4.8, we require existentially-quantified labels to not have any flow with escaping labels to forbid this example.

4.4.5 Soundness

We have proven that programs that check under $\lambda^{cfl}_{\exists}$ are reducible to $\lambda^{cp}_{\exists}$. The first step is to define a translation function $\Psi_{C,I}$ that takes $\lambda^{cfl}_{\exists}$ types and transforms them to $\lambda^{cp}_{\exists}$ types. For monomorphic types $\Psi_{C,I}$ is simply the identity. To translate a polymorphic

$\lambda_{\exists}^{cfl}$ type $(\forall\vec{l}.\tau, \vec{l})$ or $\exists^{l}\vec{l}.\tau$ into a $\lambda_{\exists}^{cp}$ type $\forall\vec{l}[C'].\tau$ or $\exists^{l}\vec{l}[C'].\tau$, respectively, $\Psi_{C,I}$ needs to produce a bound constraint set C'. Rehof et al [132, 38] were able to choose $C' = \{l_1 \leq l_2 \mid C \vdash l_1 \leq l_2\}$, i.e., the closure of C. However, the addition of first class existentials causes this approach to fail, because, for example, instantiating a $\forall$ type containing a type $\exists^{l}\vec{l}[C].\tau$ could rename some variables in C (since C contains all variables used in the program) and thereby violate the inductive hypothesis. Thus we introduce a projection function ψ_S, where we define

$$\psi_S(l) = \begin{cases} l & l \in S \cup L \\ \bigsqcup\{l' \in S \cup L \mid C \vdash l' \leq l\} & \textit{otherwise} \end{cases}$$

where $\sqcup$ represents the union of two labels. Then for a universal type, $\Psi_{C,I}$ sets $C' = \psi_{(\vec{l}\cup\vec{l})}(C)$, and for an existential type $\Psi_{C,I}$ sets $C' = \psi_{\vec{l}}(C)$. We extend $\Psi_{C,I}$ to type environments in the natural way and define $C_S = \psi_S(C)$. Now we can show:

Theorem 4.4.1 (Reduction from $\lambda_{\exists}^{cfl}$ to $\lambda_{\exists}^{cp}$) *Let $\mathcal{D}$ be a normal $\lambda_{\exists}^{cfl}$ derivation*

$$\mathcal{D} :: C; \Gamma \vdash_{cfl} e : \tau$$

Then

$$C_{fl(\Gamma)\cup fl(\tau)}; \Psi_{C,I}(\Gamma) \vdash_{cp} e : \Psi_{C,I}(\tau)$$

Proof: The proof is by induction on the derivation $\mathcal{D}$. There are two key parts of the proof. The first is a lemma that shows that the bound constraint sets chosen by $\Psi_{C,I}$ for universal and existential types are closed under substitutions at instantiation sites, so that when we translate an occurrence of [INST] or [PACK] from $\lambda_{\exists}^{cfl}$ to $\lambda_{\exists}^{cp}$ we can prove the hypothesis $C \vdash S(C')$. The other key part occurs in translating an occurrence of [UNPACK] from $\lambda_{\exists}^{cfl}$ to $\lambda_{\exists}^{cp}$. In this case, by induction on the typing derivation for e_2 we have $C_{fl(\Gamma)\cup fl(\tau)\cup fl(\tau')}; \Psi_{C,I}(\Gamma), x : \Psi_{C,I}(\tau) \vdash_{cp} e_2 : \Psi_{C,I}(\tau')$. By the last hypothesis of [UNPACK] in $\lambda_{\exists}^{cfl}$, we know that there are no constraints between the quantified labels $\vec{l}$ and any other labels. Thus we can partition the constraints on the left-hand side of the above typing judgment into two disjoint sets: $C_{(fl(\Gamma)\cup fl(\tau)\cup fl(\tau'))\setminus\vec{l}}$ and $C_{\vec{l}}$. The former are the constraints needed to type check e_1 in $\lambda_{\exists}^{cp}$, and the latter are those bound in the existential type of e_1 by $\Psi_{C,I}$. These two constraint sets form the sets C and C', respectively, needed for the [UNPACK] rule of $\lambda_{\exists}^{cp}$. A full, detailed proof can be found in Appendix B. $\square$ By combining Theorems 4.3.2 and 4.4.1, we then have soundness for the flow relation $\leq$ computed by $\lambda_{\exists}^{cfl}$. Notice that we have shown reduction but not equivalence. Rehof et al [132, 38] also only show reduction, but conjecture equivalence of their systems. In our case, equivalence clearly does not hold, because of the extra non-escaping condition on [UNPACK] in $\lambda_{\exists}^{cfl}$. We leave it as an open question whether this condition can be relaxed to yield provably equivalent systems.

Chapter 5

Mechanizing the soundness of contextual effects

5.1 Introduction

Type and effect systems are used to reason about a program's computational effects [100, 116, 154]. Such systems have various applications in program analysis, e.g., to compute the set of memory accesses, I/O calls, function calls or new threads that occur in any given part of the program. Generally speaking, a type and effect system proves judgments of the form $\varepsilon; \Gamma \vdash e : \tau$ where ε is the effect of expression e. We proposed generalizing such systems to track what we call *contextual effects*, which capture the effects of the context in which an expression occurs [113]. In our contextual effect system, judgments have the form $\Phi; \Gamma \vdash e : \tau$, where Φ is a tuple $[\alpha; \varepsilon; \omega]$ containing ε, the standard effect of e, and α and ω, the effects of the program evaluation prior to and after computing e, respectively.

This chapter presents the formalization and proof of soundness of contextual effects, which we have mechanized using the Coq proof assistant [26]. Intuitively, for all subexpressions e of a given program e_p, a contextual effect $[\alpha; \varepsilon; \omega]$ is sound for e if (1) α contains the actual, run-time effect of evaluating e_p prior to evaluating e, (2) ε contains the run-time effect of evaluating e itself, and (3) ω contains the run-time effect of evaluating the remainder of e_p after e's evaluation has finished. (Discussed in Section 5.2.)

There are two main challenges with formalizing this intuition to prove that our contextual effect system is sound. First, we must find a way to define what constitute the *actual* prior and future effects of e when it is evaluated as part of e_p. Interestingly, these effects cannot be computed compositionally (i.e., by considering the subterms of e), as they depend on the relative position of the evaluation of e within the evaluation of e_p, and not on the evaluation of e itself. Moreover, the future effect of e models the evaluation after e has reduced to a value. In a small-step semantics, specifying the future effect by finding the end of e's computation would be possible but awkward. Thus we opt for a big-step operational semantics, in which we can easily and naturally define the prior, standard, and future effect of every subterm in a derivation. (Section 5.3)

The second challenge, and the main novelty of our proof, is specifying how to

match up the contextual effect Φ of e, as determined by the *original* typing derivation of $\Phi_p; \Gamma \vdash e_p : \tau_p$, with the run-time effects of e recorded in the evaluation derivation. The difficulty here is that, due to substitution, e may appear many times and in different forms in the evaluation of e_p. In particular, a value containing e may be passed to a function $\lambda x.e'$ such that x occurs several times in e', and thus after evaluating the application, e will be duplicated. Moreover, variables within e itself could be substituted away by other reductions. Thus we cannot just syntactically match a subterm e of the original program e_p with its corresponding terms in the evaluation derivation.

To solve this problem, we define a *typed operational semantics* in which each subderivation is annotated with two typing derivations, one for the term under consideration and one for its final value. Subterms in the original program e_p are annotated with subderivations of the original typing derivation $\Phi_p; \Gamma \vdash e_p : \tau_p$. As subterms are duplicated and have substitutions applied to them, our semantics propagates the typing derivations in the natural way to the new terms. In particular, if Φ is the contextual effect of subterm e of e_p, then all of the terms derived from e will also have contextual effect Φ in the typed operational semantics. Given this semantics, we can now express soundness formally, namely that in every subderivation of the typed evaluation of a program, the contextual effect Φ in its typing contains the run-time prior, standard, and future effects of its computation. (Section 5.4)

We mechanized our proof using the Coq proof assistant, starting from the framework developed by Aydemir et al [8]. We found the mechanization process worthwhile, because our proof structure, while conceptually clear, required getting a lot of details right. Most notably, typing derivations are nested inside of evaluation derivations in the typed operational semantics, and thus the proofs of each case of the lemmas are somewhat messy. Using a proof assistant made it easy to ensure we had not missed anything. We found that, modulo some typos, our paper proof was correct, though the mechanization required that we precisely define the meaning of "subderivation." (Section 5.5)

This approach to proving soundness of contextual effects could be useful for other systems, in particular ones in which properties of subderivations depend on their position within the larger derivation in which they appear.

5.2 Background: contextual effects

This section reviews our type and effect system, and largely follows our previous presentation [113]. Readers familiar with the system can safely skip this section.

5.2.1 Language

Figure 5.1 presents our source language, a simple calculus with expressions that consist of values v (integers, functions or pointers), variables and call-by-value function application. Our language also includes updateable references, created with $\mathsf{ref}^L\ e$, along with dereference and assignment. We annotate each syntactic occurrence of ref with a label L, which serves as the abstract name for the locations allocated at that program point. Evaluating $\mathsf{ref}^L\ e$ creates a pointer r_L, where r is a fresh name in the heap and

Expressions	e	::=	$v \mid x \mid e\ e \mid \mathsf{ref}^L\ e \mid !\,e \mid e := e$
Values	v	::=	$n \mid \lambda x.e \mid r_L$
Effects	$\alpha, \varepsilon, \omega$	::=	$\emptyset \mid 1 \mid \{L\} \mid \varepsilon \cup \varepsilon$
Contextual Effects	Φ	::=	$[\alpha; \varepsilon; \omega]$
Types	τ	::=	$int \mid ref^{\varepsilon}(\tau) \mid \tau \rightarrow^{\Phi} \tau$
Environments	Γ	::=	$\cdot \mid (\Gamma, x \mapsto \tau) \mid (\Gamma, r \mapsto \tau)$
Labels	L		

Figure 5.1: Syntax

L is the declared label. Dereferencing or assigning to r_L during evaluation has effect $\{L\}$. Note that pointers r_L do not appear in the syntax of the program, but only during its evaluation. For simplicity we do not model recursive functions directly, but they can be encoded using references. Also, due to space constraints we omit let and if. They are included in the mechanized proof; supporting them is straightforward.

An *effect*, written α, ε, or ω, is a possibly-empty set of labels, and may be 1, the set of all labels. A *contextual effect*, written Φ, is a tuple $[\alpha; \varepsilon; \omega]$. If e' is a subexpression of e, and e' has contextual effect $[\alpha; \varepsilon; \omega]$, then

- The *current effect* ε is the effect of evaluating e' itself.
- The *prior effect* α is the effect of evaluating e until we begin evaluating e'.
- The *future effect* ω is the effect of the remainder of the evaluation of e after e' is fully evaluated.

Thus ε is the effect of e' itself, $\alpha \cup \omega$ is the effect of the context in which e' appears, and therefore $\alpha \cup \varepsilon \cup \omega$ is the effect of evaluating e.

To make contextual effects easier to work with, we introduce some shorthand. We write Φ^{α}, Φ^{ε}, and Φ^{ω} for the prior, current, and future effect components, respectively, of Φ. We also write $\Phi_{\emptyset}$ for the empty effect $[1; \emptyset; 1]$—by subsumption, discussed below, an expression with this effect may appear in any context. For brevity, whenever it is clear we will refer to contextual effects simply as *effects*.

5.2.2 Typing

Figure 5.2 presents our contextual type and effect system. The rules prove judgments of the form $\Phi; \Gamma \vdash e : \tau$, meaning in type environment Γ, expression e has type τ and contextual effect Φ.

Types τ, listed in Figure 5.1, include the integer type int; reference types $ref^{\varepsilon}(\tau)$, which denote a reference to memory location of type τ where the reference itself is annotated with a label $L \in \varepsilon$; and function types $\tau \rightarrow^{\Phi} \tau'$, where τ and τ' are the domain and range types, respectively, and the function has contextual effect Φ. Environments Γ, defined in Figure 5.1, are maps from variable names or (unlabeled) pointers to types.

The first two rules, [TINT] and [TVAR], assign the expected types and the empty effect, since values have no effect. Rule [TLAM] types the function body e and anno-

$$[\text{TINT}]\frac{}{\Phi_\emptyset; \Gamma \vdash n : int} \qquad [\text{TVAR}]\frac{\Gamma(x) = \tau}{\Phi_\emptyset; \Gamma \vdash x : \tau}$$

$$[\text{TLAM}]\frac{\Phi; \Gamma, x : \tau' \vdash e : \tau}{\Phi_\emptyset; \Gamma \vdash \lambda x.e : \tau' \rightarrow^{\Phi} \tau} \qquad [\text{TAPP}]\frac{\begin{array}{c}\Phi_1; \Gamma \vdash e_1 : \tau_1 \rightarrow^{\Phi_f} \tau_2 \\ \Phi_2; \Gamma \vdash e_2 : \tau_1 \\ \Phi_1 \rhd \Phi_2 \rhd \Phi_f \hookrightarrow \Phi\end{array}}{\Phi; \Gamma \vdash e_1\ e_2 : \tau_2}$$

$$[\text{TREF}]\frac{\Phi; \Gamma \vdash e : \tau}{\Phi; \Gamma \vdash \mathsf{ref}^L\ e : ref^{\{L\}}(\tau)} \qquad [\text{TDEREF}]\frac{\begin{array}{c}\Phi_1; \Gamma \vdash e : ref^{\varepsilon}(\tau) \\ \Phi_2^{\varepsilon} = \varepsilon \qquad \Phi_1 \rhd \Phi_2 \hookrightarrow \Phi\end{array}}{\Phi; \Gamma \vdash\, !\, e : \tau}$$

$$[\text{TASSIGN}]\frac{\begin{array}{c}\Phi_1; \Gamma \vdash e_1 : ref^{\varepsilon}(\tau) \qquad \Phi_2; \Gamma \vdash e_2 : \tau \qquad \Phi_3^{\varepsilon} = \varepsilon \\ \Phi_1 \rhd \Phi_2 \rhd \Phi_3 \hookrightarrow \Phi\end{array}}{\Phi; \Gamma \vdash e_1 := e_2 : \tau}$$

$$[\text{TLOC}]\frac{\Gamma(r) = \tau}{\Phi_\emptyset; \Gamma \vdash r_L : ref^{\{L\}}(\tau)} \qquad [\text{TSUB}]\frac{\begin{array}{c}\Phi'; \Gamma \vdash e : \tau' \\ \tau' \leq \tau \qquad \Phi' \leq \Phi\end{array}}{\Phi; \Gamma \vdash e : \tau}$$

$$[\text{XFLOW-CTXT}]\frac{\begin{array}{c}\Phi_1 = [\alpha_1; \varepsilon_1; (\varepsilon_2 \cup \omega_2)] \qquad \Phi_2 = [(\varepsilon_1 \cup \alpha_1); \varepsilon_2; \omega_2] \\ \Phi = [\alpha_1; (\varepsilon_1 \cup \varepsilon_2); \omega_2]\end{array}}{\Phi_1 \rhd \Phi_2\ \hookrightarrow\ \Phi}$$

$$[\text{SINT}]\frac{}{int \leq int} \qquad [\text{SREF}]\frac{\tau \leq \tau' \qquad \tau' \leq \tau \qquad \varepsilon \subseteq \varepsilon'}{ref^{\varepsilon}(\tau) \leq ref^{\varepsilon'}(\tau')}$$

$$[\text{SFUN}]\frac{\begin{array}{c}\tau_1' \leq \tau_1 \qquad \tau_2 \leq \tau_2' \\ \Phi \leq \Phi'\end{array}}{\tau_1 \rightarrow^{\Phi} \tau_2 \leq \tau_1' \rightarrow^{\Phi'} \tau_2'} \qquad [\text{SCTXT}]\frac{\begin{array}{c}\alpha_2 \subseteq \alpha_1 \qquad \varepsilon_1 \subseteq \varepsilon_2 \\ \omega_2 \subseteq \omega_1\end{array}}{[\alpha_1; \varepsilon_1; \omega_1] \leq [\alpha_2; \varepsilon_2; \omega_2]}$$

Figure 5.2: Typing

tates the function's type with the effect of e. The expression as a whole has no effect, since the function produces no run-time effects until it is actually called. Rule [TAPP] types function application, which combines Φ_1, the effect of e_1, with Φ_2, the effect of e_2, and Φ_f, the effect of the function. We specify the sequencing of effects with the combinator $\Phi_1 \rhd \Phi_2\ \hookrightarrow \Phi$, defined by [XFLOW-CTXT]. Since e_1 evaluates before e_2, this rule requires that the future effect of e_1 be $\varepsilon_2 \cup \omega_2$, i.e., everything that happens during the evaluation of e_2, captured by ε_2, plus everything that happens after, captured by ω_2. Similarly, the past effect of e_2 must be $\varepsilon_1 \cup \alpha_1$, since e_2 is evaluated just after e_1. Lastly, the effect Φ of the entire expression has α_1 as its prior effect, since e_1 is evaluated first; ω_2 as its future effect, since e_2 is evaluated last; and $\varepsilon_1 \cup \varepsilon_2$ as its current effect,

since both e_1 and e_2 are evaluated. We write $\Phi_1 \rhd \Phi_2 \rhd \Phi_3 \hookrightarrow \Phi$ as shorthand for $(\Phi_1 \rhd \Phi_2 \hookrightarrow \Phi') \wedge (\Phi' \rhd \Phi_3 \hookrightarrow \Phi)$.

[TREF] types memory allocation, which has no effect but places the annotation L into a singleton effect $\{L\}$ on the output type. This singleton effect can be increased as necessary by using subsumption. [TDEREF] types the dereference of a memory location of type $ref^{\varepsilon}(\tau)$. In a standard effect system, the effect of $!\,e$ is the effect of e plus the effect ε of accessing the pointed-to memory. Here, the effect of e is captured by Φ_1, and because the dereference occurs after e is evaluated, [TDEREF] puts Φ_1 in sequence just before some Φ_2 such that Φ_2's current effect is ε. Therefore by [XFLOW-CTXT], Φ^{ε} is $\Phi_1^{\varepsilon} \cup \varepsilon$, and e's future effect Φ_1^{ω} must include ε and the future effect of Φ_2. On the other hand, Φ_2^{ω} is unconstrained by this rule, but it will be constrained by the context, assuming the dereference is followed by another expression. [TASSIGN] is similar to [TDEREF], combining the effects Φ_1 and Φ_2 of its subexpressions with a Φ_3 whose current effect is ε. [TLOC] gives a pointer r_L the type of a reference to the type of r in Γ.

Finally, [TSUB] introduces subsumption on types and effects. The judgments $\tau' \leq \tau$ and $\Phi' \leq \Phi$ are defined at the bottom of Figure 5.2. [SINT], [SREF], and [SFUN] are standard, with the usual co- and contravariance where appropriate. [SCTXT] defines subsumption on effects, which is covariant in the current effect, as expected, and contravariant in both the prior and future effects. To understand the contravariance, first consider an expression e with future effect ω_1. Since ω_1 should contain (i.e., be a superset of) the locations that may be accessed in the future, we can use e in any context that accesses *at most* locations in ω_1. Similarly, since past effects should contain the locations that were accessed in the past, we can use e in any context that accessed at most locations in α_1.

5.3 Operational semantics

As discussed in the introduction, to establish the soundness of the static semantics we must address two concerns. First, we must give an operational semantics that specifies the run-time contextual effects of each subterm e appearing in the evaluation of a term e_p. Second, we must find a way to match up subterms e that arise in the evaluation of e_p with the corresponding terms e' in the unevaluated e_p, to see whether the effects ascribed to the original terms e' by the type system approximate the actual effects of the subterms e. This section defines an operational semantics that addresses the first concern, and the next section augments it to address the second concern, allowing us to prove our system sound.

5.3.1 The problem of future effects

Consider an expression e appearing in program e_p. We write $e_p = C[e]$ for a context C, to make this relationship more clear. Using a small-step operational semantics, we can intuitively view the contextual effects of e as follows:

$$\underbrace{C[e] \rightarrow \cdots \rightarrow}_{\text{prior effect } \alpha} C'[e] \overbrace{\underbrace{\rightarrow C'[e'] \rightarrow \cdots \rightarrow}_{\text{standard effect } \varepsilon}}^{\text{evaluation of } e} C'[v] \underbrace{\rightarrow \cdots \rightarrow v_p}_{\text{future effect } \omega}$$

(The evaluation of e_p could contain several evaluations of e, each of which could differ from e according to previous substitutions of e's free variables, but we ignore these difficulties for now and consider them in the next section.)

For this evaluation, the actual, run-time prior effect α of e is the effect of the evaluation that occurs before e starts evaluating, the actual standard effect ε of e is the effect of the evaluation of e to a value v, and the actual future effect ω of e is the effect of the remainder of the computation. For every expression in the program, there exist similar partitions of the evaluation to define the appropriate contextual effects.

However, while this picture is conceptually clear, formalizing contextual effects, particularly future effects, is awkward in small-step semantics. Suppose we have some contextual effect Φ associated with subterm e in the context $C'[e]$ above. Then Φ^ω, the future effect of subterm e, models everything that happens after we evaluate to $C'[v]$—but that happens some arbitrary number of steps after we begin evaluating $C'[e]$, making it difficult to associate with the subterm e. We could solve this problem by inserting "brackets" into the semantics to identify the end of a subterm's evaluation, but that adds complication, especially since there are many different subterms whose contextual effects we wish to track and prove sound.

Our solution to this problem is to use big-step semantics, since in big-step semantics, each subderivation is a full evaluation. This lets us easily identify both the beginning and the end of each sub-evaluation in the derivation tree, and gives us a natural specification of contextual effects.

5.3.2 Big-step semantics

Figure 5.3 shows key rules in a big-step operational semantics for our language. Reductions operate on *configurations* $\langle \alpha, \omega, H, e \rangle$, where α and ω are the sets of locations accessed before and after that point in the evaluation, respectively; H is the heap (a map from locations r to values); and e is the expression to be evaluated. Evaluations have the form

$$\langle \alpha, \omega, H, e \rangle \xrightarrow{\varepsilon} \langle \alpha', \omega', H', R \rangle$$

where ε is the effect of evaluating e and R is the result of reduction, either a value v or $\mathbf{err}$, indicating evaluation failed. Intuitively, as evaluation proceeds, labels move from the future effect ω to the past effect α.

With respect to the definitions of Section 5.3.1, the prior effect α in Section 5.3.1 corresponds to α here, and the future effect ω in Section 5.3.1 corresponds to ω' here. The future effect ω before the evaluation of e contains both the future and the standard effect of e, i.e., $\omega = \omega \cup \varepsilon$. Similarly, the past effect α' after the evaluation of e contains the past effect α and the effect of e, i.e., $\alpha' = \alpha \cup \varepsilon$. We prove below that our semantics preserves this property.

The reduction rules are straightforward. [ID] reduces a value to itself without changing the state or the effects. [REF] generates a fresh location r, which is bound in the heap to v and evaluates to r_L. [DEREF] reads the location r in the heap and adds L to the standard evaluation effect. This rule requires that the future effect after evaluating e have the form $\omega' \cup \{L\}$, i.e., L must be in the future effect after evaluating e, but prior

to dereferencing the result. Then L is added to α' in the output configuration of the rule. Notice that $\omega' \cup \{L\}$ is a standard union, hence L may also be in ω', which allows the same location to be accessed multiple times. Also note that we require L to be in the future effect just after the evaluation of e, but do not require that it be in ω. However, this will actually hold—below we prove that $\omega = \omega' \cup \{L\} \cup \varepsilon$, and in general when the semantics takes a step, effects move from the future to the past. [ASSIGN] behaves similarly to [DEREF]. [CALL] evaluates the first expression to a function, the second expression to a value, and then the function body with the formal argument replaced by the actual argument. Our semantics also includes rules [CALL-W], [DEREF-H-W] and [DEREF-L-W] that produce **err** when the program tries to access a location that is not in the future effect of the input, or when values are used at the wrong type. Our system includes similar error rules for assignment (not shown).

5.3.3 Standard effect soundness

We can now prove standard effect soundness. First, we prove an *adequacy* property of our semantics that helps ensure they make sense:

Lemma 5.3.1 (Adequacy of Semantics) *If*

$$\langle \alpha, \omega, H, e \rangle \xrightarrow{\varepsilon} \langle \alpha', \omega', H', v \rangle$$

then

$$\alpha' = \alpha \cup \varepsilon$$

and

$$\omega = \omega' \cup \varepsilon$$

This lemma formalizes our intuition that labels move from the future to prior effect during evaluation.

We can then prove that the static Φ^{ε} associated to a term by our type and effect system soundly approximates the actual effect ε of an expression. We ignore actual effects α and ω by setting them to 1. We say heap H is well-typed under Γ, written $\Gamma \vdash H$, if $\mathit{dom}(\Gamma) = \mathit{dom}(H)$ and for every $r \in \mathit{dom}(H)$, we have $\Phi_\emptyset; \Gamma \vdash H(r) : \Gamma(r)$. The standard effect soundness lemma is:

Theorem 5.3.2 (Standard Effect Soundness) *If*

1. $\Phi; \Gamma \vdash e : \tau$*,*
2. $\Gamma \vdash H$ *and*
3. $\langle 1, 1, H, e \rangle \xrightarrow{\varepsilon} \langle 1, 1, H', R \rangle$

then there is a Γ' *such that:*

1. *R is a value* v *for which* $\Phi_\emptyset; (\Gamma', \Gamma) \vdash v : \tau$*,*
2. $(\Gamma', \Gamma) \vdash H'$ *and*

3. $\varepsilon \subseteq \Phi^{\varepsilon}$.

Here (Γ', Γ) is the concatenation of environments Γ' and Γ. The proof of this theorem is by induction on the evaluation derivation, and follows traditional type-and-effect system proofs, adapted for our semantics.

Next, we prove that if the program evaluates to a value, then there is a *canonical evaluation* in which the program evaluates to the same value, but starting with an empty α and ending with an empty ω. This will produce an evaluation derivation with the *most precise* α and ω values for every configuration, which we can then prove we soundly approximate using our type and effect system.

Lemma 5.3.3 (Canonical Evaluation) *If*

$$\langle 1, 1, H, e\rangle \xrightarrow{\varepsilon} \langle 1, 1, H', v\rangle$$

then there exists a derivation

$$\langle \emptyset, \varepsilon, H, e\rangle \xrightarrow{\varepsilon} \langle \varepsilon, \emptyset, H', v\rangle$$

5.4 Contextual effect soundness

Now we turn to proving contextual effect soundness. We aim to show that the prior and future effect of some subterm e of a program e_p approximate the evaluation of e_p before and after, respectively, the evaluation of e. Suppose for the moment that e_p contains no function applications. As a result, an evaluation derivation D_p of e_p according to the operational semantics in Figure 5.3 will be isomorphic to a typing derivation T_p of e_p according to the rules in Figure 5.2. In this situation, soundness for contextual effects is easy to define. For any subterm e of e_p, we have an evaluation derivation D and a typing derivation T:

$$D :: \langle \alpha, \omega, H, e\rangle \xrightarrow{\varepsilon} \langle \alpha', \omega', H', v\rangle \qquad\qquad T :: \Phi; \Gamma \vdash e : \tau$$

where D is a subderivation of D_p and T is a subderivation of T_p. Then the prior and future effects computed by our contextual effect system are sound if $\alpha \subseteq \Phi^{\alpha}$ (the effect of the evaluation before e is contained in Φ^{α}) and $\omega' \subseteq \Phi^{\omega}$ (the effect of the evaluation after v is contained in Φ^{ω}).

For example, consider the evaluation of $!\,(\mathsf{ref}^{L}\ n)$.

$$\text{[DEREF]}\ \frac{\text{[REF]}\ \dfrac{\text{[ID]}\ \dfrac{}{\langle \emptyset, \emptyset \cup \{L\}, H, n\rangle \to \langle \emptyset, \emptyset \cup \{L\}, H, n\rangle}}{\langle \emptyset, \emptyset \cup \{L\}, H, \mathsf{ref}^{L}\ n\rangle \to \langle \emptyset, \emptyset \cup \{L\}, (H, r_L \mapsto n), r_L\rangle}}{\langle \emptyset, \emptyset \cup \{L\}, H, !\,(\mathsf{ref}^{L}\ n)\rangle \xrightarrow{\{L\}} \langle \emptyset \cup \{L\}, \emptyset, (H, r_L \mapsto n), n\rangle}$$

Here is the typing derivation (where we have rolled a use of [TSUB] into [TINT]):

$$\text{[TDEREF]}\ \frac{[\emptyset; \{L\}; \emptyset]^{\varepsilon} = \{L\} \qquad [\emptyset; \emptyset; \{L\}] \rhd [\emptyset; \{L\}; \emptyset] \hookrightarrow [\emptyset; \{L\}; \emptyset] \qquad \text{[TREF]}\ \dfrac{\text{[TINT']}\ \dfrac{}{[\emptyset; \emptyset; \{L\}]; \cdot \vdash n : int}}{[\emptyset; \emptyset; \{L\}]; \cdot \vdash \mathsf{ref}^{L}\ n : ref^{L}(int)}}{[\emptyset; \{L\}; \emptyset]; \cdot \vdash !\,(\mathsf{ref}^{L}\ n) : int}$$

We can see that these derivations are isomorphic, and thus it is easy to read the contextual effect from the typing derivation for $\mathsf{ref}^L\ n$ and to match it up with the actual effect of the corresponding subderivation of the evaluation derivation.

Unfortunately, function applications add significant complication because D_p and T_p are no longer isomorphic. Indeed, a subterm e of the original program e_p may appear multiple times in D_p, possibly with substitutions applied to it. For example, consider the term $(\lambda x.\,!\,x;!\,x)\ \mathsf{ref}^L\ n$ (where we introduce the sequencing operator ; with the obvious semantics, for brevity), typed as:

$$\text{[TAPP]}\ \frac{\text{[TLAM]}\ \dfrac{\Phi_f;\Gamma, x : \mathit{ref}^{\{L\}}(\mathit{int}) \vdash\ !\,x;!\,x : \mathit{int}}{\begin{array}{c}\Phi_\emptyset;\Gamma \vdash \lambda x.\,!\,x;!\,x : \mathit{ref}^{\{L\}}(\mathit{int}) \rightarrow^{\Phi_f} \mathit{int} \quad (T_1)\\ \Phi_2;\Gamma \vdash \mathsf{ref}^L\ n : \mathit{ref}^{\{L\}}(\mathit{int}) \quad (T_2)\end{array}} \qquad \Phi_\emptyset \rhd \Phi_2 \rhd \Phi_f \hookrightarrow \Phi}{\Phi;\Gamma \vdash (\lambda x.\,!\,x;!\,x)\ \mathsf{ref}^L\ n : \mathit{int}}$$

The evaluation derivation has the following structure:

$$\text{[CALL]}\ \frac{\begin{array}{c}\langle \emptyset, \emptyset \cup \{L\}, H, (\lambda x.\,!\,x;!\,x)\rangle \rightarrow \langle \emptyset, \emptyset \cup \{L\}, H, (\lambda x.\,!\,x;!\,x)\rangle \quad (1)\\ \langle \emptyset, \emptyset \cup \{L\}, H, \mathsf{ref}^L\ n\rangle \rightarrow \langle \emptyset, \emptyset \cup \{L\}, H', r_L\rangle \quad (2)\\ \langle \emptyset, \emptyset \cup \{L\}, H', [r_L \mapsto (!\,x;!\,x)]x\rangle \xrightarrow{\{L\}} \langle \emptyset \cup \{L\}, \emptyset, H', n\rangle \quad (3)\end{array}}{\langle \emptyset, \emptyset \cup \{L\}, H, (\lambda x.\,!\,x;!\,x)\ \mathsf{ref}^L\ n\rangle \xrightarrow{\{L\}} \langle \emptyset \cup \{L\}, \emptyset, H', n\rangle}$$

where $H' = (H, r_L \mapsto n)$. Subderivations *(1)* and *(2)* correspond to the two subderivations (T_1) and (T_2) of [TAPP], but there is no analogue for subderivation *(3)*, which captures the actual evaluation of the function. Clearly this relates to the function's effect Φ_f, but how exactly is not structurally apparent from the derivation. Returning to our example, we must match up the effect in the typing derivation for $!\,x$, which is part of the typing of the function $(\lambda x.\,!\,x;!\,x)$, with evaluation of $!\,r_L$ that occurs when the function evaluates in subderivation *(3)*.

To do this, we instrument the big-step semantics from Figure 5.3 with typing derivations, and define exactly how to associate a typing derivation with each derived subterm in an evaluation derivation. The key property of the resulting *typed operational semantics* is that the contextual effect Φ associated with a subterm e in the original typing derivation T_p is also associated with all terms derived from e via copying or substitution. In the example, the relevant typing subderivation for $!\,x$ in T_p will be copied and substituted according to the evaluation so that it can be matched with $!\,r_L$ in subderivation *(3)*.

5.4.1 Typed operational semantics

In our typed operational semantics, evaluations have the form:

$$\langle T, \alpha, \omega, H, e\rangle \xrightarrow{\varepsilon} \langle T', \alpha', \omega', H', v\rangle$$

where T is a typing derivation for the expression e, and T' is a typing derivation for v:

$$T :: \Phi;\Gamma \vdash e : \tau \qquad\qquad T' :: \Phi_\emptyset;(\Gamma', \Gamma) \vdash v : \tau$$

Note that we include T' in our rules mostly to emphasize that v is well-typed with the same type as e. The only information from T' we need that is not present in T is the new environment (Γ', Γ), which may contain the types of pointers newly allocated in the heap during the evaluation of e. Also, the environments Γ and Γ' only refer to heap locations, since e and v have no free variables and could always be typed under the empty environment.

Figure 5.4 presents the typed evaluation rules. New hypotheses are highlighted with a gray background. While these rules look complicated, they are actually quite easy to construct. We begin with the original rules in Figure 5.3, add a typing derivation to each configuration, and then specify appropriate hypotheses about each typing derivation to connect up the derivation of the whole term with the derivation of each of the subterms. We discuss this process for each of the rules.

[ID-A] is the same as [ID], except we introduce typing derivations T_v and T'_v for the left- and right-hand sides of the evaluation, respectively. T_v may be any typing derivation that assigns a type to v. Here, and in the other rules in the typed operational semantics, we allow subsumption in the typing derivations on the left-hand side of a reduction. Thus T_v may type the value v under some effect Φ that is not $\Phi_\emptyset$. The output typing derivation T'_v is the same as T_v, except it uses the effect $\Phi_\emptyset$ (recall the only information we use from T'_v is the new environment, which is this case is unchanged from T_v).

[REF-A] is a more complicated case. Here the typing derivation T must (by observation of the rules in Figure 5.2) assign $\mathsf{ref}^L\ e$ a type $ref^{\varepsilon}(\tau)$ and some effect Φ. By inversion, then, we know that T must in fact assign the subterm e the type τ as witnessed by some typing derivation T', which we use in the typed evaluation of e. We allow $\Phi' \leq \Phi$ to account for subsumption applied to the term $\mathsf{ref}^L\ e$. Note that this rule does not specify how to construct T' from T. Later on, we will prove that if there is a valid standard reduction of a well-typed term, then there is a valid typed reduction of the same term. Continuing with the rule, our semantics assigns some typing derivation T_v to v. Then the output typing derivation T_r should assign a type to r_L. Hence we take the environment Γ' from T_v, which contains types for locations in the heap allocated thus far, and extend it with a new binding for r of the correct type.

[DEREF-A] follows the same pattern as above. Given the initial typing derivation T of the term $!\,e$, we assume there exists a typing derivation T' of the appropriate shape for subterm e. Reducing e yields a new typing derivation T_r, and the final typing derivation T_v assigns the type τ to the value $H'(r)$ returned by the dereference. As above, we add subtyping constraints $\Phi' \leq \Phi$ and $\tau' \leq \tau$ to account for subsumption of the term $!\,e$. The most interesting feature of this rule is the last constraint, $\Phi_1 \rhd [\alpha_1; \varepsilon'; \omega_1] \hookrightarrow \Phi'$, which states that the effect $\Phi \geq \Phi'$ of the whole expression $!\,e$ (from typing derivation T) must contain the effect Φ_1 of e followed by some contextual effect containing standard effect ε'. Again, we will prove below that it is always possible to construct a typed derivation that satisfies this constraint, intuitively because [DEREF] from Figure 5.2 enforces exactly the same constraint. [ASSIGN-A] is similar to [DEREF].

[CALL-A] is the most complex of the four rules, but the approach is exactly the same as above. Starting with typing derivation T for the function application, we require that there exist typing derivations T_1 and T_2 for e_1 and e_2, where the type of e_2 is the domain type of e_1. Furthermore, T_f and T_{v_2} assign the same types as T_1 and T_2, respectively.

Then by the substitution lemma, we know there exists a typing derivation T_3 that assigns type τ to the function body e in which the formal x is mapped to the actual v_2. The output typing derivation T_v assigns v the same type τ as T_3 assigns to the function body. We finish the rule with the usual effect sequencing and subtyping constraints.

5.4.2 Soundness

The semantics in Figure 5.4 precisely associate a typing derivation—and most importantly, a contextual effect—with each subterm in an evaluation derivation. We prove soundness in two steps. First, we argue that given a typing derivation of a program and an evaluation derivation according to the rules in Figure 5.3, we can always construct a typed evaluation derivation.

Lemma 5.4.1 (Typed evaluation derivations exist) *If* $T :: \Phi; \Gamma \vdash e : \tau$ *and* $D :: \langle \alpha, \omega, H, e\rangle \xrightarrow{\varepsilon} \langle \alpha'\omega', H', v\rangle$ *where* $\Gamma \vdash H$*, then there exists* T_v *such that*

$$\langle T, \alpha, \omega, H, e\rangle \xrightarrow{\varepsilon} \langle T_v, \alpha', \omega', H', v\rangle$$

The proof is by induction on the evaluation derivation D. For each case, we show we can always construct a typed evaluation by performing inversion on the typing derivation T, using T's premises to apply the corresponding typed operational semantics rule. Due to subsumption, we cannot perform direct inversion on T. Instead, we used a number of inversion lemmas (not shown) that generalize the premises of the syntax-driven typing rule that applies to e, for any number of following [TSUB] applications.

Next, we prove that if we have a typed evaluation derivation, then the contextual effects assigned in the derivation soundly model the actual run-time effects. Since contextual effects are non-compositional, we reason about the soundness of contextual effects in a derivation in relation to its context inside a larger derivation. To do that, we use $E_1 \in E_2$ to denote that E_1 is a subderivation of E_2. We define the subderivation relation inductively on evaluation derivations in the typed operational semantics, with base cases corresponding to each evaluation rule, and one inductive case for transitivity. For example, given an application of [CALL-A] (uninteresting premises omitted):

$$\frac{\begin{array}{c} \vdots \\ E_1 :: \langle T_1, \alpha, \omega, H, e_1\rangle \xrightarrow{\varepsilon_1} \langle T_f, \alpha_1, \omega_1, H_1, \lambda x.e\rangle \\ E_2 :: \langle T_2, \alpha_1, \omega_1, H_1, e_2\rangle \xrightarrow{\varepsilon_2} \langle T_{v_2}, \alpha_2, \omega_2, H_2, v_2\rangle \\ E_3 :: \langle T_3, \alpha_2, \omega_2, H_2, [x \mapsto e]v_2\rangle \xrightarrow{\varepsilon_3} \langle T_v, \alpha', \omega', H', v\rangle \end{array}}{E :: \langle T, \alpha, \omega, H, e_1\ e_2\rangle \xrightarrow{\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3} \langle T', \alpha', \omega', H, v\rangle}$$

we have $E_1 \in E$, $E_2 \in E$ and $E_3 \in E$. The subderivation relationship is also transitive, i.e., if $E_1 \in E_2$ and $E_2 \in E_3$ then $E_1 \in E_3$.

The following lemma states that if E_2 is an evaluation derivation whose contextual effects are sound (premises 2, 5, and 6) and E_1 is a subderivation of E_2 (premise 3), then the effects of E_1 are sound (conclusions 2 and 3).

Lemma 5.4.2 (Soundness of sub-derivation contextual effects) *If*

1. $E_1 :: \langle T_1, \alpha_1, \omega_1, H_1, e_1\rangle \xrightarrow{\varepsilon_1} \langle T_{v_1}, \alpha_1', \omega_1', H_1', v_1\rangle$ *with* $T_1 :: \Phi_1; \Gamma_1 \vdash e_1 : \tau_1$,
2. $E_2 :: \langle T_2, \alpha_2, \omega_2, H_2, e_2\rangle \xrightarrow{\varepsilon_2} \langle T_{v_2}, \alpha_2', \omega_2', H_2', v_2\rangle$ *with* $T_2 :: \Phi_2; \Gamma_2 \vdash e_2 : \tau_2$,
3. $E_1 \in E_2$
4. $\Gamma_2 \vdash H_2$
5. $\alpha_2 \subseteq \Phi_2^\alpha$
6. $\omega_2 \subseteq \Phi_2^\omega$

then

1. $\Gamma_1 \vdash H_1$
2. $\alpha_1 \subseteq \Phi_1^\alpha$
3. $\omega_1 \subseteq \Phi_1^\omega$

The proof is by induction on $E_1 \in E_2$. The work occurs in the base cases of the $\in$ relation, and the transitivity case trivially applies induction.

The statement of Lemma 5.4.2 may seem odd: we assume a derivation's effects are sound and then prove the soundness of the effects of its subderivation(s). Nevertheless, this technique is efficacious. If E_2 is the topmost derivation (for the whole program) then the lemma can be trivially applied for E_2 and any of its subderivations, as α_2 and ω_2' will be $\emptyset$, and thus trivially approximated by the effects defined in Φ_2. Given this, and the fact (from Lemma 5.4.1) that typed derivations always exist, we can easily state and prove contextual effect soundness.

Theorem 5.4.3 (Contextual Effect Soundness) *Given a program e_p with no free variables, a typing derivation T and a (standard) evaluation D according to the rules in Figure 5.3, we can construct a typed evaluation derivation*

$$E :: \langle T, \emptyset, \varepsilon_p, \emptyset, e_p\rangle \xrightarrow{\varepsilon_p} \langle T_v, \varepsilon_p, \emptyset, H, v\rangle$$

such that for every subderivation E' of E:

$$E' :: \langle T', \alpha, \omega, H, e\rangle \xrightarrow{\varepsilon} \langle T_v, \alpha', \omega', H', v\rangle$$

with $T' :: \Phi; \Gamma \vdash e : \tau$, it is always the case that $\alpha \subseteq \Phi^\alpha$, $\varepsilon \subseteq \Phi^\varepsilon$, and $\omega' \subseteq \Phi^\omega$.

This theorem follows as a corollary of Lemmas 5.3.3, 5.4.1 and 5.4.2, since the initial heap and Γ are empty, and the whole program is typed under $[\emptyset; \varepsilon; \emptyset]$, where ε soundly approximates the effect of the whole program by Theorem 5.3.2.

The full (paper) proof can be found in Appendix C.

5.5 Mechanization

We encoded the above formalization and soundness proof using the Coq proof assistant. We were pleased that the mechanization of the system largely followed the paper proof, with only a few minor differences.

First, we used the framework developed by Aydemir et al. [8] for modeling bound and named variables, whereas the paper proof assumes alpha equivalence of all terms and does not reason about capturing and renaming.

Second, Lemma 5.4.2 states a property of all subderivations of a derivation. On paper, we had left the definition of subderivation informal, whereas we had to formally define it in Coq. This was straightforward if tedious. In Coq we defined $E \in E'$, described earlier, as an inductive relation, with one case for each premise of each evaluation rule.

While our mechanized proof is similar to our paper proof, it does have some awkwardness. Our encoding of typed operational semantics is dependent on typing derivations, and the encoding of the subderivation relation is dependent on typed evaluations. This causes the definitions of typed evaluations and subderivations to be dependent on large sets of variables, which decreases readability. We were unable to use Coq's system for implicit variables to address this issue, due to its current limitations.

In total, the formalization and proof scripts for the contextual effect system takes 5,503 lines of Coq, of which we wrote 2,692 lines and the remaining 2,811 lines came from Aydemir et al [8]. It took approximately ten days to encode the definitions and lemmas and do the proofs, starting from minimal Coq experience, limited to attending a tutorial at POPL 2008. It took roughly equal time and effort to construct the encodings as to do the actual proofs. In the process of performing the proofs, we discovered some typographical errors in the paper proof, and we found some cases where we had failed to account for possible subsumption in the type and effect system. Perhaps the biggest insight we gained was that to prove Lemma 5.4.2, we needed to do induction on the subderivation relation, rather than on the derivation itself.

$$[\textsc{Id}]\ \frac{}{\langle \alpha, \omega, H, v\rangle \xrightarrow{\emptyset} \langle \alpha, \omega, H, v\rangle}$$

Heaps $H ::= \emptyset \mid H, r \mapsto v$

$$[\textsc{Ref}]\ \frac{\langle \alpha, \omega, H, e\rangle \xrightarrow{\varepsilon} \langle \alpha', \omega', H', v\rangle \qquad r \notin \mathit{dom}(H')}{\langle \alpha, \omega, H, \mathsf{ref}^L\ e\rangle \xrightarrow{\varepsilon} \langle \alpha', \omega', (H', r \mapsto v), r_L\rangle}$$

$$[\textsc{Deref}]\ \frac{\langle \alpha, \omega, H, e\rangle \xrightarrow{\varepsilon} \langle \alpha', \omega' \cup \{L\}, H', r_L\rangle \qquad r \in \mathit{dom}(H')}{\langle \alpha, \omega, H, !\, e\rangle \xrightarrow{\varepsilon \cup \{L\}} \langle \alpha' \cup \{L\}, \omega', H', H'(r)\rangle}$$

$$[\textsc{Assign}]\ \frac{\begin{array}{c}\langle \alpha, \omega, H, e_1\rangle \xrightarrow{\varepsilon_1} \langle \alpha_1, \omega_1, H_1, r_L\rangle \\ \langle \alpha_1, \omega_1, H_1, e_2\rangle \xrightarrow{\varepsilon_2} \langle \alpha_2, \omega_2 \cup \{L\}, (H_2, r \mapsto v'), v\rangle\end{array}}{\langle \alpha, \omega, H, e_1 := e_2\rangle \xrightarrow{\varepsilon_1 \cup \varepsilon_2 \cup \{L\}} \langle \alpha_2 \cup \{L\}, \omega_2, (H_2, r \mapsto v), v\rangle}$$

$$[\textsc{Call}]\ \frac{\begin{array}{c}\langle \alpha, \omega, H, e_1\rangle \xrightarrow{\varepsilon_1} \langle \alpha_1, \omega_1, H_1, \lambda x.e\rangle \\ \langle \alpha_1, \omega_1, H_1, e_2\rangle \xrightarrow{\varepsilon_2} \langle \alpha_2, \omega_2, H_2, v_2\rangle \\ \langle \alpha_2, \omega_2, H_2, [v_2 \mapsto e]x\rangle \xrightarrow{\varepsilon_3} \langle \alpha', \omega', H', v\rangle\end{array}}{\langle \alpha, \omega, H, e_1\ e_2\rangle \xrightarrow{\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3} \langle \alpha', \omega', H', v\rangle}$$

$$[\textsc{Call-W}]\ \frac{\langle \alpha, \omega, H, e_1\rangle \xrightarrow{\varepsilon_1} \langle \alpha', \omega', H', v\rangle \qquad v \neq \lambda x.e}{\langle \alpha, \omega, H, e_1\ e_2\rangle \xrightarrow{\emptyset} \langle \alpha, \omega, H, \mathbf{err}\rangle}$$

$$[\textsc{Deref-H-W}]\ \frac{\langle \alpha, \omega, H, e\rangle \xrightarrow{\varepsilon} \langle \alpha', \omega', H', r_L\rangle \qquad r \notin \mathit{dom}(H')}{\langle \alpha, \omega, H, !\, e\rangle \xrightarrow{\emptyset} \langle \alpha, \omega, H, \mathbf{err}\rangle}$$

$$[\textsc{Deref-L-W}]\ \frac{\langle \alpha, \omega, H, e\rangle \xrightarrow{\varepsilon} \langle \alpha', \omega', H', r_L\rangle \qquad r \in \mathit{dom}(H') \qquad L \notin \omega'}{\langle \alpha, \omega, H, !\, e\rangle \xrightarrow{\emptyset} \langle \alpha, \omega, H, \mathbf{err}\rangle}$$

Figure 5.3: Operational Semantics

$$\text{[ID-A]}\ \frac{T_v :: \Phi; \Gamma \vdash v : \tau \qquad T'_v :: \Phi_\emptyset; \Gamma \vdash v : \tau}{\langle T_v, \alpha, \omega, H, v\rangle \xrightarrow{\emptyset} \langle T'_v, \alpha, \omega, H, v\rangle}$$

$$\text{[REF-A]}\ \frac{\begin{array}{c}\langle T', \alpha, \omega, H, e\rangle \xrightarrow{\varepsilon} \langle T_v, \alpha', \omega', H', v\rangle \qquad r \notin \mathit{dom}(H) \\ T :: \Phi; \Gamma \vdash \mathsf{ref}^L\ e : \mathit{ref}^\varepsilon(\tau) \qquad T' :: \Phi'; \Gamma \vdash e : \tau \\ T_v :: \Phi_\emptyset; \Gamma' \vdash v : \tau \qquad T_r :: \Phi_\emptyset; (\Gamma', r \mapsto \tau) \vdash r_L : \mathit{ref}^\varepsilon(\tau) \qquad \Phi' \leq \Phi\end{array}}{\langle T, \alpha, \omega, H, \mathsf{ref}^L\ e\rangle \xrightarrow{\varepsilon} \langle T_r, \alpha', \omega', (H', r \mapsto v), r_L\rangle}$$

$$\text{[DEREF-A]}\ \frac{\begin{array}{c}\langle T', \alpha, \omega, H, e\rangle \xrightarrow{\varepsilon} \langle T_r, \alpha', \omega' \cup \{L\}, H', r_L\rangle \qquad r \in \mathit{dom}(H') \\ T :: \Phi; \Gamma \vdash !\, e : \tau \qquad T' :: \Phi_1; \Gamma \vdash e : \mathit{ref}^{\varepsilon'}(\tau') \\ T_r :: \Phi_\emptyset; \Gamma' \vdash r_L : \mathit{ref}^{\varepsilon'}(\tau') \qquad T_v :: \Phi_\emptyset; \Gamma' \vdash H'(r) : \tau \\ \Phi' \leq \Phi \qquad \tau' \leq \tau \qquad \Phi_1 \rhd [\alpha_1; \varepsilon'; \omega_1] \hookrightarrow \Phi'\end{array}}{\langle T, \alpha, \omega, H, !\, e\rangle \xrightarrow{\varepsilon \cup \{L\}} \langle T_v, \alpha' \cup \{L\}, \omega', H', H'(r)\rangle}$$

$$\text{[ASSIGN-A]}\ \frac{\begin{array}{c}\langle T_1, \alpha, \omega, H, e_1\rangle \xrightarrow{\varepsilon_1} \langle T_r, \alpha_1, \omega_1, H_1, r_L\rangle \\ \langle T_2, \alpha_1, \omega_1, H_1, e_2\rangle \xrightarrow{\varepsilon_2} \langle T_v, \alpha_2, \omega_2 \cup \{L\}, (H_2, r \mapsto v'), v\rangle \\ T :: \Phi; \Gamma \vdash e_1 := e_2 : \tau \qquad T_1 :: \Phi_1; \Gamma \vdash e_1 : \mathit{ref}^\varepsilon(\tau') \\ T_r :: \Phi_\emptyset; \Gamma_1 \vdash r_L : \mathit{ref}^\varepsilon(\tau') \qquad T_2 :: \Phi_2; \Gamma_1 \vdash e_2 : \tau' \\ T_v :: \Phi_\emptyset; \Gamma_2 \vdash v : \tau' \qquad T'_v :: \Phi_\emptyset; \Gamma_2 \vdash v : \tau \\ \Phi' \leq \Phi \qquad \tau' \leq \tau \qquad \Phi_1 \rhd \Phi_2 \rhd [\alpha_3; \varepsilon; \omega_3] \hookrightarrow \Phi'\end{array}}{\langle T, \alpha, \omega, H, e_1 := e_2\rangle \xrightarrow{\varepsilon_1 \cup \varepsilon_2 \cup \{L\}} \langle T'_v, \alpha_2 \cup \{L\}, \omega_2, (H_2, r \mapsto v), v\rangle}$$

$$\text{[CALL-A]}\ \frac{\begin{array}{c}\langle T_1, \alpha, \omega, H, e_1\rangle \xrightarrow{\varepsilon_1} \langle T_f, \alpha_1, \omega_1, H_1, \lambda x.e\rangle \\ \langle T_2, \alpha_1, \omega_1, H_1, e_2\rangle \xrightarrow{\varepsilon_2} \langle T_{v_2}, \alpha_2, \omega_2, H_2, v_2\rangle \\ \langle T_3, \alpha_2, \omega_2, H_2, [x \mapsto e]v_2\rangle \xrightarrow{\varepsilon_3} \langle T_v, \alpha', \omega', H', v\rangle \\ T :: \Phi; \Gamma \vdash e_1\ e_2 : \tau \qquad T_1 :: \Phi_1; \Gamma \vdash e_1 : \tau_1 \to^{\Phi_f} \tau_2 \\ T_f :: \Phi_\emptyset; \Gamma_1 \vdash \lambda x.e : \tau_1 \to^{\Phi_f} \tau_2 \qquad T_2 :: \Phi_2; \Gamma_1 \vdash e_2 : \tau_1 \\ T_{v_2} :: \Phi_\emptyset; \Gamma_2 \vdash v_2 : \tau_1 \qquad T_3 :: \Phi_f; \Gamma_2 \vdash [v_2 \mapsto e]x : \tau \\ T_v :: \Phi_\emptyset; \Gamma_3 \vdash v : \tau \qquad \Phi_1 \rhd \Phi_2 \rhd \Phi_f \hookrightarrow \Phi' \qquad \Phi' \leq \Phi\end{array}}{\langle T, \alpha, \omega, H, e_1\ e_2\rangle \xrightarrow{\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3} \langle T_v, \alpha', \omega', H, v\rangle}$$

Figure 5.4: Typed operational semantics

Chapter 6

Implementation

6.1 Implementation Overview

This chapter presents an overview of all the analyses in LOCKSMITH. Fig. 6.1 shows the architecture of LOCKSMITH, which is structured as a series of sub-analyses that each generate and solve constraints. In this figure, plain boxes represent processes and shaded boxes represent data. LOCKSMITH is implemented using CIL, which parses the input C program and simplifies it to a core sub-language [114].

In the remainder of this chapter, we sketch each of LOCKSMITH's components and then summarize the results of applying LOCKSMITH to a benchmark suite. In the subsequent discussion, we will use the code in Fig. 6.2 as a running example.

The program in Fig. 6.2 begins by defining four global variables, locks lock1 and lock2 and integers count1 and count2. Then lines 4–8 define a function atomic_inc that takes pointers to a lock and an integer as arguments, and then increments the integer while holding the lock. The main function on lines 10–21 allocates an integer variable local, initializes the two locks, and then spawns three threads that execute functions thread1, thread2 and thread3, passing variable local to thread1 and NULL to thread2 and thread3. We annotate each thread creation and function call site, except calls to the special mutex initialization function, with an index i, whose use will be explained below. The thread executing thread1 (lines 23–28) first extracts the pointer-to-integer argument into variable y and then continuously increments the integer. The thread executing thread2 (lines 30–37) consists of an infinite loop that increases count1 while holding lock lock1 and count2 without holding a lock. The thread executing thread3 (lines 39–45) consists of an infinite loop that calls atomic_inc twice, to increment count1 under lock1 and count2 under lock2.

There are several interesting things to notice about the locking behavior in this program. First, observe that though the variable local is accessed both in the parent thread (lines 12,17) and its child thread thread1 (via the alias $*$y on line 26), no race is possible despite the lack of synchronization. This is because these accesses cannot occur simultaneously, because the parent only accesses local before the thread for thread1 is created, and never afterward. Thus both accesses are local to a particular thread. Second, tracking of lock acquires and releases must be flow-sensitive, so we know that the access on line 33 is guarded by a lock, and the access on line 35 is not. Lastly, the atomic_inc function is

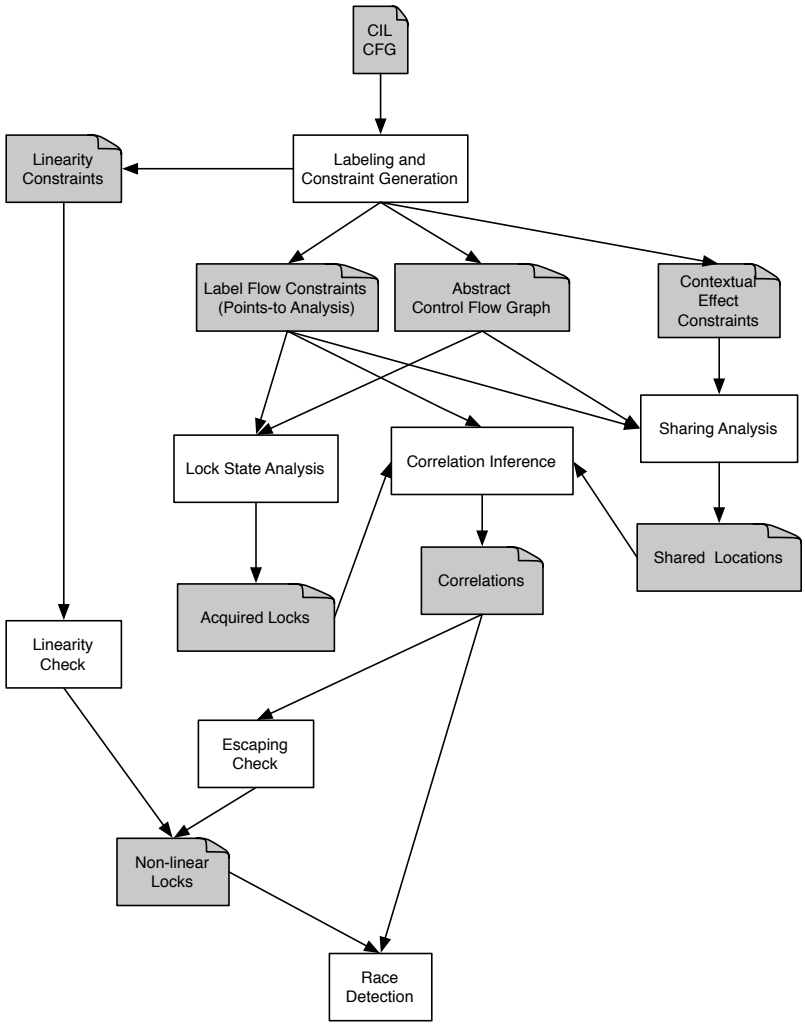


Figure 6.1: LOCKSMITH architecture

called twice (lines 42–43) with two different locks and integer pointers. We need context sensitivity to avoid conflating these two calls, which would lead to false alarms.

6.2 Labeling and constraint generation

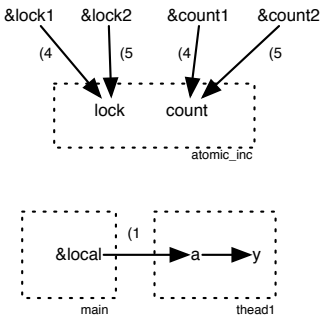
The first phase of LOCKSMITH is *labeling and constraint generation*, which traverses the CIL CFG and generates two key abstractions that form the basis of subsequent analyses: *label flow constraints*, to model the flow of data within the program, and *abstract control flow constraints*, to model the sequencing of key actions and relate them to the label flow constraints. Because a set label flow constraints can be conveniently visualized as a graph, we will often refer to them as a *label flow graph*, and do likewise for a set of abstract control flow constraints. Chapter 2 presents the notion of label flow analysis in detail.

```
1  pthread_mutex_t lock1, lock2;
2  int count1 = 0, count2 = 0;
3
4  void atomic_inc(pthread_mutex_t *lock, int *count) {
5    pthread_mutex_lock(lock);
6    *count++;
7    pthread_mutex_unlock(lock);
8  }
9
10 int main(void) {
11   pthread t1, t2, t3;
12   int local = 0;
13
14   pthread_mutex_init(&lock1, NULL);
15   pthread_mutex_init(&lock2, NULL);
16
17   local++;
18   pthread_create1(&t1, NULL, thread1, &local);
19   pthread_create2(&t2, NULL, thread2, NULL);
20   pthread_create3(&t3, NULL, thread3, NULL);
21 }
22
23 void *thread1(void *a) {
24   int *y = (int *) a; /* int* always */
25   while(1) {
26     *y++;  /* thread local */
27   }
28 }
29
30 void *thread2(void *c) {
31   while(1) {
32     pthread_mutex_lock(&lock1);
33     count1++;
34     pthread_mutex_unlock(&lock1);
35     count2++; /* access without lock */
36   }
37 }
38
39 void *thread3(void *b) {
40   while(1) {
41     /* needs polymorphism for atomic_inc */
42     atomic_inc4(&lock1, &count1);
43     atomic_inc5(&lock2, &count2);
44   }
45 }
```

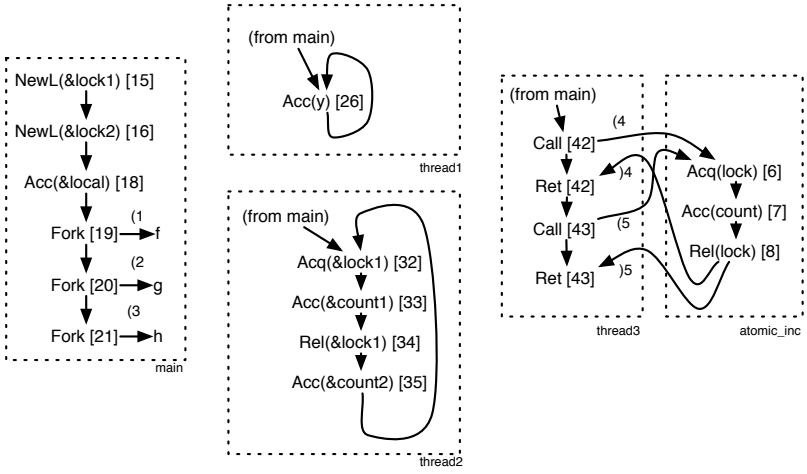
Figure 6.2: Example multi-threaded C program

6.2.1 Label flow graph

Fig. 6.3(a) shows the label flow graph for the example from Fig. 6.2. Nodes are static representations of the run-time memory locations (addresses) that contain locks or



(a) Label flow graph



(b) Abstract control flow graph

Figure 6.3: Constraint graphs generated for example in Fig. 6.2

other data. Edges represent the "flow" of data through the program [109, 132, 88], e.g., according to assignment statements or function calls. The source of a path in the label flow graph is an allocation site, e.g., it is the address of a global or local variable (e.g., &lock1, &count1, or &local in Fig. 6.2), or the representation of a program line which a malloc() occurs. We distinguish addresses of locks from those of other data (which may be subject to races); generally speaking we refer to the former using metavariable ℓ and the latter using metavariable ρ.

LOCKSMITH's label flow analysis is *field-sensitive* when modeling C struct types, in which each field of each instance of a struct is modeled separately. We found that field-sensitivity significantly improves precision. To make our algorithm sufficiently scalable, we modeled fields lazily [58]—only if (and when) a field was actually accessed by the program does LOCKSMITH model it, as opposed to eagerly tracking each field of a given instance from the time the instance is created. We found that over all benchmarks only 35%, on average, of the declared fields of struct variables in the program are actually accessed, and so the lazy approach afforded significant savings.

LOCKSMITH also tries to model C's `void*` types precisely, yet efficiently. In our final design, when a `void*` pointer might point to two different types, we assume that it is not used to cast between them, but rather that the programmer always casts the `void*` pointer to the correct type before using it (in the style of an untagged union). We also tried two alternative strategies. First, and most conservatively, if a type is cast to `void*`, we conflate *all* pointers in that type with each other and the `void*`. While sound, this technique is quite conservative, and the significant amount of false aliasing it produces degrades LOCKSMITH's subsequent analyses. A second alternative we considered behaves in exactly the same way, but only when more than one type is cast to the same `void*` pointer. Assuming a given `void*` is only cast to/from a single type, we can relate any pointers occurring within the type to the particular type instances cast to the `void*`, as if the type was never cast to `void*` in the first place. We found that approximately one third of all `void*` pointers in our benchmarks alias one type, so this strategy increased the precision compared to simply conflating all pointers casted to a `void*`. Nevertheless, we found that our final design is more precise *and* more efficient, in that it prunes several superficial or imprecise constraints.

To achieve context-sensitivity, we incorporate additional information about function calls into the flow graph. Input and output edges corresponding to a call indexed by i in the program are labeled with $(i$ and $)i$, respectively. During constraint resolution, we know that two edges correspond to the same call only if they are labeled by the same index. For example, in Fig. 6.3(a) the edges from $\&$lock1 and $\&$count1 are labeled with $(4$ since they arise from the call on line 42, and analogously the edges from $\&$lock2 and $\&$count2 are labeled with $(5$. We use a variation on *context-free language reachability (CFLR)* to compute the flow of data through the program [125]. In this particular example, since count is accessed with lock held, we would discover that count$i$ is accessed with lock$i$ held for $i \in 1..2$. Without the labeled edges, we could not distinguish the two call sites, and LOCKSMITH would lose precision. In particular, LOCKSMITH would think that lock could be either lock1 or lock2, and thus we would not know which one was held at the access to count, causing us to report a potential data race on line 6.

Section 6.9 discusses the label flow analysis in detail, not considering context sensitivity, while Section 6.11 discusses extensions to this analysis to handle struct and `void*` types. We initially present context-insensitive algorithms for each LOCKSMITH phase, and discuss context sensitivity for all parts in Section 6.12.

6.2.2 Abstract control flow graph

Fig. 6.3(b) shows the *abstract control flow graph* (ACFG) for the example from Fig. 6.2. Nodes in the ACFG capture operations in the program that are important for data race detection, and relate them to the labels from the label flow graph. ACFGs contain 7 kinds of nodes (the notation $[n]$ next to each node indicates the line number n from which the node is induced). NewL(ℓ) represents a statement that creates a new lock at location ℓ, and Acq(ℓ) and Rel(ℓ) denote the acquire and release, respectively, of the lock ℓ. Reads and writes of memory location ρ are represented by Acc(ρ). Thread creation is indicated by Fork nodes, which have two successors: the next statement in the parent thread, and the first statement of the child thread's called function. The edge to the

latter is annotated with an index just as in the label flow graph, to allow us to relate the two graphs. For example, the child edge for the Fork corresponding to line 18 is labeled with $(1$, which is the same annotation used for the edge from &local to a in the label flow graph. Lastly, function calls and returns are represented by Call and Ret nodes in the graph. For call site i, we label the edge to the callee with $(i$, and we label the return edge with $)i$, again to allow us to relate the label flow graph with the ACFG. The edges from a Call to the corresponding Ret allow us to flow information "around" callees, often increasing precision; we defer discussion of this feature to Section 6.9.3.

In addition to label flow constraints and the abstract control-flow graph, the first phase of LOCKSMITH generates *linearity constraints* and *contextual effect constraints*, which are discussed below.

6.3 Sharing analysis

The next LOCKSMITH phase determines the set of locations that could be potentially simultaneously accessed by two or more threads during a program's execution. We refer to these as the program's *shared locations*. We limit subsequent analysis for possible data races to these shared locations. In particular, if a location is *not* shared, then it need not be consistently accessed with a particular lock held. Moreover, if an access site (that is, a read or a write through a pointer) never targets a shared variable, it need not be considered by the analysis.

As shown in Fig. 6.1, this phase takes as input *contextual effect constraints*, which are also produced during labeling and constraint generation. In standard effect systems [155], the effect of a program statement is the set of locations that may be accessed (read or written) when the statement is executed. Our contextual effect system additionally determines, for each program state, the *future effect*, which contains the locations that may be accessed by the remainder of the current thread. To compute the shared locations in a program, at each thread creation point we intersect the standard effect of the created thread with the future effect of the parent thread. If a location is in both effects, then the location is shared. Note that the future effect of the parent includes the effects of any threads that the parent creates later. Chapter 5 presents the sharing analysis of LOCKSMITH and the contextual effect system in detail.

For example, consider line 18 in Fig. 6.2. The spawned thread1 has standard effect $\{\&\mathsf{local}\}$. The parent thread by itself has no future effect, since it accesses no interesting variables. However, it spawns two child threads which themselves access count1 and count2. Therefore, the future effect of line 18 is $\{\&\mathsf{count1}, \&\mathsf{count2}\}$. Since $\{\&\mathsf{local1}\} \cap \{\&\mathsf{count1}, \&\mathsf{count2}\} = \emptyset$, there are no shared locations created by this fork. In particular, even though local was accessed in the past by the parent thread (line 17), our sharing analysis correctly determines all its accesses to be thread local.

On the other hand, consider line 19. Here the effect of the spawned thread2 is $\{\&\mathsf{count1}, \&\mathsf{count2}\}$, and the future effect at line 19 is also $\{\&\mathsf{count1}, \&\mathsf{count2}\}$. Thus we determine from this call that count1 and count2 are shared. Notice that here it was critical to include the effect of thread3 when computing the future effect of the parent, since the parent thread itself does not access anything interesting.

In our implementation, we also distinguish read and write effects, and only mark a location ρ as shared if at least one of the accesses to ρ in the intersected effects is a write.

The analysis just described only determines if a location is *ever* shared. It could be that a location starts off as thread local and only later becomes shared, meaning that its initial accesses need not be protected by a consistent lock, while subsequent ones do. For example, notice that &count1 in Fig. 6.2 becomes shared due to the thread creations at lines 19 and 20, since both thread2 and thread3 access it. So while the accesses at lines 33 and 26 (via line 42) must consider &count1 as shared, &count1 would not need to be considered shared if it were accessed at, say, line 18. We can use a simple dataflow analysis to distinguish these two cases, and thus avoid reporting a false alarm in the latter case. Section 6.10 presents the sharing analysis and this variant in more detail.

6.4 Lock state analysis

In the next phase, LOCKSMITH computes the state of each lock at every program point. To do this, we use the ACFG to compute the set of locks ℓ held before and after each statement.

In the ACFG in Fig. 6.3(b), we begin at the entry node by assuming all locks are released. In the subsequent discussion, we write A_i for the set of locks that are definitely acquired after statement i. Since statements 15–21 do not affect the set of locks held, we have $A_{15} = A_{16} = A_{18} = A_{19} = A_{20} = A_{21} = A_{\mathsf{Entry}} = \emptyset$.

We continue propagation for the control flow of the three created threads. Note that even if a lock is acquired at a fork point, it is released in the new thread, so we should not propagate the set of acquired locks along the child Fork edge. For thread1, we find simply that $A_{26} = \emptyset$. For thread2, we have $A_{32} = A_{33} = \{\&\mathsf{lock1}\}$, and $A_{34} = A_{35} = \emptyset$. And lastly for thread3, we have $A_{42} = A_{43} = A_8 = \emptyset$ and $A_6 = A_7 = \{\mathsf{lock}\}$. Notice that this last set contains the name of the formal parameter lock. When we perform correlation inference, discussed next, we will need to translate information about lock back into information about the actual arguments at the two call sites.

6.5 Correlation inference

The next phase of LOCKSMITH is correlation inference, which is the core race detection algorithm. For each shared variable, we intersect the sets of locks held at all its accesses. We call this the *guarded-by* set for that location, and if the set is empty, we report a possible data race. Chapter 3 presents the correlation inference system in detail.

We begin by generating initial *correlation constraints* at each $\mathsf{Acc}(\rho)$ node such that ρ may be shared according to the sharing analysis. Correlation constraints have the form $\rho \rhd \{\ell_1, \ldots, \ell_n\}$, meaning location ρ is accessed with locks ℓ_1 through ℓ_n held. We write C_n for the set of correlation constraints inferred for statement n.

The first access in the program, on line 17, yields no correlation constraints ($C_{18} = \emptyset$) because, as we discussed above, the sharing analysis determines the &local is not a shared variable. Similarly, $C_{26} = \emptyset$ because the only location that "flows to" y in the label

flow graph is &local, which is not shared. On line 33, on the other hand, we have an access to a shared variable, and so we initialize $C_{33} = \{\&\mathsf{count1} \rhd \{\&\mathsf{lock1}\}\}$, using the output of the lock state analysis to report which locks are held. Similarly, $C_{35} = \{\&\mathsf{count2} \rhd \emptyset\}$, since no locks are held at that access. Finally, $C_7 = \{\mathsf{count} \rhd \{\mathsf{lock}\}\}$. Here we determine count may be shared because at least one shared variable flows to it in the label flow graph.

Notice that this last correlation constraint is in terms of the local variables of function atomic_inc. Thus at each call to atomic_inc, we must instantiate this constraint in terms of the caller's variables. We use an iterative fixpoint algorithm to propagate correlations through the control-flow graph, instantiating as necessary until we reach the entry node of main. At this point, all correlation constraints are in terms of the names visible in the top-level scope, and so we can perform the set intersections to look for races. Note that, as is standard for label flow analysis, when we label a syntactic occurrence of malloc() or any other memory allocation or lock creation site, we treat that label as a top-level name.

We begin by propagating C_7 backwards, setting $C_6 = C_7$. Continuing the backwards propagation, we encounter two Call edges. For each call site i in the program, there exists a substitution S_i that maps the formal parameters to the actual parameters; this substitution is equivalent to a polymorphic type instantiation [132, 124]. For call site 4 we have $S_4 = [\mathsf{lock} \mapsto \&\mathsf{lock1}, \mathsf{count} \mapsto \&\mathsf{count1}]$. Then when we propagate the constraints from C_7 backwards across the edge annotated $(_4$, we apply S_4 to instantiate the constraints for the caller. In this case we set $C_{42} = S_4(\{\mathsf{count} \rhd \{\mathsf{lock}\}\}) = \{\&\mathsf{count1} \rhd \{\&\mathsf{lock1}\}\}$ and thus we have derived the correct correlation constraint inside of thread3. Similarly, when we propagate C_6 backwards across the edge annotated $(_5$, we find $C_{43} = \{\&\mathsf{count2} \rhd \{\&\mathsf{lock2}\}\}$.

We continue backwards propagation, and eventually push all the correlations we have mentioned so far back to the entry of main:

$\&\mathsf{count1} \rhd \{\&\mathsf{lock1}\}$	from line 33
$\&\mathsf{count2} \rhd \emptyset$	from line 35
$\&\mathsf{count1} \rhd \{\&\mathsf{lock1}\}$	from the call on line 42
$\&\mathsf{count2} \rhd \{\&\mathsf{lock2}\}$	from the call on line 43

(Note that there are substitutions for the calls indexed by 1–3, but they do not rename $\&\mathsf{count}i$ or $\&\mathsf{lock}i$, since those are global names.) We now intersect the lock sets from each correlation, and find that &count1 is consistently correlated with (or guarded by) lock1, whereas &count2 is not consistently correlated with a lock. We then report a race on &count2.

One important detail we have omitted is that when we propagate correlation constraints back to callers, we need to interpret them with respect to the "closure" of the label flow graph. For example, given a constraint $\mathsf{x} \rhd \{\&\mathsf{lock}\}$, if &y flows to x in the label flow graph, then we also derive $\&\mathsf{y} \rhd \{\&\mathsf{lock}\}$. More information about the closure computation can be found elsewhere [125].

Propagating correlation constraints backwards through the ACFG also helps to improve the reporting of potential data races. In our implementation, we also associate a *program path*, consisting of a sequence of file and line numbers, with each correlation

constraint. When we generate an initial constraint at an access in the ACFG, the associated path contains just the syntactic location of that access. Whenever we propagate a constraint backwards across a Call edge, we prepend the file and line number of the call to the path. In this way, when the correlation constraints reach the main, they describe a path of function calls from main to the access site, essentially capturing a stack trace of a thread at the point of a potentially racing access, and developers can use these paths to help understand error messages.

Section 6.9.3 presents the algorithm for solving correlation constraints and inferring all correlations in the program. Since correlation analysis is an iterative fixpoint computation, in which we iteratively convert local names to their global equivalents, we compute correlations using the same framework we used to infer the state of locks.

6.6 Linearity and escape checking

The constraint generation phase also creates *linearity constraints*, which we use to ensure that a static lock name ℓ used in the analysis never represents two or more run-time locks that are simultaneously live. Without this assurance, we could not model lock acquire and release precisely. That is, suppose during the lock state analysis we encounter a node $\mathsf{Acq}(\ell)$. If ℓ is non-linear, it may represent more than one lock, and thus we do not know which one will actually be acquired by this statement. On the other hand, if ℓ is linear, then it represents exactly one location at run time, and hence after $\mathsf{Acq}(\ell)$ we may assume that ℓ is acquired.

Lock ℓ could be non-linear for a number of reasons. Consider, for example, a linked list data structure where each node of the linked list contains a lock, meant to guard access to the data reachable from that node. Standard label flow analysis will label each element in such a recursive structure with the same name ρ whose pointed-to memory is a record containing some lock ℓ. Thus, ρ statically represents arbitrarily many run-time locations, and consequently ℓ represents arbitrarily many locks. With such a representation, a naive analysis would not complain about a program that acquires a lock contained in one list element but then accesses data present in other elements.

To be conservative, LOCKSMITH treats locks ℓ such as these as *non-linear*, with the consequence that nodes $\mathsf{Acq}(\ell)$ and $\mathsf{Rel}(\ell)$ of such non-linear ℓ are ignored. This approach solves the problem of missing potential races, but is more likely to generate false positives, e.g., when there is an access that is actually guarded by ℓ at run time. LOCKSMITH addresses this issue by using user-specified *existential types* to allow locks in data structures to sometimes be linear, and includes an escape checking phase to ensure existential types are used correctly. Chapter 4 presents a generic analysis that uses existential types to analyze data structures with precision, and its application in LOCKSMITH.

6.7 Results

Fig. 6.4 summarizes the results of running LOCKSMITH on a set of benchmarks varying in size, complexity and coding style. The results shown correspond to the default

	Benchmark	Size (LOC)	Time (sec)	Warnings	Unguarded	Races
POSIX thread programs	aget	1,914	0.85	62	31	31
	ctrace	2,212	0.59	10	9	2
	engine	2,608	0.88	7	0	0
	knot	1,985	0.78	12	8	8
	pfscan	1,948	0.46	6	0	0
	smtprc	8,624	5.37	46	1	1
Linux device drivers	3c501	17,443	9.18	15	5	4
	eql	16,568	21.38	35	0	0
	hp100	20,370	143.23	14	9	8
	plip	19,141	19.14	42	11	11
	sis900	20,428	71.03	6	0	0
	slip	22,693	16.99	3	0	0
	sundance	19,951	106.79	5	1	1
	synclink	24,691	1521.07	139	2	0
	wavelan	20,099	19.70	10	1	1

Figure 6.4: Benchmarks

configuration for all LOCKSMITH analyses, in particular: context-sensitive, field-sensitive label flow analysis, with lazy field propagation and no conflation under `void*`; flow- and context-sensitive sharing analysis; and context-sensitive lock state analysis and correlation inference.

The first part of the table presents a set of applications that use POSIX threads, whereas the second part of the table presents the results for a set of network drivers taken from the Linux kernel. The first column gives the benchmark name and the second column presents the number of preprocessed and merged lines of code for every benchmark. We used the CIL merger to combine all the code for every benchmark into a single C file, also removing comments and redundant declarations. The next column lists the running time for LOCKSMITH. Experiments in this paper were performed on a dual-core, 3GHz Pentium D CPU with 4GB of physical memory. All times reported are the median of three runs. The fourth column lists the total number of warnings (shared locations that are not protected by any lock) that LOCKSMITH reports. The next column lists how many of those warnings correspond to cases in which shared memory locations are not protected by any lock. The last column lists how many of those we believe correspond to races. Note that in some cases there is a difference between the unguarded and races columns, where an unguarded location is not a race. These are caused by the use of other synchronization constructs, such as pthread_join, semaphores, or inline atomic assembly instructions, which LOCKSMITH does not model.

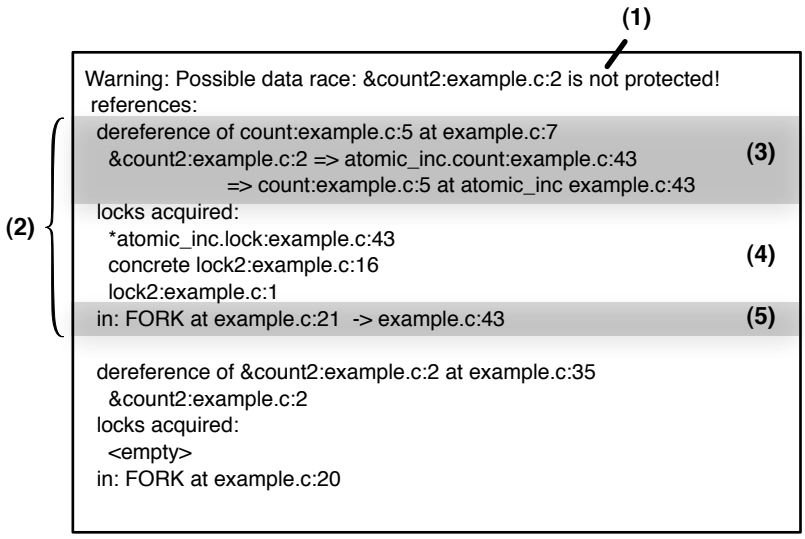


Figure 6.5: Sample LOCKSMITH warning. Highlighting and markers added for expository purposes.

6.7.1 Warnings

Each warning produced by LOCKSMITH contains information to explain the potential data race to the user. Fig. 6.5 shows a sample warning from the analysis of the example program shown in Fig. 6.2, stored in file example.c. The actual output of LOCKSMITH is pure text; here we have added some highlighting and markers (referred to below) for expository purposes.

LOCKSMITH issues one warning per allocation site that is shared but inconsistently (or un-)protected. In this example, the suspect allocation site is the contents of the global variable count2, declared on line 2 of file example.c (1). After reporting the allocation site, LOCKSMITH then describes each syntactic access of the shared location.

The text block indicated by (2) describes the first access site at line 6, accessing variable count which is declared at line 4. The other text block shown in the error report (each block is separated by a newline) corresponds to a different access site. Within a block, LOCKSMITH first describes why the expression that was actually accessed aliases the shared location (3). In this case, the shared location &count2 "flows to" (indicated by $\Rightarrow$) the argument count passed to the function call of atomic_inc at line 43 (written as atomic_inc.count), in the label flow graph. That, in turn, flows to the formal argument count of the function, declared at line 4, due to the invocation at line 43. If there is more than one such path in the label flow graph, we list the shortest one.

Next, LOCKSMITH prints the set of acquired locks at the access (4). We specially designate *concrete* lock labels, which correspond to variables initialized by pthread_mutex_init(), from aliases of those variables. Aliases are included in the error report to potentially help the programmer locate the relevant pthread_mutex_lock() and pthread_mutex_unlock() statements. In this case, the second lock listed is a concrete lock created at line 15 and named lock2, after the variable that stores the result of pthread_mutex_init(). The global

variable lock2 itself, listed third, is a different lock that aliases the concrete lock created at line 15. There is also an additional alias listed first, the argument ∗lock of the call to function atomic_inc at line 43 (denoted as ∗atomic_inc.lock : example.c : 43). We do not list aliasing paths for the lock sets, because we list all the aliases, and also printing the paths between them would only add confusion by replicating the lock aliases' names many times.

Finally, LOCKSMITH gives stack traces leading up to the access site (5). Each trace starts with the thread creation point (either a call to pthread_create() or the start of main()), followed by a list of function invocations. For this access site, the thread that might perform the access is created at line 20 and then makes a call at example.c : 43 to the function that contains the access. We generate this information during correlation inference, as we propagate information from the access point backwards through the ACFG (Section 6.9.3).

The other dereference listed has the same structure. Notice that the intersection of the acquired lock sets of the two sites is empty, triggering the warning. In this case, the warning corresponds to a true race, caused by the unguarded write to count2 at line 35, listed as the second dereference in the warning message. To check whether a warning corresponds to an actual race, the programmer has to verify that the listed accesses might actually happen simultaneously, including the case where a single access occurs simultaneously in several threads. Also, the programmer would have to verify that the aliasing listed indeed occurs during the program execution, and is not simply an artifact of imprecision in the points-to analysis. Moreover, the programmer must check whether the set of acquired locks contains additional locks that LOCKSMITH cannot verify are acquired. Last, but not least, the programmer needs to validate that the location listed in the warning is in fact shared among different threads.

6.7.2 Races

We found races in many of the benchmarks. In knot, all of the races are on counter variables always accessed without locks held. These variables are used to generate usage statistics, which races could render inaccurate, but this would not affect the behavior of knot. In aget, most of the races are due to an unprotected global array of shared information. The programmer intended for each element of the array to be thread-local, but a race on an unrelated memory location in the signal handling code can trigger erroneous computation of array indexes, causing races that may trigger a program crash. The remaining two races are due to unprotected writes to global variables, which might cause a progress bar to be printed incorrectly. In ctrace, two global flag variables can be read and set at the same time, causing them to have erroneous values. In smtprc, a variable containing the number of threads is set in two different threads without synchronization. This can result in not counting some threads, which in turn may cause the main thread to not wait for all child threads at the end of the program. The result is a memory leak, but in this case it does not cause erroneous behavior since it occurs at the end of the program. In most of the Linux drivers, the races correspond to integer counters or flags, but do not correspond to bugs that could crash the program, as there is usually a subsequent check that restores the correct value to a variable. The rest of the warnings for the Linux drivers

can potentially cause data corruption, although we could not verify that any can cause the kernel to crash.

6.7.3 False positives

The majority of false positives are caused by over-approximation in the sharing analysis. The primary reason for this is conservativism in the label flow (points-to) analysis, which can cause many thread-local locations to be spuriously conflated with shared locations. Since thread-local memory need not be, and usually is not, protected by locks, this causes many false warnings of races on those locations. Overly-conservative aliasing has several main causes: structural subtyping where the same data is cast between two different types (e.g. different structs that share a common prefix), `asm` blocks, casting to or from numerical types, and pointer arithmetic, in order of significance. One approach to better handling structural subtyping may be adapting physical subtyping [145] to LOCKSMITH. Currently, when a `struct` type is cast to a different `struct` type, LOCKSMITH does not compute field offsets to match individual fields, but rather conservatively assumes that all labels of one type could alias all labels of the other.

The second largest source of false positives in the benchmarks is the flow-sensitivity of the sharedness property, in more detail than our current flow-sensitive sharing propagation can capture. Specifically, any time that a memory location might be accessed by two threads, we consider it shared immediately when the second thread is created. However, in many cases thread-local memory is first initialized locally, and then becomes shared indirectly, e.g., via an assignment to a global or otherwise shared variable. We eliminate some false positives using a simple intra-procedural uniqueness analysis—a location via a unique pointer as determined by this analysis is surely not shared—but it is too weak for many other situations.

6.8 Implementation

We present the implementation of the LOCKSMITH analyses in detail, starting from its core analysis engine (Section 6.9), with separate consideration of lock state analysis (Section 6.9.3), correlation inference (Section 6.9.3), the sharing analysis (Section 6.10), techniques for effectively modeling C `struct` and `void*` types (Section 6.11), and extensions to enable context-sensitive analysis (Section 6.12). For many of LOCKSMITH's analysis components, we implemented several possible algorithms, and measured the algorithms' effects on the precision and efficiency of LOCKSMITH. By combining a careful exposition of LOCKSMITH's inner workings with such detailed measurements, we have endeavored to provide useful data to inform further developments in the static analysis of C programs (multi-threaded or otherwise).

$$
\begin{array}{rcl}
e & ::= & x \mid v \mid e\ e \mid \mathsf{if0}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \\
& \mid & (e, e) \mid e.j \mid \mathsf{ref}\ e \mid\ !\, e \mid e := e \\
& \mid & \mathsf{newlock} \mid \mathsf{acquire}\ e \mid \mathsf{release}\ e \mid \mathsf{fork}\ e \\
v & ::= & n \mid \lambda x : t.e \mid (v_1, v_2) \\
t & ::= & \mathit{int} \mid t \times t \mid t \to t \mid \mathit{ref}(t) \mid \mathit{lock}
\end{array}
$$

Figure 6.6: Source language

6.9 Labeling and constraint generation

We present LOCKSMITH's key algorithms on the language in Figure 6.6. This language extends the standard lambda calculus, which consists of variables x, functions $\lambda x : t.e$ (where the argument x has type t), and function application $e\ e$. To model conditional control flow, we add integers n and the conditional form $\mathsf{if0}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$, which evaluates to e_1 if e_0 evaluates to 0, and to e_2 otherwise. To model structured data (i.e., C structs) we introduce pairs (e, e) along with projection $e.j$. The latter form returns the jth element of the pair ($j \in 1, 2$). We model pointers and dynamic allocation as using references. The expression $\mathsf{ref}\ e$ allocates a fresh memory location m, initializing it with the result of evaluating e and returning m. The expression $!\, e$ reads the value stored in memory location e, and the expression $e_1 := e_2$ updates the value in location e_1 with the result of evaluating e_2.

We model locks with three expressions: $\mathsf{newlock}$ dynamically allocates and returns a new lock, and $\mathsf{acquire}\ e$ and $\mathsf{release}\ e$ acquire and release, respectively, lock e. Our language also includes the expression $\mathsf{fork}\ e$, which creates a new thread that evaluates e in parallel with the current thread. The expression $\mathsf{fork}\ e$ is asynchronous, i.e., it returns to the parent immediately without waiting for the child thread to complete.

Source language types t include the integer type int, pair types $t \times t$, function types $t \to t$, reference (or pointer) types $\mathit{ref}(t)$, and the type lock of locks. Note that our source language is monomorphically typed and that, in a function $\lambda x : t.e$, the type t of the formal argument x is supplied by the programmer. This matches C, which includes programmer annotations on formal arguments. If we wish to apply LOCKSMITH to a language without these annotations, we can always apply standard type inference to determine such types as a preliminary step.

6.9.1 Labeling and constraint generation

As discussed in section 6.1, the first stage of LOCKSMITH is *labeling and constraint generation*, which produces both label flow constraints, to model memory locations and locks, and abstract control-flow constraints, to model control-flow. We specify the constraint generation phase using type inference rules. We discuss the label flow constraints only briefly here, and refer the reader to prior work [109, 40, 132, 87, 82, 124], as well as Chapter 3, for more details. In our implementation, we use BANSHEE [88], a set-constraint solving engine, to represent and solve label flow constraints. For the time being, we present a purely monomorphic (context-insensitive) analysis; Section 6.12 dis-

$$
\begin{array}{rcl}
\tau & ::= & \mathit{int} \mid \tau \times \tau \mid (\tau, \phi) \rightarrow^{\ell} (\tau, \phi) \mid \mathit{ref}^{\rho}(\tau) \mid \mathit{lock}^{\ell} \\
C & ::= & C \cup C \mid \tau \leq \tau \mid \rho \leq \rho \mid \ell \leq \ell \mid \phi \leq \phi \mid \phi : \kappa \\
\kappa & ::= & \mathsf{Acc}(\rho) \mid \mathsf{NewL}(\ell) \mid \mathsf{Acq}(\ell) \mid \mathsf{Rel}(\ell) \\
 & \mid & \mathsf{Fork} \mid \mathsf{Call}(\ell, \phi) \mid \mathsf{Ret}(\ell, \phi)
\end{array}
$$

$$
\begin{array}{rcl}
\rho & \in & \textit{abstract locations} \\
\ell & \in & \textit{abstract locks} \\
\phi & \in & \textit{abstract statement labels}
\end{array}
$$

$$
\begin{array}{rcll}
\langle\langle \mathit{int} \rangle\rangle & = & \mathit{int} & \\
\langle\langle t_1 \times t_2 \rangle\rangle & = & \langle\langle t_1 \rangle\rangle \times \langle\langle t_2 \rangle\rangle & \\
\langle\langle t_1 \rightarrow t_1 \rangle\rangle & = & (\langle\langle t_1 \rangle\rangle, \phi_1) \rightarrow^{\ell} (\langle\langle t_2 \rangle\rangle, \phi_2) & \phi_1, \phi_2, \ell \textit{ fresh} \\
\langle\langle \mathit{ref}(t) \rangle\rangle & = & \mathit{ref}^{\rho}(\langle\langle t \rangle\rangle) & \rho \textit{ fresh} \\
\langle\langle \mathit{lock} \rangle\rangle & = & \mathit{lock}^{\ell} & \ell \textit{ fresh}
\end{array}
$$

$\langle\langle \tau \rangle\rangle = \tau'$ where τ' is τ with fresh ρ, ℓ, ϕ's, as above

$C \vdash \phi \leq (\phi' : \kappa) \equiv C \vdash \phi \leq \phi'$ and $C \vdash \phi' : \kappa$ and ϕ' *fresh*

Figure 6.7: Auxiliary definitions

cusses context-sensitivity.

We extend source languages types t to *labeled types* τ, defined by the grammar at the top of Figure 6.7. The type grammar is mostly the same as before, with two main changes. First, reference and lock types now have the forms $\mathit{ref}^{\rho}(\tau)$ and lock^{ℓ}, where ρ is an abstract location and ℓ is an abstract lock. As mentioned in the last section, each ρ and ℓ stands for one or more concrete, run-time locations. Second, function types now have the form $(\tau, \phi) \rightarrow^{\ell} (\tau', \phi')$, where τ and τ' are the domain and range type, and ℓ is a lock, discussed further below. In this function type, ϕ and ϕ' are *statement labels* that represent the entry and exit node of the function. We will usually say *statement* ϕ instead of *statement label* ϕ when the distinction is made clear by the use of a ϕ variable.

During type inference, our type rules generate *constraints* C, including *flow constraints* of the form $\tau \leq \tau'$, indicating a value of type τ *flows to* a position of type τ'. Flow constraints among types are ultimately reduced to flow constraints $\rho \leq \rho'$ and $\ell \leq \ell'$ among locations and locks, respectively. When we draw a set of label flow constraints as a graph, as e.g. in Figure 6.3(a), ρ's and ℓ's form the nodes, and each constraint $x \leq y$ is drawn as a directed edge from x to y.

We also generate two kinds of constraints that, put together, define the ACFG. Whenever statement ϕ occurs immediately before statement ϕ', our type system generates a constraint $\phi \leq \phi'$. As above, we drawn such a constraint as an edge from ϕ to ϕ'. We generate *kind constraints* of the form $\phi : \kappa$ to indicate that statement ϕ has *kind* κ, where the kind indicates the statement label's relevant behavior, as described in Section 6.2. Note that our type rules assign at most one kind to each statement label, and thus we showed only the kinds of statement labels in Figure 6.3(b). Statement labels with no kind, including join points and function entries and exits, have no interesting effect.

$$[\textsc{Var}]\frac{}{C;\phi;\Gamma,x:\tau\vdash x:\tau;\phi} \qquad [\textsc{Int}]\frac{}{C;\phi;\Gamma\vdash n:\mathit{int};\phi}$$

$$[\textsc{Pair}]\frac{C;\phi;\Gamma\vdash e_1:\tau_1;\phi_1 \quad C;\phi_1;\Gamma\vdash e_2:\tau_2;\phi_2}{C;\phi;\Gamma\vdash(e_1,e_2):\tau_1\times\tau_2;\phi_2} \qquad [\textsc{Proj}]\frac{C;\phi;\Gamma\vdash e:\tau_1\times\tau_2;\phi' \quad j\in 1,2}{C;\phi;\Gamma\vdash e.j:\tau_j;\phi'}$$

$$[\textsc{Ref}]\frac{C;\phi;\Gamma\vdash e:\tau;\phi' \quad \rho\ \mathit{fresh}}{C;\phi;\Gamma\vdash \mathsf{ref}\ e:\mathit{ref}^{\rho}(\tau);\phi'} \qquad [\textsc{Deref}]\frac{C;\phi;\Gamma\vdash e_1:\mathit{ref}^{\rho}(\tau);\phi_1 \quad C\vdash\phi_1\le(\phi':\mathsf{Acc}(\rho))}{C;\phi;\Gamma\vdash !\,e_1:\tau;\phi'}$$

$$[\textsc{Fork}]\frac{C;\phi';\Gamma\vdash e:\tau;\phi'' \quad C\vdash\phi\le(\phi':\mathsf{Fork})}{C;\phi;\Gamma\vdash\mathsf{fork}\ e:\mathit{int};\phi} \qquad [\textsc{Newlock}]\frac{C\vdash\phi\le(\phi':\mathsf{NewL}(\ell)) \quad \ell\ \mathit{fresh}}{C;\phi;\Gamma\vdash\mathsf{newlock}:\mathit{lock}^{\ell};\phi'}$$

$$[\textsc{Acquire}]\frac{C;\phi;\Gamma\vdash e:\mathit{lock}^{\ell};\phi' \quad C\vdash\phi'\le(\phi'':\mathsf{Acq}(\ell))}{C;\phi;\Gamma\vdash\mathsf{acquire}\ e:\mathit{int};\phi''} \qquad [\textsc{Release}]\frac{C;\phi;\Gamma\vdash e:\mathit{lock}^{\ell};\phi' \quad C\vdash\phi'\le(\phi'':\mathsf{Rel}(\ell))}{C;\phi;\Gamma\vdash\mathsf{release}\ e:\mathit{int};\phi''}$$

$$[\textsc{Assign}]\frac{\begin{array}{c}C;\phi;\Gamma\vdash e_1:\mathit{ref}^{\rho}(\tau_1);\phi_1 \quad C;\phi_1;\Gamma\vdash e_2:\tau_2;\phi_2\\ C\vdash\tau_2\le\tau_1 \quad C\vdash\phi_2\le(\phi':\mathsf{Acc}(\rho))\end{array}}{C;\phi;\Gamma\vdash e_1:=e_2:\tau_2;\phi'} \qquad [\textsc{Lam}]\frac{\begin{array}{c}\tau=\langle\langle t\rangle\rangle \quad \phi_\lambda\ \mathit{fresh}\\ C;\phi_\lambda;\Gamma,x:\tau\vdash e:\tau';\phi'_\lambda\\ \ell=\{\text{locks } e \text{ may access}\}\end{array}}{C;\phi;\Gamma\vdash\lambda x:t.e:(\tau,\phi_\lambda)\to^{\ell}(\tau',\phi'_\lambda);\phi}$$

$$[\textsc{App}]\frac{\begin{array}{c}C;\phi;\Gamma\vdash e_1:(\tau,\phi_\lambda)\to^{\ell}(\tau',\phi'_\lambda);\phi_1 \quad \phi_1;\Gamma\vdash e_2:\tau_2;\phi_2 \quad C\vdash\tau_2\le\tau\\ C\vdash\phi_2\le(\phi_3:\mathsf{Call}(\ell,\phi')) \quad C\vdash\phi_3\le\phi_\lambda \quad C\vdash\phi'_\lambda\le(\phi':\mathsf{Ret}(\ell,\phi_3))\end{array}}{C;\phi;\Gamma\vdash e_1\ e_2:\tau';\phi'}$$

$$[\textsc{Cond}]\frac{\begin{array}{c}C;\phi;\Gamma\vdash e_0:\mathit{int};\phi_0 \quad C;\phi_0;\Gamma\vdash e_1:\tau_1;\phi_1 \quad C;\phi_0;\Gamma\vdash e_2:\tau_2;\phi_2\\ \tau=\langle\langle\tau_1\rangle\rangle \quad C\vdash\tau_1\le\tau \quad C\vdash\tau_2\le\tau \quad C\vdash\phi_1\le\phi' \quad C\vdash\phi_1\le\phi' \quad \phi'\ \mathit{fresh}\end{array}}{C;\phi;\Gamma\vdash\mathsf{if0}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2:\tau;\phi'}$$

Figure 6.8: Type inference rules

The bottom half of Figure 6.7 defines some useful shorthands. The notation $\langle\langle\cdot\rangle\rangle$ denotes a function that takes a either standard type or a labeled type and returns a new labeled type with the same shape but with fresh abstract locations, locks, and statement labels at each relevant position. By *fresh* we mean a variable that has not been introduced elsewhere in the typing derivation. We also use the abbreviation $C\vdash\phi\le(\phi':\kappa)$, which stands for $C\vdash\phi\le\phi'$, $C\vdash\phi':\kappa$, and ϕ' *fresh*. These three operations often go together in our type inference rules.

Figure 6.8 gives type inference rules that prove judgments of the form $C;\phi;\Gamma\vdash e:\tau;\phi'$, meaning under constraints C and type environment Γ (a mapping from variable names to labeled types), if the preceding statement label is ϕ (the *input statement label*), then expression e has type τ and has the behavior described by statement ϕ' (the *output statement label*). In these rules, the notation $C\vdash C'$ means that C must contain the constraints C'. Viewing these rules as defining a constraint generation algorithm, we

interpret this judgment as *generating* the constraint C' and adding it to C.

We discuss the rules briefly. [VAR] and [INT] are standard and yield no constraints. The output statement labels of these rules are the same as the input statement labels, since accessing a variable or referring to an integer has no effect.

In [PAIR], we type e_1 with the input statement ϕ for the whole expression, yielding output statement ϕ_1. We then type e_2 starting in ϕ_1 and yielding ϕ_2, the output statement label for the whole expression. Notice we assume a left-to-right order of evaluation. In [PROJ], we type check the subexpression e, and the output statement label of the whole expression is the output of e.

[REF] types memory allocation, associating a fresh abstract location ρ with the newly-created updateable reference. Notice that this rule associates ρ with a syntactic occurrence of ref, but if that ref is in a function, it may be executed multiple times. Hence the single ρ chosen by [REF] may stand for multiple run-time locations.

[DEREF] is the first rule to actually introduce a new statement label into the abstract control-flow graph. We type e_1, yielding a pointer to location ρ and an output statement ϕ_1. We then add a new statement ϕ' to the control-flow graph, occurring immediately after ϕ_1, and give ϕ' the kind $\mathsf{Acc}(\rho)$ to indicate the dereference. Statement ϕ' is the output of the whole expression. [ASSIGN] is similar, but also requires the type τ_2 of e_2 be a subtype of the type τ_1 of data referenced by the pointer e_1.

[NEWLOCK] types a lock allocation, assigning it a fresh abstract lock ℓ, similarly to [REF]. [ACQUIRE] and [RELEASE] both require that their argument be a lock, and both return some int (in C these functions typically return `void`). All three of these rules introduce a new statement label of the appropriate kind into the control-flow graph immediately after the output statement label of the last subexpression.

In [FORK], the control-flow is somewhat different than the other rules, to match the asynchronous nature of fork. At run time, the expression e is evaluated in a new thread. Hence we introduce a new statement ϕ' with the special kind Fork, to mark thread creation, and type e with input statement ϕ'. We sequence ϕ' immediately after ϕ, since that is their order in the control flow. The output statement label of the fork expression as a whole is the same as the input statement ϕ, since no state in the parent changes after the fork.

In [COND], we sequence the subexpressions as expected. We type both e_1 and e_2 with input statement ϕ_0, since either may occur immediately after e_0 is evaluated. We also create a fresh statement ϕ' representing the join point of the condition, and add appropriate constraints to C. Since the join point has no effect on the program state, we do not associate a kind with it. We also join the types τ_1 and τ_2 of the two branches, constraining them to flow to a type τ, which has the same shape as τ_1 but has fresh locations and locks. Note that for the constraints in this rule to be satisfied, τ_2 must have the same shape as τ_1, and so we could equivalently have written $\tau = \langle\langle \tau_2 \rangle\rangle$.

[LAM] type checks the function body e in an environment with x bound to τ, which is the standard type t annotated with fresh locations and locks. We create a new statement ϕ_λ to represent the function entry, and use that as the input statement label when typing the function body e. We place ϕ_λ and ϕ'_λ, the output statement label after e has been evaluated, in the type of the function. We also add an abstract lock ℓ to the function type to represent the function's *lock effect*, which is the set of locks that may be acquired

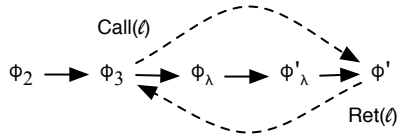
$$
\begin{array}{rcl}
C \cup \{int \le int\} & \Rightarrow & C \\
C \cup \{ref^{\rho}(\tau) \le ref^{\rho'}(\tau')\} & \Rightarrow & C \cup \{\rho \le \rho', \tau \le \tau', \tau' \le \tau\} \\
C \cup \{lock^{\ell} \le lock^{\ell'}\} & \Rightarrow & C \cup \{\ell \le \ell'\} \\
C \cup \{\tau_1 \times \tau_2 \le \tau_1' \times \tau_2'\} & \Rightarrow & C \cup \{\tau_1 \le \tau_1', \tau_2 \le \tau_2'\} \\
C \cup \{(\tau_1, \phi_1) \to (\tau_1', \phi_1') \le (\tau_2, \phi_2) \to (\tau_2', \phi_2')\} & \Rightarrow & C \cup \{\tau_2 \le \tau_1, \tau_1' \le \tau_2'\}
\end{array}
$$

$$
\begin{array}{rcl}
C \cup \{\rho \le \rho', \rho' \le \rho''\} & \cup \Rightarrow & \{\rho \le \rho''\} \\
C \cup \{\ell \le \ell', \ell' \le \ell''\} & \cup \Rightarrow & \{\ell \le \ell''\}
\end{array}
$$

Figure 6.9: Label flow constraint rewriting rules

or released when the function executes. For each lock ℓ' that either e acquires or releases directly or that appears on the arrow of a function called in e, a separate effect analysis (not shown) generates a constraint $\ell' \le \ell$. Then during constraint resolution, we compute the set of locks that flow to ℓ to compute the lock effect. We discuss the use of lock effects in Section 6.5. The output statement label for the expression as a whole is the same as the input statement label, since defining a function has no effect.

Finally, [APP] requires that e_2's type be a subtype of e_1's domain type. We also add the appropriate statement labels to the control-flow graph. Statement ϕ_2 is the output of e_2, and ϕ_λ is the entry node for the function. Thus clearly we need to add control flow from ϕ_2 to ϕ_λ. Moreover, ϕ_λ' is the output statement label of the function body, and that should be the last statement label in the function application as a whole. However, rather than directly inserting ϕ_λ and ϕ_λ' in the control-flow graph, we introduce two intermediate statement labels, ϕ_3 just before the call, and ϕ' just after. Statement ϕ_3 has kind $\mathsf{Call}(\ell, \phi')$, and statement ϕ' has kind $\mathsf{Ret}(\ell, \phi_3)$. Pictorially, the control-flow graph looks like the following, where the ϕ's in the kinds of the Call and Ret nodes are drawn with dashed lines:



Using this structure, we can propagate certain dataflow facts "around" functions—i.e., directly from Call to Ret, rather than through the function body—thereby improving precision and gaining some speed up. In particular, we use this for our lock state computation (Section 6.4).

6.9.2 Label flow constraint resolution

After generating constraints using the rules in Figure 6.8, we can then solve the constraints to compute various information about the analyzed program. We use *flow constraints* to answer questions about which locations and locks are used by various statements in the program, i.e., to perform a label flow analysis. These constraints have the form $c \le c'$ where c and c' are either locations ρ, locks ℓ, or types τ.

Similarly to the system in Chapter 3, we apply the rewriting rules in Figure 6.9 to translate the constraints to a simpler, solved form. The first group of rewriting rules operate on constraints of the form $\tau \leq \tau'$. These rules are standard structural subtyping rules, matching the shapes of the left- and right-hand sides of the constraints and then propagating subtyping to the components in the usual way (e.g., invariant for references and contravariant for function domains) [121]. We will assume that the source program is type correct with respect to the standard types, so that these rewriting rules will never encounter a constraint they cannot reduce further, i.e., in the constraint $\tau \leq \tau'$, the types τ and τ' will always be the same modulo abstract locations and locks.

After applying these rewriting rules, we are left with constraints $\rho \leq \rho'$ and $\ell \leq \ell'$. The remaining rewrite rules add any transitively-implied constraints. Here the notation $C \cup \Rightarrow \ C'$ means we add the constraints C' to C. We define $Sol(C)$ to be the set of constraints computed by exhaustively applying the rules in Figure 6.9 to C. We can then define

$$\begin{aligned} Flow(C, \rho) &= \{\rho' \mid \rho' \leq \rho \in Sol(C)\} \\ Flow(C, \ell) &= \{\ell' \mid \ell' \leq \ell \in Sol(C)\} \end{aligned}$$

In other words, $Flow(C, \rho)$ is the set of abstract locations that *flow to* ρ in the constraints C, and similarly for $Flow(C, \ell)$. LOCKSMITH uses BANSHEE to solve label flow constraints and compute the $Flow(C, \rho)$ sets. We can use this information to answer questions about locations and locks in the program. For example, given a dereference site $!\, e$, if type inference assigns e the type $ref^{\rho}(int)$ and ref e' the type $ref^{\rho'}(int)$, then if it is possible for e to evaluate to ref e', then $\rho' \in Flow(C, \rho)$. In other words, the set $Flow(C, \rho)$ conservatively models the set of locations that may flow to the reference annotated with ρ.

Consider the example of Section 6.1. In function thread1() (lines 23–28) the argument a corresponds to a variable with type $ref^{\rho_{\&a}}(ref^{\rho_a}(int))$ in this language, ignoring for now the special `void*` type. (We follow the convention that the location name is subscripted by the program variable that names the location.) Similarly, the local variable y in f() corresponds to a variable with type $ref^{\rho_{\&y}}(ref^{\rho_y}(int))$. (Because in C variables can be l-values, we consider all variables to be references to the corresponding type, adding an extra level of reference that is implicit in the C program.) In C, variable names are implicitly dereferenced when they occur in a read context. For example, the assignment to y in thread1() (line 24) can be written as y = a, where the occurrence of y at the left hand side of the assignment denotes the location y whereas the occurrence of a at the right hand side denotes its contents. In our formal language, that assignment corresponds to $y := !\, a$. Typing this assignment with [ASSIGN] and [DEREF] creates the flow constraint $ref^{\rho_a}(int) \leq ref^{\rho_y}(int)$. Then, the second and first rewriting rules in Figure 6.9 solve the constraint reducing it to $\rho_a \leq \rho_y$, and thus $\rho_a \in Flow(C, \rho_y)$.

Note that the constraints in this section are monomorphic and do not include the $(i$ and $)i$ edges we introduced in Section 6.1 for context-sensitivity. We will discuss how to incorporate context-sensitivity into this system in Section 6.12.

ϕ kind	$Acq_{out}(\phi)$
$\mathsf{Acc}(\rho)$	$Acq_{in}(\phi)$
Fork	$\emptyset$
$\mathsf{NewL}(\ell)$	$Acq_{in}(\phi)$
$\mathsf{Acq}(\ell)$	$Acq_{in}(\phi) \cup Flow(C, \ell)$

ϕ kind	$Acq_{out}(\phi)$
$\mathsf{Rel}(\ell)$	$Acq_{in}(\phi) \setminus Flow(C, \ell)$
$\mathsf{Call}(\ell, \phi')$	$Acq_{in}(\phi) \cap Flow(C, \ell)$ $Split(\phi') = Acq_{in}(\phi) \setminus Flow(C, \ell)$
$\mathsf{Ret}(\ell, \phi')$	$Acq_{in}(\phi) \cup Split(\phi)$

Figure 6.10: Transfer functions for lock state inference

6.9.3 Data flow analysis with the abstract control flow graph

LOCKSMITH uses a generic, mostly-standard dataflow analysis engine to compute per-program point information, such as which locks are held, by propagating dataflow facts through the ACFG. To construct a dataflow analysis, the programmer specifies the following characteristics of the target analysis [3]:

- The direction of the analysis (forwards or backwards)
- The type of the dataflow facts to propagate
- Initial dataflow facts at the program entry, and at each statement label
- A merge function to join dataflow facts
- Transfer functions for each kind of statement label

In the remainder of this section, we discuss two such dataflow analyses used by LOCKSMITH—lock state analysis and correlation inference—and then compare the performance of various strategies for implementing the fixpoint computation.

Lock State: Forwards dataflow

LOCKSMITH's lock state analysis is a forwards dataflow analysis, where the sets of dataflow facts are the set of acquired locks. The set of held locks is initially empty for all statement labels, and the merge function is set intersection. Figure 6.10 lists the transfer functions for each kind of statement label. $\mathsf{Acc}(\rho)$ and $\mathsf{NewL}(\ell)$ do not alter the lock state, since neither acquires or releases a lock. The transfer function for Fork always returns the empty set of locks, as every new thread starts with all locks released. (Recall from Figure 6.8 that the first statement label in a thread has kind Fork.) The transfer functions for $\mathsf{Acq}(\ell)$ and $\mathsf{Rel}(\ell)$ add and remove, respectively, $Flow(C, \ell)$ from the lock state. This latter set includes ℓ and all its aliases. The separate linearity check (mentioned in Section 6.6) ensures this set contains only one run-time lock. In our implementation, we also signal a warning at an attempt to acquire or release a lock that is already acquired or released, respectively.

The transfer function for $\mathsf{Call}(\ell, \phi')$ partitions the acquired locks into two non-overlapping sets. The transfer function sets the output set of acquired locks to be the

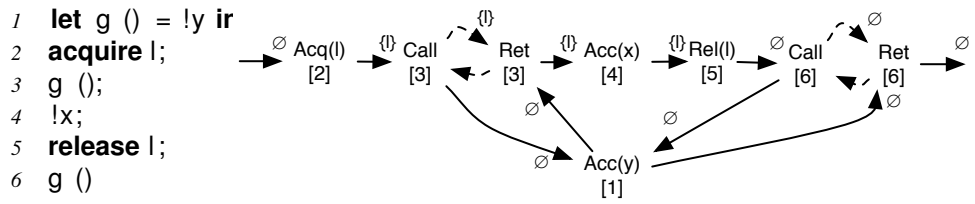


Figure 6.11: Splitting the lock state at a function call

input set intersected with $Flow(C, \ell)$, which contains the lock effect of the function, i.e., the aliases of all locks that may be changed by the called function. It is this intersection that will be propagated into the body of the function. The transfer function saves the remaining acquired locks in the set $Split(\phi')$. Then the transfer function for $\mathsf{Ret}(\ell, \phi')$ adds the saved locks back to the acquired set.

For example, consider the program in Figure 6.11. This program calls function g twice, once with l held, and once with l released. It also accesses x with l held, just after the first call to g. The function g itself accesses y. The right side of the figure shows the ACFG for this example, annotated with the lock state $Acq_{out}(\phi)$ at the end of each edge from statement ϕ. Statement numbers are given below the statement kinds. Initially, no locks are held ($\emptyset$ on the edge to Acq(l) [2]), and after line 2 the lock state is $\{\mathsf{l}\}$ (shown on the edge from Acq(l) [2]). Then at the call to g, we split the lock state into two parts. Since g does not acquire or release any locks, we propagate $\{\mathsf{l}\} \cap \emptyset = \emptyset$ from the Call [3] to Acc(y) [11]. During correlation inference, we will recover the fact that y was accessed with l held. We propagate the other part of the lock state, $\{\mathsf{l}\} \setminus \emptyset$, from Call [3] to Ret [3]. Next, after Ret [3], the lock state is $\{\mathsf{l}\} \cup \emptyset$, i.e., the locks that flowed "around" g plus the locks that flowed "through" g. Thus at Acc(x) [4], we see that l is held. Continuing through the program, at the next call to g, the empty set of locks is held, and that is propagated into g as before. This time after the Ret no locks are held, and the program continues.

Critically, with this analysis we can discover that x is guarded by l. Imagine if we had not split the lock state. Then we would have no dashed lines in the graph in Figure 6.11, and we would directly connect the Call and Ret nodes to Acc(y). But then we would have two edges flowing to Acc(y), one with state $\{\mathsf{l}\}$, and one with state $\emptyset$. We would then intersect these sets and decide that no locks were held at the entry to g. That summarization is fine for g, but when we propagate this information, we would decide no locks were held after the Ret statement labels in the ACFG, and thus we would think x was accessed with no lock held.

In essence, by splitting the lock state, we make functions parametric in locks they do not change; similar approaches have been used in other type systems [150, 59]. (We do propagate information about changed locks into the function, since otherwise we would not be able to track the state of those locks correctly.) We have found this kind of lightweight polymorphism critical to LOCKSMITH's precision. It is particularly important for commonly called functions such as printf, which would otherwise almost always cause the lock state to be empty upon their return.

ϕ kind	$Corr_{out}(\phi)$
$\mathsf{Acc}(\rho)$ (ρ shared)	$Corr_{in}(\phi) \cup \{\rho \rhd Acq_{out}(\phi)\}$
Fork	$Corr_{in}(\phi)$
$\mathsf{NewL}(\ell)$	$Corr_{in}(\phi)$

ϕ kind	$Corr_{out}(\phi)$
$\mathsf{Acq}(\ell)$	$Corr_{in}(\phi)$
$\mathsf{Rel}(\ell)$	$Corr_{in}(\phi)$
$\mathsf{Call}(\ell, \phi')$	$(Corr_{in}(\phi) + Split(\phi')) \cup Corr_{in}(\phi')$
$\mathsf{Ret}(\ell, \phi')$	$\emptyset$

Figure 6.12: Transfer functions for correlation inference

Precision Note that this analysis computes the set of locks that *must* be acquired at each program point. Since the analysis is necessarily conservative, it 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.

Correlation Inference: Backwards dataflow

LOCKSMITH also uses the dataflow-flow analysis engine to implement correlation inference. Recall from Section 6.5 that a correlation constraint has the form $\rho \rhd \{\ell_1, \ldots, \ell_n\}$, where the ℓ_i are the locks that are held during an access to ρ. To generate such constraints the analysis uses a backwards propagation, where the per-ϕ state is a set $Corr$ of correlations. Initially the set of correlations is empty for all statement labels, and the merge function is set union.

Figure 6.12 shows the transfer rules for correlation inference. Note that since this is a backwards analysis, $Corr_{in}(\phi)$ corresponds to the state after statement ϕ, and $Corr_{out}(\phi)$ corresponds to the state before ϕ.

The transfer function for $\mathsf{Acc}(\rho)$ adds $\rho \rhd Acq_{out}(\phi)$ to the set of correlations, where ρ is determined to be shared according to the sharing analysis (Section 6.10), and $Acq_{out}(\phi)$ was computed by the lock state inference. The transfer functions for Fork, $\mathsf{NewL}(\ell)$, $\mathsf{Acq}(\ell)$, and $\mathsf{Rel}(\ell)$ simply propagate the set of correlations. The last two transfer functions are the most interesting. Recall that during the lock state computation, any acquired locks that are not changed by a function are not propagated through the function body. However, any lock that is acquired for the duration of a function call clearly is correlated with all accesses that occur in the body of the function. Because of this, the transfer function for $\mathsf{Call}(\ell, \phi')$ adds the set $Split(\phi')$ of acquired locks that are "hidden" from the function body, to the lock set of every correlation for the function. We define $Corr_{in}(\phi) + Split(\phi')$ to be the set of correlations $\{\rho \rhd (\{\ell_1, \ldots, \ell_n\} \cup Split(\phi'))\}$ where $\rho \rhd \{\ell_1, \ldots, \ell_n\} \in Corr_{in}(\phi)$.

We also include in the output of $\mathsf{Call}(\ell, \phi')$ all correlations that occur after the function returns, $Corr_{in}(\phi')$. Then, the transfer function for $\mathsf{Ret}(\ell, \phi')$ always returns the empty set, as all correlations occurring after the function call are already propagated to the point before it. This speeds up correlation inference, because we need not propagate correlation information into called functions. Also, recall that due to the use of the spe-

cial call and return kinds in the lock state analysis, we "hide" a set of acquired locks for every function call. Clearly, as the locks in the split set are acquired during the function call, they protect all dereferences that occur in this given evaluation of the function body. We therefore need to add that "hidden" set of acquired locks to the set of correlations that occur in the function. Propagating correlations this way facilitates that, as only the correlations that occur in the function body are propagated through the $\mathsf{Call}(\ell, \phi')$ node.

To see these transfer functions in action, consider again the program in Figure 6.11. There are two initial correlations in this program: $\mathsf{x} \rhd \{\mathsf{l}\}$ (generated at $\mathsf{Acc(x)}$ [4]) and $\mathsf{y} \rhd \emptyset$ (generated at $\mathsf{Acc(y)}$ [1]). The correlation constraint $\mathsf{x} \rhd \{\mathsf{l}\}$ is first propagated backward to Ret [3], then to Call [3], then $\mathsf{Acq(l)}$ [2], and then to the beginning of the program. Notice that following the rule for Ret in Figure 6.12, we do not propagate this constraint backwards into node [11] in the called function.

There are two backward paths for the second correlation, on y. When we propagate it to the Call [3], we add the "hidden" lock l to the correlation, yielding $\mathsf{y} \rhd \{\mathsf{l}\}$ at Call [3]. When we propagate it to Call [6], there are no hidden locks, and so we get correlation $\mathsf{y} \rhd \emptyset$. We propagate both of these constraints unchanged to the start of the program. Thus we have that y is correlated with both $\{\mathsf{l}\}$ and $\emptyset$, and therefore y is not consistently guarded by a lock.

In our implementation, we also associate a call stack with each correlation constraint. When we propagate information through a Call node, we add the name of the called function to the call stack. In this way, once correlations reach the entry of the whole program, we can report not only what locations are correlated with which locks, but also on what paths the dereferences occurred. For example, for the code in Figure 6.11, if y were shared, we would report a data race, indicating that the accesses were due to the call on line 3 and the call on line 6. We have found that this "path" information makes LOCKSMITH error reports much easier to understand.

Finding races from inferred correlations After solving the correlation constraints, LOCKSMITH checks for consistent correlation among the inferred correlations in the set $Corr_{in}(\phi^{\mathsf{main}})$ which correspond to the initial state ϕ of function $\mathsf{main}()$. We compute the *correlation set* of a location label ρ to include every set of locks correlated with ρ:

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

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.

Fixpoint Computation Strategies

We implemented several strategies for finding a fixpoint of the sets computed by our dataflow analyses. First, we experimented with worklist-based schemes. In these approaches, each time the input set (i.e. the output of predecessors or successors, for a forwards or backwards analysis, respectively) computed for some node ϕ changed, we

Benchmark	Queue	Stack	Double Stack	Postorder Set	No worklist
aget	0.84	0.82	0.83	0.80	0.85
ctrace	0.61	0.58	0.60	0.57	0.59
engine	0.88	0.88	0.87	0.89	0.88
knot	0.77	0.74	0.76	0.74	0.78
pfscan	0.43	0.43	0.41	0.43	0.46
smtprc	5.92	6.77	5.63	4.31	5.37
3c501	9.03	9.46	9.21	9.39	9.18
eql	timeout	timeout	timeout	timeout	21.38
hp100	timeout	timeout	timeout	2524.49	143.23
plip	30.73	30.21	28.53	25.11	19.14
sis900	85.07	82.89	84.97	79.37	71.03
slip	timeout	timeout	timeout	timeout	16.99
sundance	104.22	108.92	108.59	103.73	106.79
synclink	timeout	timeout	timeout	timeout	1521.07
wavelan	19.69	20.08	19.66	19.76	19.70

Figure 6.13: Time (in seconds) to perform correlation inference using several fixpoint strategies

would add ϕ ϕ to the worklist for reconsideration. We tried four particular worklist implementations, discussed in detail by Cooper et al [25]: Queue, Stack, Double Stack, and Postorder Set. Initially, our implementations avoided adding duplicate nodes to the worklist, but we found that the cost to detect and eliminate duplicates is comparable to the gain from not processing the additional nodes. Thus, none of the implementations reported here attempt to eliminate duplicate nodes.

Second, we implemented the following simple strategy for backwards (forwards) analysis without a worklist:

1. Starting from the exit (entry) nodes, perform a postorder (reverse postorder) visit of the whole graph, applying the transfer function at each node to propagate the state to its predecessors (successors).

2. If anything changed during the last visit, then revisit the whole graph.

Using postorder for backwards analysis and reverse postorder for forwards analysis is extremely important [3]. For example, postorder traversal visits successors of ϕ before a node ϕ. Thus in a backwards analysis, a postorder traversal will (in the absence of cycles) require only a single pass through the ACFG to reach a fixpoint.

Figure 6.13 shows the times to perform correlation inference using the various strategies. A timeout indicates a run that did not terminate within one hour. We found that for our benchmarks, the Queue, Stack, and Double Stack strategies take roughly the same time, and the Postorder Set strategy is slightly faster. Surprisingly, we discovered that getting rid of the worklist altogether is the optimal strategy, performing significantly better on the larger benchmarks (the Linux kernel drivers). For example, on slip, all four

$$[\text{EFF-DEREF}]\frac{\Phi_1;\Gamma \vdash e : \mathit{ref}^{\rho}(\tau) \quad \Phi_2^{\varepsilon} = \mathit{Flow}(C,\rho) \quad \Phi_1 \rhd \Phi_2 \hookrightarrow \Phi}{\Phi;\Gamma \vdash\, !\, e : \tau} \qquad [\text{XFLOW-CTXT}]\frac{\Phi_1 = [\varepsilon_1;(\varepsilon_2 \cup \omega_2)] \quad \Phi_2 = [\varepsilon_2;\omega_2] \quad \Phi = [(\varepsilon_1 \cup \varepsilon_2);\omega_2]}{\Phi_1 \rhd \Phi_2 \hookrightarrow \Phi}$$

$$[\text{EFF-FORK}]\frac{\Phi_e;\Gamma \vdash e : \tau \quad \Phi_e^{\varepsilon} \subseteq \Phi^{\varepsilon} \quad \Phi_e^{\varepsilon} \cap \Phi^{\omega} \subseteq \mathit{SharLocs}}{\Phi;\Gamma \vdash \mathsf{fork}\ e : \tau}$$

Figure 6.14: Contextual effect rules for finding shared locations (selected)

worklist algorithms time out after one hour, whereas the no worklist strategy terminates in 17 seconds.

One would expect that the worklist strategies would visit far fewer nodes than the no-worklist strategy, and indeed this is the case for our benchmarks. However, LOCKSMITH uses hashconsing of state data structures to memoize transfer functions on control-flow nodes, and thus avoids re-computing the output state for every control-flow node visited, if the input has not changed. As a result, the cost of maintaining the worklist far exceeds the cost of redundant visits to nodes.

We do not present the respective measurements for the lock state analysis. We found that because each benchmark includes a relatively small set of locks, the sets of acquired locks at each program point are small. This makes the lock state analysis quite fast, as little information needs to be propagated. Indeed, for all the benchmark programs the running time for lock state analysis is negligible compared to the total running time, and all five strategies work equally well.

6.10 Sharing analysis

As we discussed in Section 6.9.3, during correlation inference we can safely ignore accesses to thread-local data, since such data need not be protected by locks. In this section we show how we compute the set of thread-shared locations. We found that our analysis allows LOCKSMITH to ignore a significant—usually dominant—fraction of the accesses in the program as thread-local. To get good precision, we had to develop some additional optimizations based on variable scoping and a simple uniqueness analysis. We also developed a refinement to our core analysis that determines, for each access to a thread-shared location, whether it occurs before or after the location becomes shared, so that only the latter accesses need be protected by locks. We found that this analysis, while potential useful, does not add much benefit in practice.

6.10.1 Contextual effects for finding shared locations

LOCKSMITH uses an effect system to infer which locations are thread-shared. Given some expression e, the *effect* ε of e is the set of all locations ρ that e could dereference during its evaluation [155]. In Chapter 5 we describe a generalization of effects that we

call *contextual effects* [113]. The contextual effect of e consists not only of the effect of e's computation, but also the effect α of the computation that has already occurred—called the *prior effect*—and the effect ω of the computation yet to take place—called the *future effect*. At every occurrence of fork e in the program, we compute the effect ε of e, the child thread, and the future effect ω of the parent thread. We consider as thread-shared those locations in the intersection of these two sets.

Figure 6.14 contains selected typing rules from our contextual effect system as applied to this sharing analysis. Full details can be found in the paper introducing contextual effects [113] and in Chapter 5. In these rules, a contextual effect Φ consists of a pair $[\varepsilon; \omega]$, where the first element is the standard effect, and the second element is the future effect; here, we ignore consideration of prior effects for simplicity. In our implementation, we generate effect-related constraints along with label flow constraints. For simplicity, we present effect typing rules here as a separate judgment, but it would be straightforward to merge these rules with those in Figure 6.9.

[EFF-DEREF] types a dereference $!\,e$. The first premise types expression e with effect Φ_1, and the second premise defines an effect Φ_2 to capture the behavior of the actual dereference, with a standard effect composed of the dereferenced location ρ and any locations that flow to it according to the label flow analysis (written $Flow(C, \rho)$). Here the syntax Φ^ε refers to the ε (i.e., the first) component of Φ. Finally, the third premise computes the contextual effect Φ of the entire expression by combining the effect Φ_1 of the subexpression e with the effect Φ_2 of the dereference itself, in the judgment $\Phi_1 \triangleright \Phi_2 \hookrightarrow \Phi$, defined by rule [XFLOW-CTXT].

[XFLOW-CTXT] defines the judgement $\Phi_1 \triangleright \Phi_2 \hookrightarrow \Phi$ that combines two effects Φ_1 and Φ_2 into effect Φ. We combine an effect Φ_1 with an effect Φ_2 when the expression with effect Φ_1 is evaluated *immediately before* the expression with effect Φ_2. This creates an effect Φ to describe the behavior of both expressions together. Specifically, since Φ_1 occurs *before* Φ_2, the future effect of Φ_1 should include Φ_2's standard effect ε_2 and its future effect ω_2, as shown in the first premise. The second premise is straightforward, defining the effect Φ_2. Since effect Φ describes both expressions with effects Φ_1 and Φ_2, its standard effect must contain both standard effects, and its future effect is the future effect of the last, namely Φ_2, as shown in the third premise.

Finally, [EFF-FORK] type checks thread creation. The second premise of [FORK] indicates that the standard effect Φ_e^ε of the thread itself should be contained in the effect of the parent. Notice the future effect Φ_e^ω of the thread is unconstrained (effectively making it $\emptyset$). In particular, as expected, it contains no information about the effect of the parent, since the two will execute in parallel. Finally, the third premise adds to the set *SharLocs* the locations that the parent and child thread (and threads they fork) could access in parallel: the intersection of the standard effect of the child thread Φ_e^ε and the future effect Φ^ω of the parent.

We can see this analysis in action in Figure 6.15. The left side of the figure shows a simple code example, and the right side shows the typing derivation for the last part of the example, involving the two thread forks.[1] Within this derivation, the boxed portion shows

[1]Note that the syntax $e_1; e_2$ can be treated as shorthand for $(\lambda x.e_2)\ e_1$ for some x that does not occur free in e_2.

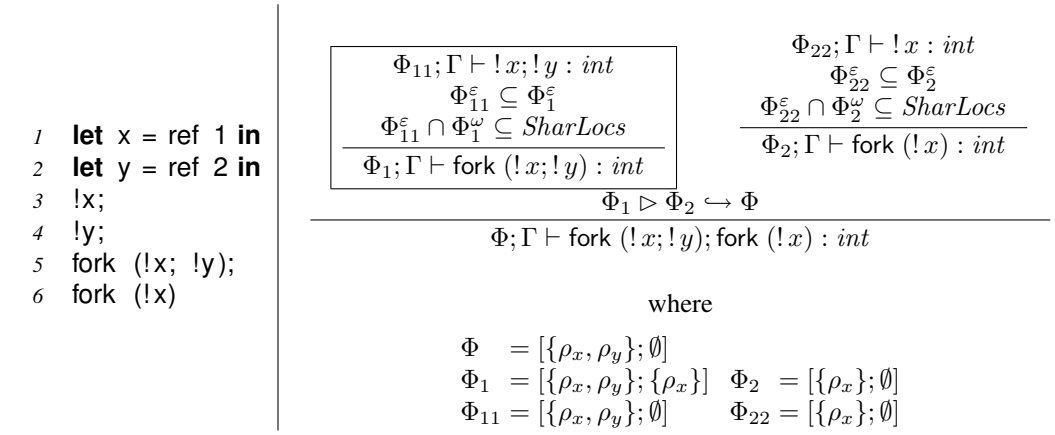


Figure 6.15: Example program to illustrate sharing analysis

the subderivation for the expression $\mathsf{fork}\ (!\,x; !\,y)$. Here we can see that the standard effect of this expression is $\Phi_1^{\varepsilon} = \{\rho_x, \rho_y\}$, i.e., locations corresponding to x and y. The future effect Φ_1^{ω} consists of the location ρ_x—this is because the effect of the subsequent thread spawn is $\{\rho_x\}$, and by [EFFDEREF], this effect is also attributed to the parent thread. Consequently, the intersection of these two effects is $\{\rho_x\}$, indicating that x is potentially accessed simultaneously by two threads. The second thread spawn yields no additional shared locations (since $\Phi_2^{\omega} = \emptyset$).

There are two interesting things to notice about this example. First, y is accessed in the parent thread, at line 4, and then subsequently in the first child thread. Nevertheless, our analysis does not consider y as shared, and indeed, y can never be simultaneously accessed by both threads. Second, the example illustrates that it is crucial to include the effect of a child thread in the effect of its parent. Otherwise, we would not have discovered that the two child threads both might simultaneously access x.

Fig. 6.16 measures the precision of the sharing analysis. For each benchmark, the second column shows the total number of pointers and the third column shows the number of shared pointers, computed by intersecting the effects at thread-creation points. We ignore the fourth column for now and return to it in Section 6.10.2. We also report the results of the sharing analysis in terms of allocation sites, where an allocation site is either a call to malloc() or the location of a variable definition (fourth and fifth columns, ignore the last column). The results underline the effectiveness of using contextual effects to compute shared memory locations; only 16% of all pointers and only 8% of all allocation sites are in the *ShLocs* set computed by [FORK]. This precision is critical to reducing false positives in LOCKSMITH: thread-local data is almost always accessed without a lock held, and thus if LOCKSMITH incorrectly determines a location is thread-shared, it will likely report a data race for that location.

6.10.2 Scoping and uniqueness

We used two additional optimizations to improve the results of the sharing analysis even further. Consider the program in Fig. 6.17. Here the reference g (line 1) is visible within the function f, which allocates two references x and y (lines 3–4), then

Benchmark	Pointers			Allocation Sites		
	Total	Shared	In scope	Total	Shared	In scope
aget	1,411	258	235	352	64	62
ctrace	1,089	129	116	311	12	12
engine	1,441	60	17	410	11	7
knot	1,238	338	238	321	30	15
pfscan	987	53	48	240	8	7
smtprc	4,275	196	67	1079	74	46
3c501	10,020	954	913	408	20	20
eql	4,572	2,377	2,168	273	43	35
hp100	19,401	5,268	5,210	497	15	15
plip	13,249	2,867	2,823	466	49	49
sis900	38,624	2,648	2,594	779	11	9
slip	13,748	1,338	1,281	382	20	19
sundance	34,142	3,313	3,267	753	9	9
synclink	51,147	11,621	11,472	1,298	155	139
wavelan	18,799	2,535	2,125	695	128	10

Figure 6.16: Precision of sharing analysis and scoping

writes to them (lines 5–6), and then assigns x to g (line 7). The program calls function f twice (lines 9–10) in two parallel threads. In this example, the sharing analysis from Sections 6.10.1 and 6.10.3 will determine that x and y are thread-shared at the writes on lines 5–6, because they are in the effects of both threads. However, in both cases this is overly conservative.

Scoping Optimization Notice that the location y refers to is allocated in the scope of f and never escapes. Hence, the uses of y by the two different threads must refer to distinct memory locations. More generally, when computing thread-shared locations, we can *hide* effects on locations that must be thread-local due to scoping. Formally, we change the type rule for fork as follows:

$$[\textsc{Eff-Fork-Down}]\frac{\begin{array}{c}\Phi_e;\Gamma \vdash e:\tau \\ \Phi_e^{\varepsilon} \subseteq \Phi^{\varepsilon} \\ \vec{\rho} = \bigcup_{\rho \in fl(\Gamma)} Flow(C,\rho) \\ \Phi_e^{\varepsilon} \cap \Phi^{\omega} \cap \vec{\rho} \subseteq SharLocs\end{array}}{\Phi;\Gamma \vdash \mathsf{fork}\ e:\tau}$$

Here we compute the set $\vec{\rho}$ of labels that are reachable in the label flow graph from locations in Γ, meaning they are visible to both the parent and child thread. Then we only add locations to $SharLocs$ if they are also in $\vec{\rho}$.

Fig. 6.16 shows the benefit of this optimization. The fourth column shows the number of shared pointers and the last column shows the number of shared allocation

```
1  let g = ref (ref 0) in
2  let f () =
3    let x = (ref 42) in
4    let y = (ref 0) in
5    y := 1;
6    x := 43;
7    g := x
8  in
9  fork f ();
10 f ()
```

Figure 6.17: Limitations of core sharing analysis

sites when using the revised rule [EFF-FORK-DOWN]. The average percentage of shared pointers and allocation sites improves to 15% and 6%, respectively.

Returning to the example program in Fig.6.17, note that the scoping optimization does not apply to for x, since it escapes via a write to the global variable g. However, while x does escape the scope of f eventually, there is no way that it can be accessed by any other thread during the write at line 6, since the aliasing on line 7 has not yet occurred. So, we can safely ignore the access to x at line 6, not requiring it to be protected by a lock.

This situation can occur in C programs when a struct is malloc'd and initialized thread-locally before becoming shared. To model this situation precisely, we developed a *uniqueness analysis* to determine when a memory access is guaranteed to be thread-local because the accessed location has not yet become aliased. We then ignore these accesses during the correlation analysis.

Our uniqueness analysis is implemented as a simple, intra-procedural dataflow analysis. Whenever a location is created (through a local variable definition or a call to malloc), we mark it as unique. When a unique location ρ is assigned to any non-unique location, or a variable with location ρ has its address taken, ρ becomes non-unique. Using this analysis, we discover that x is unique on line 4 in our example, and hence is thread-local.

Figure 6.18 shows the aggregate effect of these two optimizations for LOCKSMITH. We compare the running time and number of warnings when using both techniques, shown in the second and third columns, against the running time and number of warnings without them, in the fourth and fifth columns. Our results show that the effect on running times is negligible, but in several of the benchmarks the two optimizations determine that many accesses are thread-local, significantly reducing the number of warnings. In all benchmarks but ctrace, the gain in precision is due purely to the scoping optimization rather than the uniqueness analysis.

6.10.3 Flow-sensitive sharedness

Consider the example in Fig. 6.15 again. Suppose we add acquire l and release l before and after, respectively, each access $!x$ in the two child threads (lines 5 and 6), for some lock l. Also suppose that x is aliased by a global variable immediately after its allocation. This changed program will be correct, but our analysis will falsely complain

Benchmark	With scoping and uniqueness		Without scoping and uniqueness	
	Time (s)	Warnings	Time (s)	Warnings
aget	0.85	62	0.88	64
ctrace	0.59	10	0.58	10
engine	0.88	7	0.69	11
knot	0.78	12	0.83	28
pfscan	0.46	6	0.43	7
smtprc	5.37	46	5.46	74
3c501	9.18	15	8.93	15
eql	21.38	35	20.01	35
hp100	143.23	14	140.97	14
plip	19.14	42	18.75	44
sis900	71.03	6	70.77	6
slip	16.99	3	16.65	3
sundance	106.79	5	103.21	5
synclink	1521.07	139	1454.91	155
wavelan	19.70	10	20.60	128

Figure 6.18: Scoping and uniqueness

ϕ kind	$Sh_{out}(\phi)$
Fork	the portion of *SharLocs* computed at this fork site
all others	$Sh_{in}(\phi)$

Figure 6.19: Sharedness Inference

that the dereference of x at line 3 is a potential race because it is not protected by a lock. Moreover, the scoping optimization and uniqueness analysis described in Section 6.10.2 do not apply, as x is aliased by a global variable. However, at this dereference site, x is not actually shared, and thus requires no guarding lock. This is because it will not become shared until both of the child threads are spawned, at lines 5 and 6.

In this case, as in the uniqueness analysis in Section 6.10.2, we can safely ignore a memory access when the accessed location is thread-local during the access, even if it later becomes shared.

To address this problem, we can use a simple dataflow analysis along the ACFG to determine at which sites in the program a location can be dereferenced after it becomes shared. Any sites that occur before it becomes shared can be dropped from correlation inference. The transfer functions for the analysis are straightforward, as shown in Fig. 6.19. In essence, we seed the analysis at the fork points with those locations made possibly shared due that fork. Specifically, we combine the type rules in Fig. 6.14 with Fig. 6.8 to get a judgement of the form $C; \phi; \Phi; \Gamma \vdash \mathsf{fork}\ e : \tau; \phi$ for typing fork e expressions. Then,

Benchmark	No dataflow		Context insensitive		Context sensitive	
	Time(s)	Warnings	Time(s)	Warnings	Time(s)	Warnings
aget	0.80	62	0.85	62	1.73	62
ctrace	0.58	10	0.59	10	0.81	10
engine	0.89	7	0.88	7	1.27	7
knot	0.64	12	0.78	12	1.70	12
pfscan	0.45	6	0.46	6	0.57	2
smtprc	5.11	46	5.37	46	16.71	46
3c501	9.29	15	9.18	15	11.46	15
eql	21.39	35	21.38	35	24.35	35
hp100	143.45	14	143.23	14	172.97	14
plip	19.18	42	19.14	42	37.80	42
sis900	71.96	6	71.03	6	82.35	6
slip	17.05	3	16.99	3	18.92	3
sundance	106.89	5	106.79	5	117.01	5
synclink	1513.94	139	1521.07	139	1823.91	139
wavelan	19.69	10	19.70	10	26.84	10

Figure 6.20: The effect on LOCKSMITH's results of different dataflow strategies for finding shared location dereference sites

at every such point in the program, we set $\Phi_e^\varepsilon \cap \Phi^\omega \subseteq Sh_{in}(\phi)$ to seed the analysis.

Unfortunately, while this optimization adds precision in general, it is not very helpful for our benchmarks, as shown in Fig. 6.20. Columns 2 and 3 in the figure show the results of LOCKSMITH when using the contextual effects analysis (including scoping and uniqueness) to compute shared locations, without using the dataflow analysis, while columns 4 and 5 include the dataflow analysis. We see that the running times are nearly the same, but unfortunately, so are the warning counts. One reason for this is that a location that is eventually shared may be written to by the parent *after* the child is forked, and then shared with the child by writing to a global variable. The dataflow analysis conservatively considers all accesses after the fork to be potentially shared. When we make the dataflow analysis context-sensitive (Section 6.12.3), we see an improvement in one case—pfscan has 4 fewer warnings. However the context-sensitive results are clearly more expensive to compute. Thus, because employing dataflow is essentially free and could increase precision, we enable it by default, while context-sensitive dataflow for discovering sharing can be enabled via a command-line flag. (Thus the results from columns 4 and 5 in Fig. 6.20 match the results in Fig. 6.4.)

6.10.4 Contextual effects at fork points

A simple optimization that reduces the number of effect variables, is to reuse the same effect variable throughout any function that does not transitively fork. The only reason LOCKSMITH uses flow-sensitive effects, is to determine shared locations by in-

Benchmark	Functions		Effect variables		Time	
	Never fork	Include fork	All	Forking only	Unoptimized	Optimized
aget	74	3	4353	2703	0.99	0.85
ctrace	91	2	4379	1911	0.79	0.60
engine	64	1	4759	2243	1.28	0.88
knot	84	4	3959	2171	0.92	0.77
pfscan	83	1	4117	1937	0.56	0.45
smtprc	117	2	15521	6361	6.86	5.34
3c501	59	1	11165	8993	9.51	8.98
eql	47	1	3979	2417	21.62	26.48
hp100	108	1	12639	6757	147.03	181.45
plip	78	1	9749	6059	19.63	19.65
sis900	114	1	30967	26065	69.55	71.06
slip	72	1	10405	6269	17.44	15.92
sundance	98	1	26431	22189	109.83	103.21
synclink	163	1	21619	9143	1507.59	1420.27
wavelan	110	1	11479	6251	20.26	19.66

Table 6.1: Benefits of no-fork-effect optimization

tersecting the effect of the new thread with the effect of the main thread at every fork point. Hence, for any function that does not fork, there is no need for flow-sensitive precision when calculating continuation effects. We have measured the number of functions that might transitively fork in all the benchmarks, the number of labels used for effects both when using the optimization and when not, and the running time. The results are presented in Figure 6.1.

6.11 Efficiently and precisely modeling `struct` and `void*` types

In this section we discuss some additional techniques that we used to increase the speed and precision of LOCKSMITH as applied to C programs. In particular, we explain how we analyze `struct` and `void*` types effectively. We initially explored some of these ideas when developing CQual [58]. This work's contribution is to express the ideas more precisely, in particular using a new formalism for our analysis of structures, and to measure their costs and benefits directly, measuring LOCKSMITH's performance and precision when using one or the other of several different strategies.

6.11.1 Field-sensitivity

In designing a static analysis for C, one important decision whether to model C `struct` types *field-insensitively* or *field-sensitively* [70]. In a field-insensitive analysis, all fields of a `struct` type are conflated, i.e., `x.f` and `x.g` are treated as the same location by the analysis for any fields `f` and `g`. In a field-sensitive analysis, different `struct` fields are

Program	Field-sensitive					Field-insensitive				
	CGen Tm (s)	Total Tm (s)	Labels	Shr	Wrn	CGen Tm (s)	Total Tm (s)	Labels	Shr	Wrn
aget	0.55	0.85	5,634	62	62	0.50	0.67	5,490	62	62(11)
ctrace	0.40	0.59	4,351	12	10	0.38	0.53	4,285	15	13(5)
engine	0.76	0.88	5,051	7	7	0.79	0.91	4,989	59	59(7)
knot	0.55	0.78	4,752	15	12	0.52	0.83	4,566	24	21(12)
pfscan	0.36	0.46	4,143	7	6	0.36	0.46	4,139	15	14(5)
smtprc	3.09	5.37	14,815	46	46	3.08	5.14	14,917	97	97(43)
3c501	7.92	9.18	25,905	20	15	7.60	18.56	22,976	42	42(6)
eql	2.72	21.38	8,954	35	35	2.39	17.99	7,484	42	42(18)
hp100	35.92	143.23	31,609	15	14	34.18	976.12	22,214	41	41(10)
plip	16.41	19.14	24,124	49	42	17.82	103.21	18,969	60	60(6)
sis900	65.66	71.03	84,797	9	6	60.45	132.18	71,630	42	42(6)
slip	15.11	16.99	25,371	19	3	15.44	33.24	18,333	56	31(5)
sundance	96.72	106.79	73,552	9	5	81.44	6835.26	61,540	44	44(8)
synclink	1433.56	1521.07	68,643	139	139	1232.05	timeout	n/a	171	n/a
wavelan	17.89	19.70	30,052	10	10	16.90	40.19	21,071	43	44(6)

Figure 6.21: Field-sensitivity

distinguished, i.e., x.f and x.g are treated as different locations.[2] These two design points potentially trade off efficiency and precision—field-insensitive analysis may be less precise but more scalable, because it distinguishes fewer locations. In particular, if there are m occurrences of struct types, each of which has n fields, then field-sensitive analysis would annotate $O(mn)$ types with fresh locations, whereas field-insensitive analysis would only annotate $O(m)$ types.

We implemented support for both field-insensitive and field-sensitive analysis in LOCKSMITH. Field insensitivity is actually somewhat tricky to use in an analysis like LOCKSMITH, which is layered on top of C types: True field-insensitivity would throw away those types, thereby requiring some significant approximations in the analysis (e.g., conflating all labels of all fields of a struct). Thus, since we had a full field-sensitive analysis, we opted to implement a simple variation to simulate the following key aspect of field-insensitivity: each instance of a struct uses a single location ρ to represent the top-level location of all fields. Otherwise we use the standard flow-sensitive implementation. For example, suppose we have a struct with three fields int x, int $*$ y, and int $*$ z. Then for an instance of this struct, fields x, y, and z would have types $ref^{\rho}(int)$ (since x can be written to, it has a ref type), $ref^{\rho}(ref^{\rho'_y}(int))$, and $ref^{\rho}(ref^{\rho'_z}(int))$, respectively.

Figure 6.21 compares the two approaches. For each style of analysis, we list the time for constraint generation (including annotating types with fresh labels, i.e., abstract locations and locks), the total analysis time, the number of generated labels (locations and locks), the number of shared locations, and the number of reported warnings. Note

[2]There is a third design point, a *field-based* analysis, in which x.f and x.g are different, but x.f and y.f are the same if x and y are instances of the same struct type. We did not explore this option for LOCKSMITH, however, because it would greatly reduce the effectiveness of context-sensitivity, which we found was important to improving precision.

that since LOCKSMITH issues one warning per unprotected shared location, this means warning counts from field-insensitive and -sensitive analysis are incomparable: A single warning from field-insensitive analysis might actually correspond to multiple races from the field-sensitive analysis. To normalize the warning counts, if there is a warning on a location corresponding to a struct field, we count that as n warnings for comparison purposes, where n is the number of fields in that struct instance, as computed by our lazy fields algorithm (Section 6.11.2). The normalized warnings for the field-insensitive analysis are listed in the rightmost column, with the raw number of warnings in parentheses.

These results show that the field-insensitive analysis takes less time to generate constraints and generally creates fewer labels than the field-sensitive analysis.[3] However, even for the very slightly reduced precision with field-insensitivity—conflating the locations of all struct fields—many more locations are considered shared, which in turn makes LOCKSMITH as a whole both less precise, as evidenced by the warning counts, and considerably slower, since it must infer correlations for more aliases.

6.11.2 Lazy struct fields

Since field-sensitive analysis can potentially be expensive, in order to achieve the performance reported in Figure 6.21 we had to implement field-sensitivity carefully. At first, we used a naive approach, in which we fully annotated all field types of all struct instances. We quickly ran in to scalability problems, however, and were not able to analyze any but the smallest benchmarks.

Examining our benchmarks, we found that many C struct types have a large number of fields (up to 300!). However, many large struct types are declared by a library and only used in a small subset of the code, and this subset often accesses only a fraction of the struct's total fields. Our naive implementation was assigning abstract locations and locks to all of the rarely- or never-used fields, wasting memory and time generating constraints among them.

To regain scalability, our field-sensitive implementation *lazily* annotates the fields of struct types [58]. Initially we leave all occurrence of struct types unannotated. Then whenever we encounter a field access in the program, we add the accessed field to the corresponding struct type. If we create a label flow constraint between two struct types, we equate the labels on their fields.

Figure 6.22 extends the inference rules from Figure 6.8 to implement this lazy field generation algorithm. Our formalism uses pairs instead of general structs, and so we illustrate our approach by modeling pairs lazily. Figure 6.22(a) gives the new type and constraint definitions. Types are the same as before, except pair types now have the form $t \times^{\zeta} t$, where ζ is a *pair label*. Notice that this pair type contains unannotated types t. The pair label ζ is used to track the labeled components of the type: the constraint $\zeta[j] = \tau$ indicates that component j of any pair type annotated with ζ has annotated type τ. The constraint $\zeta_1 = \zeta_2$ indicates the corresponding components of pairs labeled with ζ_1 and ζ_2

[3]For one program, smtprc, there are fewer field-sensitive labels than field-insensitive labels. This is because the field-insensitive analysis always creates at least one label (for the location of all fields) for every occurrence of a struct, whereas the field-sensitive analysis might avoid creating any labels for a struct instance if it is created and then immediately equated with another struct instance.

$$
\begin{array}{rcl}
\tau & ::= & \ldots \mid t \times^{\zeta} t \\
C & ::= & \ldots \mid \zeta[j] = \tau \mid \zeta = \zeta
\end{array}
$$

$$
\zeta \quad \in \quad \textit{pair labels}
$$

$$
\begin{array}{rcl}
\langle\langle t_1 \times t_2 \rangle\rangle & = & t_1 \times^{\zeta} t_2 \qquad \zeta \textit{ fresh} \\
\langle\langle \textit{other } t \rangle\rangle & = & \tau \text{ as in Figure 6.7} \\
\langle\langle \tau \rangle\rangle & = & \tau' \text{ where } \tau' \text{ is } \tau \text{ with fresh } \rho, \ell, \phi, \zeta\text{'s}
\end{array}
$$

(a) Auxiliary definitions

$$
[\textsc{Pair}]\frac{\begin{array}{c} C; \phi; \Gamma \vdash e_1 : \tau_1; \phi_1 \\ C; \phi_1; \Gamma \vdash e_2 : \tau_2; \phi_2 \\ \zeta \textit{ fresh} \\ C \vdash \zeta[1] = \tau_1 \\ C \vdash \zeta[2] = \tau_2 \\ t_j = \textit{std type of } e_j,\ j \in 1..2 \end{array}}{C; \phi; \Gamma \vdash (e_1, e_2) : t_1 \times^{\zeta} t_2; \phi_2}
$$

$$
[\textsc{Proj}]\frac{\begin{array}{c} C; \phi; \Gamma \vdash e : t_1 \times^{\zeta} t_2; \phi' \\ j = 1, 2 \\ \tau_j = \langle\langle t_j \rangle\rangle \\ C \vdash \zeta[j] = \tau_j \end{array}}{C; \phi; \Gamma \vdash e.j : \tau_j; \phi'}
$$

(b) Type inference rules

$$
\begin{array}{rcl}
C \cup \{t_1 \times^{\zeta} t_2 \leq t_1 \times^{\zeta'} t_2\} & \Rightarrow & C \cup \{\zeta = \zeta'\} \\
C \cup \{\zeta = \zeta'\} & \Rightarrow & C[\zeta/\zeta'] \\
C \cup \{\zeta[j] = \tau_1, \zeta[j] = \tau_2\} & \cup \Rightarrow & \{\tau_1 = \tau_2\}
\end{array}
$$

(c) Constraint resolution rule

Figure 6.22: Type inference rules for modeling pairs lazily

have the same types.

We also extend $\langle\langle \cdot \rangle\rangle$ to introduce fresh pair labels. As shown, when we translate a standard pair type into a labeled pair type, we tag it with a fresh pair label but do *not* introduce labels for the component types. This annotation function is used in [LAM] from Figure 6.8 to give fresh labels to programmer-supplied types. Thus we see the laziness of this approach: we do not automatically create labels for a pair type when it is mentioned in the program text.

Figure 6.22(b) gives our modified type rules. [PAIR] types pair creation, which now associates a fresh pair label ζ with the output type and constrains the components of ζ to their corresponding labeled types. Notice that there is no laziness in this rule, because we have labeled types for e_1 and e_2. Here we compute the standard types t_1 and t_2 (of e_1 and e_2, respectively) by stripping off all labels from τ_1 and τ_2. Using standard types keeps the analysis "lazy." Standard types have no (wasted) location or lock annotations; instead, we track the omitted locations off to the side using ζ.

More interestingly, [PROJ] types projection, which lazily annotates only the accessed component of the pair. We create a type τ_j with fresh labels and constrain it to be equal to $\zeta[j]$. In our implementation, rather that creating constraints $\zeta[j] = \tau$ and then solving them later, we solve these constraints on-line, as we apply the type inference rules. We maintain a partial mapping ST from each $\zeta[j]$ to its type. Initially ST is empty. In [PROJ], if $ST(\zeta[j])$ exists, we use it in place of τ_j rather than making up a fresh type. Otherwise, we do make a fresh type τ_j and set $ST(\zeta[j]) = \tau_j$. Our implementation for [PAIR] is similar.

Figure 6.22(c) gives the resolution rules for our new constraint forms. The first rewriting rule is for subtyping among pair types. Here we assume the standard types of the pairs match (i.e., the program passes the standard type checker) and equate the pair labels, which are merged by the next rewriting rule. The last rule equates types for different occurrences of $\zeta[j]$.

Notice that we lose some precision here compared to the previous type system. In our lazy approach, subtyping among pair types requires their component types to be equal. The constraint resolution rule from Figure 6.9, on the other hand, permits subtyping the component types, which is more precise. However, in practice, C programs mostly manipulate pointers to structs, and subtyping pointer types requires that the pointed-to types are equal (Figure 6.9), which negates any benefits of the more precise subtyping rule. Thus we lose little practical precision with this approach.

Moreover, in our implementation, we maintain pair labels ζ in a union-find data structure. Given the constraint $\zeta = \zeta'$, we unify the two sides of the equation and equate the associated types in ST. This unification process reduces the need to create types for fields. For example, if $\zeta[0] = \tau$ and $\zeta'[1] = \tau'$ are the only mappings in ST, then after we unify the pair labels we will have $\zeta[0] = \zeta'[0] = \tau$ and $\zeta[1] = \zeta'[1] = \tau'$, without creating any additional field types. Thus, we also gain efficiency from this approach.

We already saw in Figure 6.21 that field-sensitivity, while it typically produces more labels that field-insensitive analysis, does not yield an inordinate number of labels. Figure 6.23 gives some measurements that illustrate why this is the case. For each benchmark, we list the total number of struct types in the program, the number of instances of all struct types, the total possible number of instance fields, and the instance fields that are actually used, both in absolute numbers and as a percentage of the total number of fields. We define the total number of instance fields as all the used fields of all instances of struct types. For example, if a program defines two instances of a single struct type with three fields and the program accesses all fields of one instance and two of the other so that the lazy field analysis only populates those with location labels, we "use" five instance fields out of a total of six. This data shows that on average across all the benchmarks, only 35.47% of the possible instance fields are actually used. Thus, lazy field analysis is effective because modeling those fields would otherwise consume memory and time with no gain in precision.

6.11.3 Modelling `void*`

In addition to deciding how to model structs, another important decision in analyzing C code is determining how to model `void*`, which typically used by C programmers

Benchmark	struct Types	Instances		
		Total	Total fields	Used fields
aget	11	61	563	179 (32%)
ctrace	12	43	427	79 (19%)
engine	14	48	618	85 (14%)
knot	16	68	565	156 (28%)
pfscan	13	65	639	78 (12%)
smtprc	19	80	1019	106 (10%)
3c501	37	1025	14691	3768 (26%)
eql	33	888	9617	2301 (24%)
hp100	36	1786	41537	12072 (29%)
plip	46	1986	24161	7320 (30%)
sundance	58	4141	51400	16504 (32%)
sis900	60	4511	58952	18106 (31%)
slip	39	1426	31529	8319 (26%)
synclink	49	5431	68423	38497 (56%)
wavelan	58	2879	31823	11608 (36%)

Figure 6.23: Lazy field statistics

to express polymorphism. For LOCKSMITH, the key choice is how to track the abstract locations and locks of types that "flow" to or from `void*` positions. We experimented with three different strategies:

- *Conflate* `void*`. Since any type might be cast to or from `void*`, a conservative but sound approach is to conflate all abstract locations and locks that reach a `void*` type. More precisely, let $\tau = \mathit{ref}^{\rho}(\texttt{void})$ be an occurrence of `void*` in the program, labeled with location ρ. If we ever derive a constraint $\tau \leq \tau'$ or $\tau' \leq \tau$, we equate all the locations in τ' with ρ. This is quite conservative, since it effectively aliases all locations reachable from a type that flows to or from a `void*`. If any locks occur inside τ', LOCKSMITH warns about the loss of precision, and considers these locks non-linear and thus unable to protect memory locations.

- *Singleton* `void*`. In the previous approach, we conflated labels because `void*` types may be cast arbitrarily. However, it could be that a particular `void*` in the program is used with only one concrete type. We thus tried refining the previous approach as follows. Let $\tau = \mathit{ref}^{\rho}(\texttt{void})$ be an occurrence of `void*`. We wish to define the partial function *base_type* as a map from `void*` occurrences to the single concrete type it could be replaced with. Given a constraint $\tau \leq \tau'$ or $\tau' \leq \tau$, there are three cases. If $\mathit{base\_type}(\tau)$ is as yet undefined, we set it to τ'. Otherwise, if τ' has the same shape (i.e., underlying standard type) as $\mathit{base\_type}(\tau)$, we generate the constraint $\mathit{base\_type}(\tau) \leq \tau'$ or $\tau' \leq \mathit{base\_type}(\tau)$, as appropriate. Otherwise, we revert to the above conflation strategy, and collapse $\mathit{base\_type}(\tau)$ and any other types that flow to τ. As before, if we collapse types then we treat any locks oc-

Benchmark	Conflate `void*`		Single `void*`		Type-based `void*`	
	Tm (s)	Wrn	Tm (s)	Wrn	Tm (s)	Wrn
aget	1.89	72	1.98	72	0.85	62
ctrace	0.60	10	0.61	10	0.59	10
engine	0.90	7	0.89	7	0.88	7
knot	1.46	12	0.77	12	0.78	12
pfscan	0.45	6	0.46	6	0.46	6
smtprc	5.22	46	5.26	46	5.37	46
3c501	246.24	121	358.98	121	9.18	15
eql	12.40	41	12.58	42	21.38	35
hp100	105.80	50	782.52	50	143.23	14
plip	413.96	60	4011.22	148	19.14	42
sis900	778.20	149	6037.00	152	71.03	6
slip	88.79	68	timeout	n/a	16.99	3
sundance	3188.32	148	7661.57	148	106.79	5
synclink	timeout	n/a	timeout	n/a	1521.07	139
wavelan	168.28	112	169.03	111	19.70	10

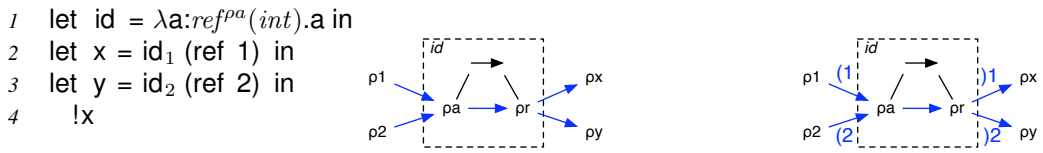
Figure 6.24: Performance of `void*` strategies

curring inside τ as non-linear, and assume they do not protect any locations. This approach is sound but is more precise than conflation in the case of `void*`s that are used with only one type. Indeed, we found that approximately one third of `void*` pointers in our benchmarks are only cast to or from one non-`void*` type.

- *Type-based* `void*`. Finally, we can improve further on the previous approach if we are willing to sacrifice completely sound modeling of `void*`. For each standard type t that flows to $\tau = \mathit{ref}^{\rho}(\texttt{void})$, we create a type $\mathit{base\_type}(\tau, t)$. Then given a constraint $\tau \leq \tau'$ or $\tau' \leq \tau$, we generate the constraint $\mathit{base\_type}(\tau, t) \leq \tau'$ or $\tau' \leq \mathit{base\_type}(\tau, t)$, where t is the underlying standard type from τ'. Thus, our *base_type* function is now indexed by the shape of the underlying type, similar to a C untagged union. We will never collapse types (or mark locks as non-linear) using this strategy. Modeling `void*`s this way is unsound, since we may miss relationships among different types that are cast to or from `void*`—this would be like storing a pointer into an untagged union but then extracting an integer. Our assumption is that this behavior is unlikely or harmless to our analysis, since if it were not the program would likely fail.

For all of these approaches, we need to integrate our modeling of `void*` with our lazy modeling of structs. That is, we might discover during constraint resolution that a `void*` type points to a struct type, or that a struct type contains a `void*` type.

Figure 6.24 compares the running times and number of warnings produced for each `void*` strategy. We can see that on most of the benchmarks, the type-based `void*` approach yields both many fewer warnings and is much faster than the other approaches.



(a) Source program (b) Monomorphic analysis (c) CFL-based polymorphic analysis

Figure 6.25: Precision gain from context-sensitive analysis for label flow

This phenomenon is similar to that of Figure 6.21: the more precise analysis causes fewer locations to be conflated, which both speeds up the computation of the $Flow()$ sets and reduces the number of shared locations. Moreover, the type-based warnings are much easier to follow for the user, as there is much less false aliasing due to conflation. Though the type-based approach could be unsound, a manual analysis of a sample of the additional warnings produced by the alternative analyses found no additional races.

6.12 Context sensitivity

So far, the analyses we have presented have been context-insensitive, meaning they conflate all calls to the same function. While the resulting analysis is easy to understand and implement, its precision suffers.

Figure 6.25 gives the canonical example illustrating the benefits of context sensitivity for label flow analysis. (We discuss context-sensitive dataflow analysis below.) This program defines an identity function id and applies it twice on distinct locations, on lines 2 and 3. As in Section 6.1, we have indexed each syntactic use of id with an integer. Figure 6.25(b) shows a simplification of the constraint graph produced by applying the context-insensitive type rules in Figure 6.8. Here ρi is the location containing integer i, locations ρa and ρr are from the domain and range types of id, respectively, and ρx and ρy are from the types of x and y. Notice that when we compute the transitive closure of these constraints, we will discover that both $\rho 1$ and $\rho 2$ flow to ρx, even though only $\rho 1$ may actually reach the dereference of x at run time.

Figure 6.25(c) shows how using context-free language reachability, which we discussed briefly in Section 6.1, eliminates this imprecision. When we use the type of id, we label the generated constraints with indexed parentheses. In our example, the call on line 2 yields edges $\rho 1 \rightarrow^{(1} \rho a$ and $\rho a \rightarrow^{)1} \rho x$, and analogously for the call on line 3. When we resolve the constraints in Figure 6.25(c), we only transitively close paths that contain no mismatched edges. In this case, that means there is a path from $\rho 1$ to ρx, since $(1$ matches $)1$, but there is no path from $\rho 2$ to ρx, since $(1$ does not match $)2$.

In the remainder of this section, we show how to incorporate this idea into our system and also apply it to dataflow analysis (analogously to Reps et al [135]). We end with experimental results illustrating the precision benefits of context sensitivity.

$$
\begin{array}{rcl}
e & ::= & \ldots \mid \mathsf{let}\ f = v\ \mathsf{in}\ e_2 \mid f_i \mid \mathsf{fix}^v\ f.t \\
\sigma & ::= & (\forall.\tau, \vec{\eta}) \\
\eta & ::= & \ell \mid \rho \mid \phi \\
C & ::= & \ldots \mid \eta \preceq^i_+ \eta \mid \eta \preceq^i_- \eta
\end{array}
$$

(a) Extensions to source language, types, and constraints

$$
[\textsc{Let}]\ \frac{C; \phi_1; \Gamma \vdash v_1 : \tau_1; \phi_2 \quad \vec{\eta} = \mathit{fl}(\Gamma) \quad C; \phi_2; \Gamma, f : (\forall.\tau_1, \vec{\eta}) \vdash e_2 : \tau_2; \phi_3}{C; \phi_1; \Gamma \vdash \mathsf{let}\ f = v_1\ \mathsf{in}\ e_2 : \tau_2; \phi_3}
\qquad
[\textsc{Inst}]\ \frac{\tau' = \langle\langle \tau \rangle\rangle \quad C \vdash \tau \preceq^i_+ \tau' \quad C \vdash \vec{\eta} \preceq^i_\pm \vec{\eta}}{C; \phi; \Gamma, f : (\forall.\tau, \vec{\eta}) \vdash f_i : \tau'; \phi}
$$

$$
[\textsc{Fix}]\ \frac{\begin{array}{c}\tau = \langle\langle t \rangle\rangle \quad \vec{\eta} = \mathit{fl}(\Gamma) \quad C; \phi_1; \Gamma, f : (\forall.\tau, \vec{\eta}) \vdash v : \tau'; \phi_2 \\ C \vdash \tau' \leq \tau \quad \tau'' = \langle\langle t \rangle\rangle \quad C \vdash \tau \preceq^i_+ \tau'' \quad C \vdash \vec{\eta} \preceq^i_\pm \vec{\eta}\end{array}}{C; \phi_1; \Gamma \vdash \mathsf{fix}^v\ f.t : \tau''; \phi_2}
$$

(b) Additional type inference rules

Figure 6.26: Extensions to Figure 6.7 and 6.8 for context sensitivity

6.12.1 Labeling and constraint generation

We use an approach pioneered by Reps et al [135] of reducing the problem of tracking flow context-sensitively through function calls to the problem of context-free language (CFL) reachability. The insight is to view a call to and return from some function f as a string containing a left and right parenthesis, respectively, subscripted by an index identifying the call-site. Thus the problem of tracking flow through function calls is one of matching like-subscripted parentheses. We draw ideas more directly from Rehof, Fähndrich et al [132, 40], which apply Reps et al.'s idea to label flow analysis and points-to analysis, respectively. To solve context-free language reachability constraints, we use BANSHEE which encodes and solves the problem using set constraints.

Figure 6.26 extends our core constraint generation rules from Figure 6.8. We begin by introducing three new kinds of expressions, as shown in Figure 6.26(a). Expression $\mathsf{let}\ f = v\ \mathsf{in}\ e$ binds f to value v during evaluation of e, assigning f a polymorphic type. Here we assume that the names of polymorphically-typed variables are syntactically distinct from other (monomorphically) typed variables. In practice for C, we only introduce polymorphism for functions, whose names are easily identified. Next, the expression f_i corresponds to a use of variable f annotated with index i. In practice, we simply assign a distinct index to each syntactic use of a function name. Finally, $\mathsf{fix}^v\ f.t$ binds f to v recursively inside of v (which will always be a function in practice). In C, all functions are potentially mutually recursive, and so we treat a C program as if it were a set of nested fix bindings.

Let- and fix-bound variables f are assigned polymorphic *type schemes* σ of the form $(\forall.\tau, \vec{\eta})$. Here τ is the generalized type, and $\vec{\eta}$ is the set of labels (i.e., ρ's, ℓ's and ϕ's) that are *not* quantified in the type scheme [71]. During typing of the new language forms, we generate *instantiation constraints* of the form $\eta \preceq^i_p \eta'$, where p is a *polarity*, either $+$ or $-$, and i is an index. Informally, such a constraint means that there is some substitution S_i

$$
\begin{array}{rcl}
C \cup \{int \preceq^i_p int\} & \Rightarrow & C \\
C \cup \{ref^\rho(\tau) \preceq^i_p ref^{\rho'}(\tau')\} & \Rightarrow & C \cup \{\rho \preceq^i_p \rho', \tau \preceq^i_\pm \tau'\} \\
C \cup \{lock^\ell \preceq^i_p lock^{\ell'}\} & \Rightarrow & C \cup \{\ell \preceq^i_p \ell'\} \\
C \cup \{t_1 \times^\zeta t_2 \preceq^i_p t_1 \times^{\zeta'} t_2\} & \Rightarrow & C \cup \{\zeta \preceq^i_\pm \zeta'\} \\
C \cup \{(\tau_1, \phi_1) \to (\tau_1', \phi_1') \preceq^i_p (\tau_2, \phi_2) \to (\tau_2', \phi_2')\} \Rightarrow & & \\
\qquad C \cup \{\tau_1 \preceq^i_{\bar{p}} \tau_2, \phi_1 \preceq^i_{\bar{p}} \phi_2, \tau_1' \preceq^i_p \tau_2', \phi_1' \preceq^i_p \phi_2'\} & &
\end{array}
$$

$$
\begin{array}{rcl}
C \cup \{\zeta \preceq^i_\pm \zeta'\} \cup \{\zeta[j] = \tau\} & \cup \Rightarrow & \{\tau \preceq^i_\pm \zeta'[j]\} \\
C \cup \{\zeta' \preceq^i_\pm \zeta\} \cup \{\zeta[j] = \tau\} & \cup \Rightarrow & \{\zeta'[j] \preceq^i_\pm \tau\}
\end{array}
$$

$$
\begin{array}{rcl}
C \cup \{\rho_1 \preceq^i_- \rho_0, \rho_1 \leq \rho_2, \rho_2 \preceq^i_+ \rho_3\} & \cup \Rightarrow & \{\rho_0 \leq \rho_3\} \\
C \cup \{\ell_1 \preceq^i_- \ell_0, \ell_1 \leq \ell_2, \ell_2 \preceq^i_+ \ell_3\} & \cup \Rightarrow & \{\ell_0 \leq \ell_3\}
\end{array}
$$

Figure 6.27: Extensions to Figures 6.9 and 6.22 for context sensitivity

that instantiates η to η'. The polarity indicates the direction of "flow". More particularly, a constraint $\eta \preceq^i_+ \eta'$ corresponds to an *output* from a function, and we draw it with an edge $\eta \to^{)i} \eta'$. Similarly, a constraint $\eta \preceq^i_- \eta'$ corresponds to an *input* to a function, and we draw it with an edge $\eta' \to^{(i} \eta'$. Notice that for a negative polarity constraint, the direction of the graph edge is opposite the direction of the $\preceq$.

Figure 6.26(b) shows the new type inference rules.[4] [LET] first types v_1, and then types e_2 with f bound to the type scheme $(\forall.\tau_1, \vec{\eta})$, where τ_1 is the type of v_1, and $\vec{\eta}$ is the set of free labels of Γ (as usual for Hindley-Milner-style polymorphism, these are the labels we cannot quantify [121]). In [INST], we instantiate a type scheme $(\forall.\tau, \vec{\eta})$ at index i. We generate a type τ' by re-annotating τ with fresh labels. We then generate an instantiation constraint $\tau \preceq^i_+ \tau'$ to indicate that τ is used at index i at type τ', and we generate constraints $\vec{\eta} \preceq^i_\pm \vec{\eta}$ to indicate that the substitution S_i represented by the constraint $\tau \preceq^i_+ \tau'$ must not rename any variables in $\vec{\eta}$, i.e., they must be instantiated to themselves. (Here the $\pm$ is shorthand for generating two constraints, one with polarity $+$ and one with polarity $-$.) Lastly, [FIX] combines [LET] and [INST], binding f to a type scheme during the typing of v, and then instantiating f to a fresh type as the result.

6.12.2 Context sensitive label flow constraint resolution

To compute the flow of labels in our new constraint system, we extend our constraint rewriting rules as shown in Figure 6.27. The first set of rewrite rules corresponds to the standard subtyping rules. We reduce $\preceq^i_p$ constraints to components of a type in a manner that is invariant for references and pairs (due to lazy fields, and thus analogously to equating pair labels in Figure 6.22), and co- and contra-variant for function return and argument types, respectively. Here we write $\bar{p}$ for the opposite of polarity p.

The next two rules propagate instantiation constraints to components of a lazy pair. The first rule requires that if ζ is instantiated to ζ', then the j component of ζ is instantiated to the j component of ζ'. The next rule handles the other direction of instantiation. Note

[4] We have implicitly relaxed the definition of Γ to also include type schemes σ.

that in both rules, the generated constraint on the right-hand side refers to $\zeta'[j]$ although that might be empty. In that case we assume that it is set to $\langle\langle\tau\rangle\rangle$ (not shown), i.e., a type of the appropriate shape, annotated with fresh labels.

The last two rewrite rules propagate constraints along paths with matched parentheses. Pictorially, given a sequence of constraints $\rho_0 \rightarrow^{(i} \rho_1 \rightarrow \rho_2 \rightarrow^{)i} \rho_3$, the first rewrite rule generates a new constraint $\rho_0 \rightarrow \rho_3$, derived from matching the parentheses on the path; this is called by *matched flow* by Rehof et al [132]. The last rewriting rule follows the same pattern for abstract locks.

Given these rewriting rules, our definition of $Flow()$ remains the same:

$$\begin{array}{rcl} Flow(C,\rho) & = & \{\rho' \mid \rho' \leq \rho \in Sol(C)\} \\ Flow(C,\ell) & = & \{\ell' \mid \ell' \leq \ell \in Sol(C)\} \end{array}$$

For example, letting C be the constraints in Figure 6.25, we have $Flow(C,\rho x) = \{\rho 1\}$ and $Flow(C,\rho y) = \{\rho 2\}$.

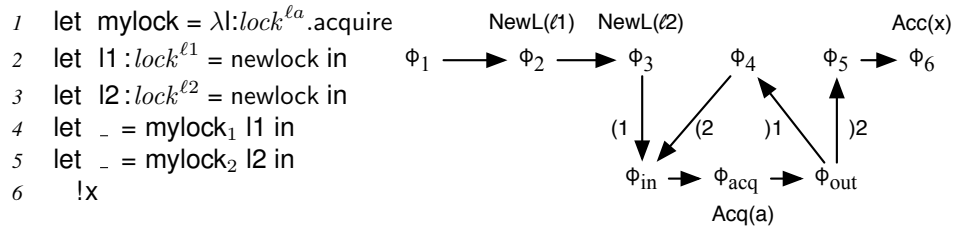
6.12.3 Context-sensitive dataflow analysis

We show now how we extend the dataflow analysis and ACFG with context sensitivity, encoding it also with parametric polymorphism.

As discussed above, each instantiation of a type scheme $\sigma = (\forall.\tau, \vec{\eta})$ at index i generates the constraint $\tau \preceq^i_+ \tau'$, where $\tau' = \langle\langle\tau\rangle\rangle$. We say that τ is the *abstract* type and τ' is the *instance* type at the instantiation i. Moreover, the instantiation i defines a substitution S_i of labels, such that $S_i(\tau) = \tau'$ and also for all labels $\eta \in \vec{\eta}$, we have $S_i(\eta) = \eta$. We represent the reverse substitution with S_i^{-1}, mapping the labels of τ' to τ. For example, the instantiation $ref^{\rho}(int) \preceq^i_+ ref^{\rho'}(int)$ defines the substitutions $S_i(\rho) = \rho'$ and $S_i^{-1}(\rho') = \rho$. Note that the substitutions S_i and S_i^{-1} only translate the labels between the abstract and the instance type, regardless of the instantiation polarity. So, even if the above instantiation had negative polarity, $ref^{\rho}(int) \preceq^i_- ref^{\rho'}(int)$, the defined substitutions remain $S_i(\rho) = \rho'$ and $S_i^{-1}(\rho') = \rho$.

Consider an instantiation of a function type according to Figure 6.27, $(\tau_1, \phi_1) \rightarrow (\tau_1', \phi_1') \preceq^i_+ (\tau_2, \phi_2) \rightarrow (\tau_2', \phi_2')$. This generates the constraints $\phi_1 \preceq^i_- \phi_2$ and $\phi_1' \preceq^i_+ \phi_2'$ among the statement labels representing the function start and end of the abstract and instance types. We extend the definition of dataflow analysis on the ACFG to account for the two kinds of instantiation edges, so that when we propagate facts across instantiation edges, they are brought to the correct context. Namely, we apply the substitution S_i to all facts propagated from ϕ_1' to ϕ_2', to translate all labels defined in the context of ϕ_1' to the corresponding labels in the context of ϕ_2'. Conversely, we apply the substitution S_i^{-1} to all facts propagated from ϕ_2 to ϕ_1 (recall that the negative instantiation polarity reverses the direction of the graph edge), to translate all facts in terms of the abstract type of the instantiation. Note that S_i is a partial map, so it is possible that not all facts defined at the left side of the instantiation can be expressed in terms of its right side, or vice versa. In general, we only propagate the facts that can be expressed at the target statement label of an instantiation edge.

Although these rules might resemble the Call and Ret kinds and edges in the control-flow graph, in fact they are orthogonal. Specifically, an instantiation edge between state-



(a) Source program (b) ACFG

Figure 6.28: Context-sensitive analysis for lock state

ment labels corresponds to an occurrence of a function name in the program, whereas statement labels with kind Call and Ret correspond to a function invocation. In many cases these happen to coincide, but one does not imply the other in general. For instance, when a program uses a function pointer to alias many functions and invokes it once, then many instantiations correspond to one invocation, whereas when the program assigns one function to a function pointer, but invokes it many times, then one instantiation has many invocations.

Lock State In the Lock State Analysis we propagate the set of acquired locks across instantiation edges as discussed above, by applying the appropriate renaming, according to the polarity of the constraint. Namely, for an instantiation $\phi \preceq^i_+ \phi'$ that corresponds to $\phi \rightarrow^{)i} \phi'$, we translate the set of acquired locks at ϕ by applying the substitution S_i, we close the translated set under aliasing, and we propagate the resulting set of acquired locks (and all their aliases) to statement ϕ:

$$Flow(C, S_i(Acq_{out}(\phi))) \subseteq Acq_{in}(\phi')$$

Similarly, for an instantiation $\phi \preceq^i_- \phi'$ that corresponds to $\phi' \rightarrow^{(i} \phi$, we translate the set of acquired locks at ϕ' by applying the substitution S_i^{-1} before propagating it to ϕ:

$$Flow(C, S_i^{-1}(Acq_{out}(\phi'))) \subseteq Acq_{in}(\phi)$$

For example, the program in Figure 6.28(a) defines a wrapper function for acquiring a lock that takes an argument a of type $lock^{\ell a}$ and acquires it. The program creates two locks and acquires them before dereferencing a variable x (not defined here, for brevity). Clearly, since the function mylock acquires its argument, the mechanism for "hiding" irrelevant locks using Call and Ret nodes has no effect here. Indeed, we need to differentiate between the two contexts of the calls to mylock (marked with indices 1 and 2) to infer that both locks l1 and l2 are acquired at the dereference point. We do this using the context-sensitive ACFG shown in Figure 6.28(b), simplified by omitting nodes of no interest for this example. During the dataflow analysis, we infer (as in the monomorphic case) that at the end of the function (ϕ_{out}) the abstract lock ℓ is acquired ($\ell \in Acq_{out}(\phi_{out})$). We also have $\phi_{out} \preceq^1_+ \phi_4$ and $\phi_{out} \preceq^2_+ \phi_5$. Moreover, from the instantiations 1 and 2 of mylock's type $(lock^{\ell a}, \phi_{in}) \rightarrow (int, \phi_{out})$, we have

$$\text{[EFF-LAM]}\frac{\Phi_f; \Gamma, x : \tau' \vdash e : \tau}{\Phi; \Gamma \vdash \lambda x.e : \tau' \rightarrow^{\Phi_f} \tau}$$

(a) Type rule for function definition

$$C \cup \{\tau_1 \rightarrow^{\Phi_1} \tau_1' \preceq_p^i \tau_2 \rightarrow^{\Phi_2} \tau_2'\} \Rightarrow C \cup \{\tau_1 \preceq_{\bar{p}}^i \tau_2, \tau_1' \preceq_p^i \tau_2', \Phi_1 \preceq_p^i \Phi_2\}$$

$$C \cup \{\Phi_1 \preceq_p^i \Phi_2\} \Rightarrow C \cup \{\Phi^\alpha \preceq_p^i \Phi^\alpha, \Phi^\omega \preceq_{\bar{p}}^i \Phi^\omega\}$$

(b) Constraint resolution rules

Figure 6.29: Context-Sensitive Contextual Effects

$S_1(\ell a) = \ell 1$ and $S_2(\ell a) = \ell 2$. To propagate the set of acquired locks along the instantiation edges $\phi_{out} \preceq_+^i \phi_4$, we apply the corresponding substitution to the set of acquired locks, propagating $S_1(Acq_{out}(\phi_{out}) = S_1(\{\ell a\}) = \{\ell 1\}$ to ϕ_4. Similarly, we propagate $S_2(Acq_{out}(\phi_{out}) = S_2(\{\ell a\}) = \{\ell 2\}$ to ϕ_5.

Correlation Inference We extend the correlation inference with context sensitivity in a similar way, adapted for a backwards analysis. Specifically, at instantiation $\phi \preceq_+^i \phi'$, due to the backwards direction of the propagation, we propagate from ϕ' to ϕ. Since ϕ' lies in the "instance" context, we use S_i^{-1} to translate the state at ϕ' to the state at ϕ. Namely, for every correlation $\rho \rhd \vec{\ell}$ at ϕ', we add a correlation $S_i^{-1}(\rho) \rhd Flow(C, S_i^{-1}(\vec{\ell}))$ to ϕ.

Likewise, for negative instantiation edges $\phi \preceq_-^i \phi'$, we propagate from ϕ to ϕ'. As in this case ϕ lies in the left side of the instantiation, we use S_i to translate the state at ϕ to the state at ϕ'. Now, for every correlation $\rho \rhd \vec{\ell}$ at ϕ, we add a correlation $S_i(\rho) \rhd Flow(C, S_i(\vec{\ell}))$ to ϕ.

6.12.4 Context-sensitive sharing analysis

We extended the sharing analysis with context sensitivity, both for computing the shared locations at fork points using context-sensitive contextual effects, and also for the flow-sensitive propagation of sharing information that marks the interesting dereferences in the program.

Contextual Effects The contextual effect system presented in Section 6.10.1 can be extended with context sensitivity in the same way as the label flow analysis. As presented in detail in previous work [113], function types are annotated with the effect Φ_f of the function. We repeat the type rule for function definition in Figure 6.29(a). Note that a function type is annotated with the effect Φ_f of the function body. Moreover, since function definition itself has no effect, it can be typed under any effect Φ. As in Section 6.10.1, we present contextual effects as a stand-alone system, although it is straightforward to combine with the rules in Figure 6.26. Figure 6.29(b) defines the instantiation for annotated

function types and contextual effects. The contextual effect of a function is instantiated covariantly, which translates to a covariant instantiation for the standard effect, and a contravariant instantiation for the future effect, because the standard effects of the function are defined inside the function and "returned" to the environment, whereas the future effect is defined outside the function, in the calling contexts, and "enters" the function.

Note that the future effect ω at a given program point in a function includes the effects of the program after the function returns. In combination with context sensitivity this might cause some locations that are in the effect (i.e. accessed after the current function returns) to not have a corresponding location, or not yet exist, in the current context. In other words, there might not be a matched parenthesis path from a location ρ, dereferenced in the continuation, to the future effect ω in the current location. For example, consider the toy program:

```
1  let f = λ x . x+1 in
2    f1 1;
3  let p = (ref 41) in
4    !p
```

Clearly, the variable p is not in scope in the body of function f, and moreover, there is no alias of p that is in scope either. This means that there can be no matched parentheses path from p to the future effect of expression $x + 1$ in the body of f. Indeed, the only path from p to the future effect of the expression involves a $(_1$ edge due to the instantiation of f. However, p is clearly in the future effect of the expression $x + 1$, as it is dereferenced later in the program, after the call to f. To address this problem, when solving for future effects at fork points to compute shared locations, we consider paths that do not contain mismatched parentheses (a.k.a. PN-flow [40]), instead of paths with only matched parentheses. For the same reason, we also use PN-flow to compute the set of labels in scope and their aliases for the scoping optimization discussed in Section 6.10.2.

Shared Locations Propagation The propagation of shared locations according to the dataflow analysis discussed in Section 6.10.3 is straightforward to extend with context sensitivity, in the same way as the above data-flow analyses for lock state and correlation inference. For positive instantiation edges, $\phi \preceq^i_+ \phi'$, we propagate from ϕ to ϕ' (forwards analysis), using S_i to translate the set of shared locations at ϕ to the context of ϕ' and adding the closed set (to account for aliasing) to the state at ϕ':

$$Flow(C, S_i(Sh_{out}(\phi))) \subseteq Sh_{out}(\phi')$$

Similarly, for negative instantiation edges $\phi \preceq^i_- \phi'$, we propagate from ϕ' to ϕ using S_i to translate the shared locations to the context of ϕ:

$$Flow(C, S_i^{-1}(Sh_{out}(\phi'))) \subseteq Sh_{out}(\phi)$$

6.12.5 Results

Figure 6.30 compares the running times and number of warnings for context-sensitive and -insensitive versions of LOCKSMITH. Note that since the context-sensitive analysis is no less sound than the context-insensitive analysis, any warning it eliminates is a

Benchmark	Context-sensitive		Context-insensitive	
	Time (s)	Warnings	Time (s)	Warnings
aget	0.85	62	0.64	77
ctrace	0.59	10	0.42	21
engine	0.88	7	0.60	15
knot	0.78	12	0.60	31
pfscan	0.46	6	0.50	26
smtprc	5.37	46	4.16	128
3c501	9.18	15	0.75	20
eql	21.38	35	0.86	41
hp100	143.23	14	2.76	25
plip	19.14	42	1.39	46
sis900	71.03	6	Out of Mem.	n/a
slip	16.99	3	Out of Mem.	n/a
sundance	106.79	5	1.32	20
synclink	1521.07	139	23.42	227
wavelan	19.70	10	9.59	143

Figure 6.30: Comparison of context-sensitivity and -insensitivity

false positive. The context-sensitive results are the same as Figure 6.4, reproduced here for convenience. These results show that context-sensitivity significantly increases the running time of the analysis, often very significantly, e.g., for most of the Linux drivers. The exceptions are the sis900 and slip benchmarks, for which the imprecision of context-insensitive analysis creates so much aliasing that LOCKSMITH runs out of memory trying to compute the closure of the label flow graph. Furthermore, we see that context sensitivity notably reduces the number of warnings reported by LOCKSMITH, eliminating many false positives.

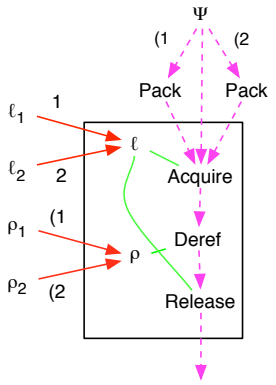
6.13 Existential quantification for data structures

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 section we briefly sketch an approach to solving this problem that has proven effective for one of our benchmarks. We present an analysis with support for existential quantification in detail in Chapter 4.

As an example, consider the program in Figure 6.31(a). This program first binds l1 and l2 to new locks labeled ℓ_1 and ℓ_2 respectively, 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

```
1  let l1 = newlock^ℓ1 in
2  let l2 = newlock^ℓ2 in
3  let x = ref^ρ1 1 in
4  let x = ref^ρ2 1 in
5  let p =
6    if b then
7      pack (l1, x)^1
8    else
9      pack (l2, y)^2
10 in
11   unpack (l, r) = p in
12     acquire l;
13     r := 3;
14     release l
```

(a) Source code



(b) Constraint graph

Figure 6.31: Existential Quantification

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, it performs *inference* on existential types using constraint resolution rules similar to those in Figure 6.27. Figure 6.31(b) shows the constraint graph for this 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, as discussed in Chapter 4.

In this figure, we represent dataflow 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 6.27—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 statement label ϕ. 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 6.27 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 pro-

grammer 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. We found 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.

Existential quantification in lock state Integrating the lock state analysis with existential quantification requires some care. In particular, existentially quantifying a lock gives two names, one abstract and one concrete, to the same linear location. However, recall that in our system it is always safe to assume that a lock is released. Hence whenever we unpack an existential type, we treat any locks in it as released. We also require as usual that existentially quantified labels do not escape the scope of the unpack.

Chapter 7

Related work

Race detection Several 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 [142] 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 [21, 117, 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 [42] for several languages, including Java [45], Java variants [15, 14], and Cyclone [64]. Such 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 [33, 46]. Static race detection in ESC/Java [53], 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 [46] can infer types for the original race-free Java system [45], but lacks context-sensitivity. More recently Agarwal and Stoller [2] and Rose et al [140] have developed algorithms that infer types based on dynamic traces, but these require sizeable test suites to avoid excessive false alarms. Flanagan and Freund [50] 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 [111]. Their system scales well to large Java programs and has found several races. 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 LOCKSMITH.

Several completely automatic static analyses have been developed for finding races in C code. Polyspace [76] 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 [73]. 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 [143] 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 set of benchmarks, finding several data races. It is unclear whether their analysis supports context sensitivity and how it models data structures. RacerX [33] does not soundly model some features of C for better scalability and to reduce false alarms, but may miss races as a result. KISS [130] builds on model checking techniques, and has been shown to find many races, but ignores possible thread interleavings, possibly missing the most subtle bugs.

Voung, Jhala and Lerner present RELAY, a race detection system for C [161] that uses flow sensitive propagation of lockset and guarded-by information similar to LOCKSMITH. RELAY scales to millions of lines of C code by making use of several machines that analyze parts of the program (using symbolic evaluation) in parallel and store the function summaries. Unlike RELAY, LOCKSMITH generates and solves the constraints for the whole program together, and is implemented to run on a single processor, limiting its scalability. One benefit of the whole program, type-based analysis in LOCKSMITH is that it can track the flow of function pointers precisely. In contrast, the modular per-file analysis in RELAY might not track aliasing of function pointers across files correctly, possibly resulting in lost control flow in certain cases.

Terauchi proposes LP-Race [156], a static analysis tool that reduces the problem of race detection to linear programming. The reduction is such that one need not directly compute acquired locks, and LP-race can handle synchronization via semaphores and signals. LP-Race scales to medium-sized programs, some of which cause LOCKSMITH to run out of memory. However, LOCKSMITH runs slightly faster though it uses a more precise aliasing and sharing analysis. We conjecture that, as a result of this more precise analysis, LOCKSMITH's reports are more precise—two abstract locations differentiated by a precise analysis could be considered to be one location in a less precise analysis. Note that both LOCKSMITH and LP-Race need to perform a linearity check for static lock variables, and might report false positives for programs with aliasing of run-time locks. LOCKSMITH's analysis is inclusion-based, and is both field- and context-sensitive. LP-Race uses a unification-based analysis, that is also field–sensitive. In one common benchmark (smtprc) LP-Race was able to eliminate false positives due to handling semaphores and thread joins. However, LP-Race produced additional false positives due to a limitation in handling loops that fork an unbounded number of threads. Due to the way we use future effects in our sharing analysis, LOCKSMITH is able to handle such loops and infer shared locations more precisely.

Work that detects violations of *atomicity*, either dynamically [47] or statically [54, 51] typically requires a program to be free of races.

Our analysis is based on ideas initially explored by Reps et al [135] and Rehof and Fähndrich [132], 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 [4, 36], which have similar requirements for non-escaping locations within a scope. Our flow-sensitive analysis is a significant extension of our

previous work on flow-sensitive type qualifiers [59], which used a similar flow-sensitive constraint graph. Both systems can be seen as inference for a variant of the calculus of capabilities [27].

Standard label flow and effect inference has been shown to be sound [99, 132], including polymorphic label flow inference. Recently, we (Iulian Neamtiu, Michael Hicks, Jeffrey Foster and I) have proved that our inference of continuation effects (as an instance of a more generic *Contextual Effect* system are sound [112].

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 [157]. Henglein et al [72] 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.

Other concurrency analyses Recent research proposes implementing atomic sections using optimistic concurrency techniques [67, 68, 74, 138, 164]. Roughly speaking, memory accesses within a transaction are logged. At the conclusion of the transaction, if the log is consistent with the current state of memory, then the writes are committed; if not, the transaction is rolled back and restarted. The main drawbacks with this approach are that first, it does not interact well with I/O, which cannot always be rolled back; second, performance can be worse than traditional pessimistic techniques due to the costs of logging and rollback [104].

Flanagan et al [51] have studied how to infer sections of Java programs that behave atomically, assuming that all synchronization has been inserted manually. Conversely, we assume the programmer designates the atomic section, and we infer the synchronization. Later work by Flanagan and Freund [49] looks at adding missing synchronization operations to eliminate data races or atomicity violations. However, this approach only works when a small number of synchronization operations are missing.

Existential context sensitivity Our work builds directly on the CFL reachability-based label flow system of Rehof et al [132]. Their cubic-time algorithm for polymorphic recursive label flow inference improves on the previously best-known $O(n^8)$ algorithm [109]. The idea of using CFL reachability in static analysis is due to Reps et al [135], who applied it to first-order dataflow analysis problems. Our contribution is to extend the use of CFL reachability further to include existential types for modeling data structures more precisely.

Existential types can be encoded in System F [121] (p. 377), in which polymorphism is first class and type inference is undecidable [165]. There have been several proposals to support first-class polymorphic type inference using type annotations to avoid the undecidability problem. In ML^{F} [13], programmers annotate function arguments that have universal types. Laufer and Odersky [90] propose an extension to ML with first-class existential types, and Remy [134] similarly proposes an extension with first-class universal types. In both systems, the programmer explicitly lists which type variables

are quantified. Packs and unpacks correspond to data structure construction and pattern matching, and hence are determined by the program text. Our system also requires the programmer to specify packs and unpacks as well as which variables are quantified, but in contrast to these three systems we support subtyping rather than unification, and thus we need polymorphically constrained types. Note that our solution is restricted to label flow, and only existential types are first-class, but we believe adding first-class universals with programmer-specified quantification would be straightforward. We conjecture that full first-class polymorphic type inference for label flow is decidable, and plan to explore such a system in future work.

Simonet [146] extends HM(X) [119], a generic constraint-based type inference framework, to include first-class existential and universal types with subtyping. Simonet requires the programmer to specify the polymorphically constrained type, including the subtyping constraints C, whereas we infer these (we assume we have the whole program). Another key difference is that we use CFL reachability for inference. Once again, however, our system is concerned only with label flow.

In ours and the above systems, both existential quantification as well as `pack` and `unpack` must be specified manually. An ideal inference algorithm requires no work from the programmer. For example, we envision a system in which all pairs and their uses are considered as candidate existential types, and the algorithm chooses to quantify only those labels that lead to a minimal flow in the graph. It is an open problem whether such an algorithm exists.

Contextual effects The original paper on contextual effects [113] presented the same type system and operational semantics shown in Sections 2 and 3, but placed scant emphasis on the details of the proof of soundness in favor of describing novel applications. Indeed, we felt that the proof technique described in the published paper was unnecessarily unintuitive and complicated, and that led us to ultimately discover the technique presented in this paper. To our knowledge, ours is the first mechanized proof of a property of typing and evaluation derivations that depends on the positions of subderivations in the super-derivation tree.

Type and effect systems [100, 116, 154] are widely used to statically enforce restrictions, check properties, or in static analysis to infer the behavior of computations [78, 147, 75, 150, 162]. Some more detailed comparisons with these systems can be found in our previous publication [113]. Talpin and Jouvelot [154] use a big-step operational semantics to prove standard effect soundness. In their system, operational semantics are not annotated with effects. Instead, the soundness property is that the static effect, unioned with a static description of the starting heap, describes the heap at the end of the computation. In addition to addressing contextual effects, our operational semantics can also be used as a definition of the *actual* effect (prior, standard, or future) of the computation, regardless of the static system used to infer or check effects. The soundness property for standard effects by Talpin and Jouvelot immediately follows for our system from Theorem 5.3.2.

Chapter 8

Future work

8.1 LOCKSMITH

There are many ways to improve the precision and performance of the analyses in LOCKSMITH.

8.1.1 Dependent types for array precision

Currently, some of the false positives in Locksmith are caused by assuming that all elements of an array are one and the same memory location. Recently, Condit et al have proposed mechanisms to add simple dependent types to CIL, that could increase the precision when analyzing arrays [24]. We believe that LOCKSMITH could benefit by incorporating these techniques into its data flow analysis.

8.1.2 Increased performance via parallelization

LOCKSMITH currently applies several analyses one after the other to the whole program. Moreover, the most common limitation in LOCKSMITH's scalability is memory usage. We believe it is worth exploring parallelization as a solution to these problems. Specifically, restructuring the analyses in LOCKSMITH to run in parallel on several machines would greatly improve LOCKSMITH's scalability. The dependencies among the different analyses, as well as dependencies among different parts of the same program within a single analysis can make this parallelization a challenging yet interesting problem to solve. A similar scenario that could improve the scalability of the system is using procedure summaries to modularize the analysis, and thus support incremental changes to software without needing to reanalyze the whole program.

8.2 Existential quantification and dataflow analysis

Our system for existential context sensitivity in labelflow analysis currently requires the programmer to specify the labels that are existentially quantified in each existential type. An interesting open problem is inferring the quantified labels (and thus which types

could be existentials at all) automatically. Unfortunately, in the case where a label could be quantified by many quantifiers we were unable to find a simple algorithm that achieves the minimality of flow. In fact, we believe that the problem of inferring existentially quantified labels so that label flow is minimized is NP-complete. It is currently an open question to show the complexity of the problem and explore heuristic solutions.

8.3 Contextual effects

Our current system for contextual effects uses big-step operational semantics to define actual effects and state the soundness property. Unfortunately, big-step operational semantics can only reason about terminating program evaluations. We believe it is an challenging open problem to extend the soundness proof to non-terminating program evaluations using coinductive big-step semantics. It is also worth exploring the possible definitions for future effects for a program that does not terminate, as future effects might not be defined for every program point in this case.

Another interesting open problem that comes to light by our work in proving the soundness of contextual effects is the applicability of our proof technique to other non-compositional properties. We believe that there are other interesting problems that pose similar problems in their definition and proofs due to non-compositionality, and that our proof techniques could be applied in these cases.

Chapter 9

Conclusions

In conclusion, this dissertation presents LOCKSMITH, a tool that uses static analysis to automatically find data races in multi-threaded C programs. Motivated by LOCKSMITH, we formalize and prove the soundness of several general theoretical systems for static analysis, that we believe have a wide applicability. First, we present a system for inferring correlations context sensitively. Second, we present a label flow analysis with support for existential context sensitivity. Third, we present a type-and-effect system to infer contextual effects, that we used to compute memory locations shared among threads. We formalize each system and prove its soundness, and discuss the mechanization of the proof for contextual effects in the Coq proof assistant. Moreover, we present the implementation of each of these ideas in LOCKSMITH, and describe the optimizations that we found necessary to scale such a system to real-world applications. Finally, we thoroughly test LOCKSMITH on a wide set of benchmarks, and present results on the precision and performance of LOCKSMITH in general and each analysis in particular, and compare against several alternative approaches.

Overall, we support the thesis that data races can be detected automatically using sound static analysis, in a practical and efficient way.

Appendix A

Soundness proof for correlation inference

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 [109]; 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) [167]. 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 [132]. 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 reduction is presented in Section A.2.

A.1 $\lambda_{\triangleright}^{cp}$: Correlation with Polymorphically-Constrained Types

This section introduces $\lambda_{\triangleright}^{cp}$ and proves it sound.

A.1.1 Operational Semantics

We begin by formalizing the operational semantics for our source language (which can be found in Figure 3.2). The operational semantics is defined using a single-step reduction relation between expressions, as shown in Figure A.1. 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), $(\delta$-if$)$, $(\delta$-pair$)$, $(\delta$-let$)$, $(\delta$-fix$)$ are also standard. Rule $(\delta$-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

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

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

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

Figure A.1: Operational Semantics and Target Language Syntax Extensions

using a heap H, which is a map from run-time locations R to values v, and allocation $\mathsf{ref}\ v$ creates a fresh location $R \notin \mathit{dom}(H)$, 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 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 $!\,v^R[L]$ and $!\,v^R[L']$ then $L = L'$, and similarly for redexes $v^R := v'[L]$, 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.1.2 Typing

Typing judgments in $\lambda_{\triangleright}^{cp}$ 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 $\forall \vec{l}[C].\tau$ (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."

$$
\begin{array}{lrcll}
\text{types} & \tau & ::= & \mathit{int} \mid \tau \times \tau \mid \tau \rightarrow^{\varepsilon} \tau' \mid \mathit{lock}^{\ell} \mid \mathit{ref}^{\rho}(\tau) & \\
\text{lock labels} & \vartheta & ::= & \ell \mid L & \\
\text{location labels} & \varphi & ::= & \rho \mid R & \\
\text{label variables} & l, \beta & ::= & \ell \mid \rho & \\
\text{labels} & l & ::= & \vartheta \mid \varphi & \\
\text{effects} & \varepsilon & ::= & \emptyset \mid \{\ell\} \mid \varepsilon \uplus \varepsilon' \mid \varepsilon \cup \varepsilon' & \\
\text{polytypes} & \sigma & ::= & \forall \vec{l}[C].\tau & \\
\text{constraint sets} & C & ::= & \emptyset \mid \{c\} \mid C \cup C & \\
\text{constraints} & c & ::= & \varphi \leq \rho & \text{(location flow)} \\
 & & \mid & \rho \rhd \ell & \text{(correlation)} \\
 & & \mid & L \leq^{1} \ell & \text{(lock allocation)} \\
 & & \mid & \nu \vec{l}[C; \varepsilon] & \text{(encapsulated constraints)}
\end{array}
$$

Figure A.2: $\lambda_{\rhd}^{cp}$ Types and Constructors

The type and constraint language for $\lambda_{\rhd}^{cp}$ is shown in Figure A.2. 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 l, β). In our type rules, we use *substitutions* S that map label variables to labels.

Definition A.1.1 (Substitution) *We define a substitution S as a function from label variables $\vec{l}$ to labels l. We write dom(S) to denote those labels for which S is not the identity, and similarly write rng(S) as the image of S applied to dom(S).*

Intuitively, the type $\forall \vec{l}[C].\tau$ stands for any type $S(\tau)$ where $S(C)$ is satisfied, for any substitution S with $\mathit{dom}(S) = \vec{l}$. 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^{ℓ} 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 $\nu \vec{l}[C; \varepsilon]$ 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 A.1.2 (Bound and Free Labels) *We write $\mathit{fl}(\cdot)$ to denote those labels that are not bound in some structure $\cdot$, where $\cdot$ could be C, Γ, τ, ε, or σ. Figure A.3 gives a formal*

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

Figure A.3: Free Labels

definition. We write strip(c) *to "strip off" the binders of encapsulated constraints; i.e.,* $\mathit{strip}(\nu \vec{l}[C; \varepsilon]) = C$*, but* $\mathit{strip}(c) = c$ *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* $\mathit{bl}(C) = \mathit{fl}(\mathit{strip}^*(C)) \setminus \mathit{fl}(C)$*.*

The typing rules are shown in Figures A.4 (monomorphic rules) and Figure A.5 (polymorphic rules). Most of the monomorphic rules are standard. The [NEWLOCK] and [REF] rules construct values of types lock^{ℓ} and $\mathit{ref}^{\rho}(\tau)$, 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 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 A.5, 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

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

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

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

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

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

$$[\textsc{Ref}]\frac{C;\Gamma\vdash_{cp} e:\tau;\varepsilon}{C;\Gamma\vdash_{cp}\mathsf{ref}\ e:ref^{\rho}(\tau);\varepsilon} \qquad [\textsc{Newlock}]\frac{}{C;\Gamma\vdash_{cp}\mathsf{newlock}:lock^{\ell};\{\ell\}}$$

$$[\textsc{Deref}]\frac{C;\Gamma\vdash_{cp} e_1:ref^{\rho}(\tau);\varepsilon_1 \qquad C;\Gamma\vdash_{cp} e_2:lock^{\ell};\varepsilon_2 \qquad C\vdash\rho\rhd\ell}{C;\Gamma\vdash_{cp}!\,e_1e_2:\tau;\varepsilon_1\uplus\varepsilon_2}$$

$$[\textsc{Assign}]\frac{\begin{array}{cc}C;\Gamma\vdash_{cp} e_1:ref^{\rho}(\tau);\varepsilon_1 & C;\Gamma\vdash_{cp} e_2:\tau;\varepsilon_2\\ C;\Gamma\vdash_{cp} e_3:lock^{\ell};\varepsilon_3 & C\vdash\rho\rhd\ell\end{array}}{C;\Gamma\vdash_{cp} e_1:=e_2e_3:\tau;\varepsilon_1\uplus\varepsilon_2\uplus\varepsilon_3}$$

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

Figure A.4: $\lambda^{cp}_{\rhd}$ Monomorphic Typing Rules

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 $S(C')$ 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 [63, 99, 18]. 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 $\nu\vec{l}[C;\varepsilon]$ which we term *encapsulated*

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

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

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

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

Figure A.5: $\lambda_{\triangleright}^{cp}$ Polymorphic Typing Rules and [Down]

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 $\vec{l}$ 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 [163]).

With this rule, we introduce the idea of an *alpha-converting substitution*. This is a technical 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 A.1.3 (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 $\mathit{dom}(\alpha^{\vec{l}}) = bl(C)$ and $\mathit{rng}(\alpha^{\vec{l}}) \cap (\vec{l} \cup \mathit{dom}(\alpha^{\vec{l}}) \cup \mathit{fl}(C)) = \emptyset$ 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 $\nu\vec{l}[C';\varepsilon']$, 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

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

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

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

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

$$[\textsc{Sub-Ref}]\frac{C \vdash \rho_1 \leq \rho_2 \quad C \vdash \tau_1 \leq \tau_2 \quad C \vdash \tau_2 \leq \tau_1}{C \vdash ref^{\rho_1}(\tau_1) \leq ref^{\rho_2}(\tau_2)}$$

Figure A.6: $\lambda_{\triangleright}^{cp}$ Subtyping

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

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

$$[\textsc{Encap-Flow}]\frac{\nu\vec{l}[C_0;\varepsilon] \in C \quad C_0 \vdash C'}{C \vdash \nu\vec{l}[C';\varepsilon]} \qquad [\textsc{Correlate}]\frac{C \vdash \varphi \leq \rho \quad \rho \rhd \ell \in C}{C \vdash \varphi \rhd \ell}$$

Figure A.7: $\lambda_{\triangleright}^{cp}$ Constraint Logic

($\phi_\alpha^{\vec{l}'}(\vec{l})$) 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 A.4 uses the subtyping rules in Figure A.6. These rules are standard.

A.1.3 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 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 3.2.1 and 3.2.2 for clarity:

Definition A.1.4 (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 A.1.5 (Consistent Correlation) *A set of constraints C is* consistently correlated *if-f*
$\forall \varphi.\ |S(C, \varphi)| \leq 1$.

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 A.7. 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 A.8 defines a well-formedness judgment $\varepsilon \vdash_{ok} C \hookrightarrow C'; \vec{l}$ on constraints, whose "inputs" are ε and C. Ignoring the "outputs" we introduce the short form of well-formedness as follows:

Definition A.1.6 *We define $\varepsilon \vdash_{ok} C$ if there exist C', $\vec{l}$ such that $\varepsilon \vdash_{ok} C \hookrightarrow C'; \vec{l}$.*

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 $\vec{l}$ in the output, and that they must be disjoint from binders $\vec{\beta}$ within constraints C', as we have $\vec{l} \uplus \vec{\beta}$. [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,

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

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

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

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

Figure A.8: $\lambda^{cp}_{\rhd}$ Constraint Set Well-Formedness

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 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 A.1.7 (Well-formed Constraint Properties) *If $\varepsilon \vdash_{ok} C \hookrightarrow C'; \vec{l}$ then*

1. *$C' = \mathit{strip}^*(C)$ and $\vec{l} = \mathit{bl}(C)$.*
2. *$\varepsilon \vdash_{ok} \alpha^{\vec{l'}}(C) \hookrightarrow \phi^{\vec{l'}}_{\alpha}(C'); \phi^{\vec{l'}}_{\alpha}(\vec{l})$ 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 $\varepsilon' \vdash_{ok} C \hookrightarrow C'; \vec{l}$.*

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

We can show well-formed constraints are consistently correlated.

Lemma A.1.8 (Consistent Correlation) *If* $\varepsilon \vdash_{ok} C \hookrightarrow C'; \vec{l}$ *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 $\varepsilon \vdash_{ok} C \hookrightarrow C'; \vec{l}$. To prove the properties mentioning C (rather than C'), observe by the rules in Figure A.7 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 A.1.7(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 A.1.9 (Encapsulated Constraint Properties) *If* $\varepsilon \vdash_{ok} C \cup \{\nu\vec{l}[C_1;\varepsilon_1]\} \hookrightarrow C' \cup C_1'; \vec{\beta} \uplus \vec{\beta'}$ *where* $\alpha^{\vec{l'}}$ *is an alpha-converting substitution on* $\nu\vec{l}[C_1;\varepsilon_1]$ *with* $\vec{l'} \supseteq \varepsilon \cup \mathit{fl}(C')$ *then*

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

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

Case [LAB-REFL]. We have

$$[\text{LAB-REFL}]\frac{}{C' \cup C_1' \cup \phi_\alpha^{\vec{l'}}(C_1') \vdash \varphi \leq \varphi}$$

and thus $\rho = \varphi$.

- Assume $\varphi \in \mathit{fl}(C' \cup C_1')$:
  (1) We have $C' \cup C_1' \vdash \varphi \leq \varphi$ by [LAB-REFL].

(2) Assume $\varphi \in \mathit{fl}(C' \cup \phi_\alpha^{\vec{l'}}(C'_1))$. We can show that $\varphi \notin \mathit{dom}(\phi_\alpha^{\vec{l'}})$ which implies that $\phi_\alpha^{\vec{l'}}(\varphi) = \varphi$ and thus $C' \cup C'_1 \vdash \varphi \leq \varphi$ by [LAB-REFL], and the result follows by taking $\rho' = \varphi$. We prove $\varphi \notin \mathit{dom}(\phi_\alpha^{\vec{l'}})$ by contradiction. Suppose $\varphi \in \mathit{dom}(\phi_\alpha^{\vec{l'}})$, and thus $\varphi \in C'_1$. Since the $\mathit{dom}(\phi_\alpha^{\vec{l'}})$ and $\mathit{rng}(\phi_\alpha^{\vec{l'}})$ must be disjoint, the fact that $\varphi \in \mathit{fl}(C' \cup \phi_\alpha^{\vec{l'}}(C'_1))$ and $\varphi \in \mathit{fl}(C' \cup C'_1)$ 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 $\nu\vec{l}[C_1; \varepsilon_1]$ and elsewhere in the $C \cup \nu\vec{l}[C_1; \varepsilon_1]$, either bound separately or free. But this is impossible since $\varepsilon \vdash_{ok} C \cup \{\nu\vec{l}[C_1; \varepsilon_1]\}$ forbids such duplication.

- Assume $\varphi \in \mathit{fl}(C' \cup \phi_\alpha^{\vec{l'}}(C'_1))$; proofs of (3) and (4) mirror (1) and (2), above.

Case [LOC-FLOW]. We have

$$[\text{LOC-FLOW}]\frac{\varphi \leq \rho \in C' \cup C'_1 \cup \phi_\alpha^{\vec{l'}}(C'_1)}{C' \cup C'_1 \cup \phi_\alpha^{\vec{l'}}(C'_1) \vdash \varphi \leq \rho}$$

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) $\varphi \leq \rho \in \phi_\alpha^{\vec{l'}}(C'_1)$. We prove the desired conditions by cases:

1. Assume $\varphi \leq \rho \in C'$, which implies that $\rho, \varphi \notin \mathit{dom}(\phi_\alpha^{\vec{l'}})$ since by $\varepsilon \vdash_{ok} C \cup \{\nu\vec{l}[C_1; \varepsilon_1]\}$ binders cannot be duplicated. As a result, we easily have $C' \vdash \varphi \leq \rho$ and $C' \vdash \varphi \leq \phi_\alpha^{\vec{l'}}(\rho)$ 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 $\rho \in \mathit{fl}(C' \cup \phi_\alpha^{\vec{l'}}(C'_1))$, then we can prove that $\rho \notin \mathit{dom}(\phi_\alpha^{\vec{l'}})$, so $\phi_\alpha^{\vec{l'}}(\rho) = \rho$ and thus $C'_1 \vdash \varphi \leq \phi_\alpha^{\vec{l'}}(\rho')$ where $\rho' = \rho$ by [LOC-FLOW]; the result follows by weakening. To prove $\rho \notin \mathit{dom}(\phi_\alpha^{\vec{l'}})$, there are two cases. If $\rho \in \mathit{fl}(C')$ then $\varepsilon \vdash_{ok} C \cup \{\nu\vec{l}[C_1; \varepsilon_1]\}$ prevents a binder in C_1 from being duplicated. If $\rho \in \mathit{fl}(\phi_\alpha^{\vec{l'}}(C'_1))$ then we follow the argument from (2) of the [LAB-REFL] case, above.

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

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

3. Assume $\varphi \leq \rho \in \phi_\alpha^{\vec{l'}}(C_1')$. 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

$$[\text{LOC-TRANS}]\frac{C' \cup C_1' \cup \phi_\alpha^{\vec{l'}}(C_1') \vdash \varphi \leq \rho' \qquad C' \cup C_1' \cup \phi_\alpha^{\vec{l'}}(C_1') \vdash \rho' \leq \rho}{C' \cup C_1' \cup \phi_\alpha^{\vec{l'}}(C_1') \vdash \varphi \leq \rho}$$

To prove (1)–(4), we consider two cases: (1) when $\rho' \in \mathit{fl}(C' \cup C_1')$ and (2) when $\rho' \in \mathit{fl}(C' \cup \phi_\alpha^{\vec{l}}(C_1')$. 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 $\rho \in \mathit{fl}(C' \cup \phi_\alpha^{\vec{l'}}(C_1'))$. By induction we have $C' \cup C_1' \vdash \varphi \leq \rho'$ and $C' \cup C_1' \vdash \rho' \leq \rho''$ where $\phi_\alpha^{\vec{l'}}(\rho'') = \rho$, and the result follows by [LOC-TRANS].

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

(4) $\varphi \in \mathit{fl}(C' \cup \phi_\alpha^{\vec{l'}}(C_1'))$ and $\rho \in \mathit{fl}(C' \cup C_1')$ so we must prove $C' \cup \phi_\alpha^{\vec{l'}}(C_1') \vdash \varphi \leq \phi_\alpha^{\vec{l'}}(\rho)$. By induction we have $C' \cup \phi_\alpha^{\vec{l'}}(C_1') \vdash \varphi \leq \phi_\alpha^{\vec{l'}}(\rho')$ and $C' \cup C_1' \vdash \rho' \leq \rho$. To get the desired result by [LOC-TRANS] we need to show $C' \cup \phi_\alpha^{\vec{l'}}(C_1') \vdash \phi_\alpha^{\vec{l'}}(\rho') \leq \phi_\alpha^{\vec{l'}}(\rho)$, but this follows by the same reasoning as case (3), above.

When assuming $\rho' \in fl(C' \cup \phi_\alpha^{\vec{l'}}(C_1'))$, the reasoning mirrors the cases above $\square$

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

Lemma A.1.10 (Duplicated Encapsulated Constraint)
If

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

then

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

where $\vec{l'} \supseteq \mathit{fl}(C_1') \cup \mathit{fl}(C') \cup \varepsilon$.

Proof: By inversion, we have

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

We want to prove

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

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

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

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

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

We have by assumption that $\varepsilon \vdash_{ok} \nu\vec{l}[C_1;\varepsilon_1] \hookrightarrow C_1'; \vec{\beta}$ and so $\varepsilon \vdash_{ok} \alpha^{\vec{l'}}(\nu\vec{l}[C_1;\varepsilon_1]) \hookrightarrow \phi_\alpha^{\vec{l'}}(C_1'); \phi_\alpha^{\vec{l'}}(\vec{\beta})$ by Lemma A.1.7(2). The final rule of this derivation must be [Con-Encap], so by inversion we have $\varepsilon \uplus \phi_\alpha^{\vec{l'}}(\varepsilon_1) \vdash_{ok} \mathit{strip}(\alpha^{\vec{l'}}(\nu\vec{l}[C_1;\varepsilon_1])) \hookrightarrow \phi_\alpha^{\vec{l'}}(C_1'); \phi_\alpha^{\vec{l'}}(\vec{\beta})$ which is the desired result.

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

Follows since $\vec{\beta} = \mathit{dom}(\phi_\alpha^{\vec{l'}})$ by Lemma A.1.7(1), and $\mathit{rng}(\phi_\alpha^{\vec{l'}}) \cap \mathit{fl}(\mathit{strip}^*(C \cup \{\nu\vec{l}[C_1;\varepsilon_1]\})) = \emptyset$ by the definition of $\vec{l'}$, and $\mathit{strip}^*(C \cup \{\nu\vec{l}[C_1;\varepsilon_1]\}) = C \cup C_1'$ by Lemma A.1.7(1).

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

We know $bl(C) = \vec{l}$ (thus $\vec{l} \subseteq \mathit{fl}(C')$) and $bl(\nu\vec{l}[C_1;\varepsilon_1]) = \mathit{dom}(\phi_\alpha^{\vec{l'}}) = \vec{\beta}$ (thus $\vec{\beta} \subseteq \mathit{fl}(C_1')$) by Lemma A.1.7(1). Consider some $l \in \mathit{fl}(C_1')$. If $l \in \mathit{dom}(\phi_\alpha^{\vec{l'}})$ then

$\phi_\alpha^{\vec{l'}}(l) \notin (\vec{l} \uplus \vec{\beta})$ because by the fact that $\phi_\alpha^{\vec{l'}}$ derives from an alpha-converting substitution we have that $\mathit{rng}(\phi_\alpha^{\vec{l'}}) \cap (\vec{l'} \cup \mathit{dom}(\phi_\alpha^{\vec{l'}})) = \emptyset$ where $\vec{l'} \supseteq (\mathit{fl}(C') \cup \mathit{fl}(C'_1)) \supseteq (\vec{l} \uplus \vec{\beta})$. If $l \notin \mathit{dom}(\phi_\alpha^{\vec{l'}})$ then $\phi_\alpha^{\vec{l'}}(l) = l$ and $l \notin \vec{\beta}$. Moreover, $l \notin \vec{l}$ by our assumption $\varepsilon \vdash_{ok} C \cup \{\nu\vec{l}[C_1; \varepsilon_1]\} \hookrightarrow C' \cup C'_1; \vec{l} \uplus \vec{\beta}$ whose last rule must be [CON-UNION] by inversion, which contains the premise $\mathit{fl}(C'_1) \cap \vec{l} = \emptyset$.

(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 $\varepsilon \vdash_{ok} C \cup \{\nu\vec{l}[C_1; \varepsilon_1]\} \hookrightarrow C' \cup C'_1; \vec{l} \uplus \vec{\beta}$ (assumption) and Lemma A.1.8. The latter is true by the fact that $\varepsilon \vdash_{ok} C \cup \{\alpha^{\vec{l'}}(\nu\vec{l}[C_1; \varepsilon_1])\} \hookrightarrow C' \cup \phi_\alpha^{\vec{l'}}(C'_1); \vec{l} \uplus \phi_\alpha^{\vec{l'}}(\vec{\beta})$ (by Lemma A.1.7(2)) and Lemma A.1.8.

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 A.7. Now consider the premises of $C' \cup C'_1 \cup \phi_\alpha^{\vec{l'}}(C'_1) \vdash \varphi \rhd \ell_2$:

   (a) $C' \cup C'_1 \cup \phi_\alpha^{\vec{l'}}(C'_1) \vdash \varphi \leq \rho'$ 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 A.1.9(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) $C' \cup C'_1 \cup \phi_\alpha^{\vec{l'}}(C'_1) \vdash \varphi \leq \rho'$ and $\rho' \rhd \ell_2 \in C' \cup \phi_\alpha^{\vec{l'}}(C'_1)$. Since $\varphi \in \mathit{fl}(C' \cup C'_1)$ and $\rho' \in C' \cup \phi_\alpha^{\vec{l'}}(C'_1)$, by Lemma A.1.9(2)(2) we must have that $C' \cup C'_1 \vdash \varphi \leq \rho''$ where $\phi_\alpha^{\vec{l'}}(\rho'') = \rho'$. Also $\rho'' \rhd \ell'_2 \in C' \cup C'_1$ where $\phi_\alpha^{\vec{l'}}(\ell'_2) = \ell_2$ since $\rho' \rhd \ell_2 \in C' \cup \phi_\alpha^{\vec{l'}}(C'_1)$ by assumption. By Lemma A.1.9(1) $\phi_\alpha^{\vec{l'}}(\ell'_2) = \ell_2$ 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 $\rho \rhd \ell_1 \in C' \cup \phi_\alpha^{\vec{l'}}(C'_1)$. If $\varphi \neq \rho$ then $\rho \in \mathit{fl}(C' \cup C'_1)$ and since $\rho \in \mathit{fl}(C' \cup \phi_\alpha^{\vec{l'}}(C'_1))$, it must be that $\rho \notin \mathit{bl}(C_1) \cup \vec{l}$. But then by Lemma A.1.9(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 A.1.9(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, $\ell_2 \in \mathit{bl}(C_1) \cup \vec{l'}$ and so by Lemma A.1.9(1) we have $\varphi \in \mathit{bl}(C_1) \cup \vec{l'}$ 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 A.1.9(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 A.1.9(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 A.1.9(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 $\ell \notin bl(C_1) \cup \vec{l}$ 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 A.1.8 implies that $L_1 = L_2$.

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

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

A.1.4 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.

Weakening Lemmas

Definition A.1.11 (Constraint Entailment) *$C' \vdash C$ if-f $\forall c \in C.\ C' \vdash c$.*

Lemma A.1.12 (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 A.1.13 (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 A.1.14 (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 A.1.12 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 A.1.13. Here are the more interesting polymorphic cases.

Case [LET]. We have

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

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

Case [FIX]. We have

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

Then since $C' \vdash C$ and $C \vdash S(C'')$, we have $C' \vdash S(C'')$ by Lemma A.1.12. Thus we can apply [FIX] to yield $C'; \Gamma \vdash_{cp} \mathsf{fix}\ f.v : S(\tau_1); \varepsilon$.

Case [INST]. Similar to [FIX].

Case [DOWN]. We have

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

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

$$\text{[ENCAP-FLOW]}\frac{\nu\vec{l}[C_0; \varepsilon_1] \in C' \qquad C_0 \vdash C_1}{C' \vdash \nu\vec{l}[C_1; \varepsilon_1]}$$

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

$$C'' \cup \nu\vec{l}[C_0; \varepsilon_1] \cup \mathit{strip}(\alpha^{\vec{l'}}(\nu\vec{l}[C_0; \varepsilon_1])); \Gamma \vdash_{cp} e : \tau; \varepsilon \uplus \phi_\alpha^{\vec{l'}}(\varepsilon_1)$$

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

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

which satisfies our assumptions that

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

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

Lemma A.1.15 (Polymorphic constraint weakening in typing) *If* $C; \Gamma, f : \forall \vec{l}[C''].\tau \vdash e : \tau; \varepsilon$ *then* $C; \Gamma, f : \forall \vec{l}[C'' \cup C'].\tau \vdash e : \tau; \varepsilon$ *where* $C \vdash C'$ *and* $\mathit{fl}(C') \cap \vec{l} = \emptyset$.

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

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

The result follows trivially. Otherwise, we have

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

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

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

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

The result follows trivially by induction. □

Substitution

Lemma A.1.16 (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 A.1.14 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

$$[\text{APP}]\frac{C; \Gamma, x : \tau' \vdash_{cp} e_1 : \tau_2 \to^{\varepsilon} \tau; \varepsilon_1 \quad C; \Gamma, x : \tau' \vdash_{cp} e_2 : \tau_2; \varepsilon_2}{C; \Gamma, x : \tau' \vdash_{cp} e_1\ e_2 : \tau; \varepsilon_1 \uplus \varepsilon_2 \uplus \varepsilon}$$

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

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

By Lemma A.1.14 and induction, we have $C' \cup C''; \Gamma \vdash_{cp} v_1[x \mapsto e'] : \tau_1; \emptyset$. By Lemma A.1.15 we have $C; \Gamma, x : \tau', f : \forall \vec{l}[C'' \cup C'].\tau_1 \vdash_{cp} e_2 : \tau_2; \varepsilon_2$, and by alpha-conversion (since $f \neq x$) and induction we have $C; \Gamma, f : \forall \vec{l}[C' \cup C''].\tau_1 \vdash e_2[x \mapsto e'] : \tau_2; \varepsilon_2$. We have

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

So the result follows by [LET].

Case [FIX]. We have

$$[\text{FIX}]\frac{\begin{array}{c} C''; \Gamma, x : \tau', f : \forall \vec{l}[C''].\tau \vdash_{cp} v : \tau; \emptyset \\ \vec{l} \subseteq (\mathit{fl}(\tau) \cup \mathit{fl}(C'')) \setminus \mathit{fl}(\Gamma, x : \tau') \\ C \vdash S(C'') \qquad \mathit{dom}(S) = \vec{l} \end{array}}{C; \Gamma, x : \tau' \vdash_{cp} \mathsf{fix}^{i}\ f.vS(\tau); \emptyset}$$

By Lemma A.1.14 and Lemma A.1.15 we have $C'' \cup C'; \Gamma, x : \tau', f : \forall \vec{l}[C'' \cup C'].\tau \vdash_{cp} v : \tau; \emptyset$, so by alpha-conversion (since $f \neq x$) and induction we have $C'' \cup C'; \Gamma, f : \forall \vec{l}[C'' \cup C'].\tau \vdash_{cp} v[x \mapsto e'] : \tau; \emptyset$. As per the reasoning in [LET], $\vec{l} \subseteq (\mathit{fl}(\tau_1) \cup \mathit{fl}(C'' \cup C')) \setminus \mathit{fl}(\Gamma)$. By our alpha-renaming convention, we have $\mathit{fl}(C') \cap \vec{l} = \emptyset$, so $S(C') = C'$ and thus $C \vdash S(C'' \cup C')$ since $C \vdash C'$ and $C \vdash S(C'')$. The result follows from [FIX].

Case [INST]. We have

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

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

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

Since $C \cup \{\nu\vec{l}[C_1;\varepsilon_1]\} \vdash C'$ by assumption, it follows that

$$C \cup \{\nu\vec{l}[C_1;\varepsilon_1]\} \cup \mathit{strip}(\alpha^{\vec{l'}}(\nu\vec{l}[C_1;\varepsilon_1])) \vdash C'$$

By induction

$$C \cup \{\nu\vec{l}[C_1;\varepsilon_1]\} \cup \mathit{strip}(\alpha^{\vec{l'}}(\nu\vec{l}[C_1;\varepsilon_1])); \Gamma \vdash_{cp} e[x \mapsto e']:\tau;\varepsilon \uplus \phi_\alpha^{\vec{l'}}(\varepsilon_1)$$

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

Lemma A.1.17 (Polymorphic substitution lemma) *If* $C; \Gamma, f : \forall\vec{l}[C'].\tau' \vdash_{cp} e : \tau;\varepsilon$ *and* $C'; \Gamma \vdash_{cp} e' : \tau';\emptyset$ *where* $\vec{l} \cap fl(\Gamma) = \emptyset$ *then* $C; \Gamma \vdash_{cp} e[f \mapsto e'] : \tau;\varepsilon$.

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

Case [INT]. Trivial.

Case [LAM]. We have

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

By alpha conversion, we can assume $\vec{l} \cap fl(\tau_1) = \emptyset$ 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

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

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

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

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,

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

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

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

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

By assumption, $C'; \Gamma \vdash_{cp} e' : \tau'; \emptyset$ so we know $S(C'); S(\Gamma) \vdash_{cp} e' : S(\tau); \emptyset$. Since $\vec{l} \cap fl(\Gamma) = \emptyset$, we then have $S(C'); \Gamma \vdash_{cp} e' : S(\tau); \emptyset$. But $C \vdash S(C')$, and so by Lemma A.1.14, $C; \Gamma \vdash_{cp} e' : S(\tau); \emptyset$, and so we have shown the conclusion, since $f^i[f \mapsto e'] = e'$.

Case [DOWN]. We have

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

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

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 A.1.12). This ensures that correlations are consistent—each R is correlated to a single, unchanging lock L—across the entire evaluation derivation.

Definition A.1.18 (Valid Evaluation) *We write* $C \vdash e \longrightarrow e'$ *if-f* $e \equiv \mathbb{E}[!\, v^R[L]]$ *or* $e \equiv \mathbb{E}[v'^R := v[L]]$ *implies* $S_g(C, R) = \{L\}$.

Lemma A.1.19 (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 $\mathsf{ref}\ e$, and there are two possible reductions. In the first case, we have $\mathsf{ref}\ e \longrightarrow \mathsf{ref}\ e'$. By assumption, we have

$$[\text{REF}]\frac{C; \Gamma \vdash_{cp} e : \tau; \varepsilon}{C; \Gamma \vdash_{cp} \mathsf{ref}\ e : \mathit{ref}^\rho(\tau); \varepsilon}$$

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 $C'; \Gamma \vdash_{cp} \mathsf{ref}\ e' : \mathit{ref}^\rho(\tau); \varepsilon'$, and we also have $C' \vdash \mathsf{ref}\ e \longrightarrow \mathsf{ref}\ e'$ by applying the $\mathbb{E}[e] \longrightarrow \mathbb{E}[e']$ evaluation rule.

In the second case we have $\mathsf{ref}\ v \longrightarrow v^R$. 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 $C' \vdash \mathsf{ref}\ v \longrightarrow v^R$ 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 A.1.14. Finally, we can prove $\varepsilon' \vdash_{ok} C'$ as follows:

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

Most of the premises follow trivially.

To prove $\mathit{fl}(\{R \le \rho\}) \cap \vec{l} = \emptyset$, we observe that if $\rho \notin \mathit{fl}(C)$ then there are no conditions on its flow, so $\vec{l}$ (where $\vec{l} = \mathit{bl}(C)$) can be safely alpha-converted. Otherwise (if $\rho \in \mathit{fl}(C)$) ρ must not appear in $\vec{l}$ 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 A.1.8. Since $R \notin C''$ (by the fact that it was fresh), we have for all $\varphi \neq R$ 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 A.1.12 and A.1.14 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 A.1.7(5). Otherwise

$C_i = C \cup C''$ for some C'', so by induction and inversion we have

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

As argued above, we can show $\varepsilon_i \uplus \varepsilon_2 \uplus \varepsilon \vdash_{ok} C \hookrightarrow C'''; \vec{l}$, 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

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

Choose $C' = C$ and $\varepsilon' = \varepsilon$. By Lemma A.1.16, $C'; \Gamma \vdash_{cp} e_1[x \mapsto v] : \tau_2; \varepsilon'$. 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 $!\, e_1 e_2$, and the reasoning follows that of [App] for the inductive cases.

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

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

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 A.1.8(1). This implies $S(C', R) = \{\ell\}$ since $C' \vdash R \rhd \ell$:

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

We have $C' \vdash L \leq^1 \ell$ by assumption, and by Lemma A.1.8(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, $e \equiv \mathsf{newlock}$ and so $\mathsf{newlock} \longrightarrow [L]$ where $L \notin C$ since it's fresh. We have

$$[\textsc{Newlock}]\frac{}{C; \Gamma \vdash_{cp} \mathsf{newlock} : lock^{\ell}; \{\ell\}}$$

Let $C' = C \cup \{L \leq^1 \ell\}$ and $\varepsilon' = \emptyset$. Clearly $C' \supseteq C$ and since $((\varepsilon' = \emptyset) - (\varepsilon = \{\ell\})) = \emptyset$ we have $(\varepsilon' - \varepsilon) \cap fl(C) = \emptyset$. 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

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

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

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

We prove $\emptyset \vdash_{ok} C \hookrightarrow C''; \vec{l}$ by assumption and weakening (Lemma A.1.7(5)). We prove $fl(\{L \leq^1 \ell\}) \cap \vec{l} = \emptyset$ 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 $\varepsilon \vdash_{ok} C \hookrightarrow C''; \vec{l}$ and Lemma A.1.7(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, $\mathsf{let}\ f = v_1\ \mathsf{in}\ e_2 \longrightarrow (e_2[f \mapsto v_1])$, and

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

Let $C' = C$ and $\varepsilon' = \varepsilon$. Thus $(\varepsilon' - \varepsilon) \cap fl(C) = \emptyset$ 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 $\vec{l} \cap fl(\Gamma) = \emptyset$ by alpha-renaming, by Lemma A.1.17 we have

$C'; \Gamma \vdash e_2[f \mapsto e_1] : \tau_2; \varepsilon'$.

Case [FIX]. In this case $\mathsf{fix}\, f.v \longrightarrow v[f \mapsto \mathsf{fix}\, f.v]$ and

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

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 S that maps all labels in $\vec{l}$ to themselves, we can apply [FIX] to show

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

Finally, from these facts, and since we can assume that $\vec{l} \cap \mathit{fl}(\Gamma) = \emptyset$ by alpha-renaming, by Lemma A.1.17 we have $C'; \Gamma \vdash v[f \mapsto \mathsf{fix}\, f.v] : \tau; \emptyset$.

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

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

Since $\varepsilon \vdash_{ok} C \cup \{\nu\vec{l}[C_1; \varepsilon_1]\} \hookrightarrow C' \cup C_1'; \vec{l} \uplus \vec{\beta}$ by assumption (and inversion via [CON-UNION] and [CON-ENCAP]), we have

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

by Lemma A.1.10. Thus by induction there exists some C_i, ε_i s.t. $C_i \supseteq C \cup \{\nu\vec{l}[C_1; \varepsilon_1]\} \cup \mathit{strip}(\alpha^{\vec{l'}}(\nu\vec{l}[C_1; \varepsilon_1]))$; and $(\varepsilon_i - (\varepsilon \uplus \phi_\alpha^{\vec{l'}}(\varepsilon_1))) \cap \mathit{fl}(C \cup \{\nu\vec{l}[C_1; \varepsilon_1]\} \cup \mathit{strip}(\alpha^{\vec{l'}}(\nu\vec{l}[C_1; \varepsilon_1]))) = \emptyset$ and $L \leq^1 \ell \in (C_i - (C \cup \{\nu\vec{l}[C_1; \varepsilon_1]\} \cup \mathit{strip}(\alpha^{\vec{l'}}(\nu\vec{l}[C_1; \varepsilon_1])))) \Rightarrow \ell \in ((\varepsilon \uplus \phi_\alpha^{\vec{l'}}(\varepsilon_1)) - \varepsilon_i)$ 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 $C' \supseteq C \cup \{\nu\vec{l}[C_1; \varepsilon_1]\}$ and $\varepsilon' \vdash_{ok} C'$ and $C' \vdash e \longrightarrow e'$ and $C'; \Gamma \vdash_{cp} e' : \tau; \varepsilon'$ follow trivially.

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

Finally, we must show that $(\varepsilon_i - \varepsilon) \cap \mathit{fl}(C \cup \{\nu\vec{l}[C_1; \varepsilon_1]\}) = \emptyset$. 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) $\ell \cap \mathit{fl}(C \cup \{\nu\vec{l}[C_1;\varepsilon_1]\} \cup \mathit{strip}(\alpha^{\vec{l'}}(\nu\vec{l}[C_1;\varepsilon_1]))) = \emptyset$, and thus $\ell' \cap \mathit{fl}(C \cup \{\nu\vec{l}[C_1;\varepsilon_1]\}) = \emptyset$ 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 $\ell \notin \mathit{fl}(C \cup \{\nu\vec{l}[C_1;\varepsilon_1]\})$ since (1) $\ell' \in \mathit{dom}(\phi_\alpha^{\vec{l'}})$ by the fact that $\mathit{dom}(\phi_\alpha^{\vec{l'}}) = \vec{l}$ and $\ell' \in \varepsilon_1 \subseteq \vec{l}$; (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 $\vec{l'} \supseteq (\mathit{fl}(\mathit{strip}^*(C)) \cup \mathit{fl}(\mathit{strip}^*(\nu\vec{l}[C_1;\varepsilon_1])) \cup \varepsilon)) \supseteq \mathit{fl}(C \cup \{\nu\vec{l}[C_1;\varepsilon_1]\})$.

□

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 A.1.8 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.

$$[\textsc{Loc-Trans}]\frac{C \vdash_{cfl} \rho_0 \leq \rho_l \quad C \vdash_{cfl} \rho_1 \leq \rho_2}{C \vdash_{cfl} \rho_0 \leq \rho_2}$$

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

(a) Location and Lock Flow

$$[\textsc{Corr-Trans}]\frac{C \vdash_{cfl} \rho \leq \rho' \quad C \vdash_{cfl} \rho' \rhd \ell}{C \vdash_{cfl} \rho \rhd \ell}$$

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

(b) Correlation Flow

Figure A.9: Constraint Flow

A.2 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 3.5–3.7. To distinguish the two systems, we will use $\vdash_{cfl}$ to indicate derivations in $\lambda_\rhd$ and $\vdash_{cp}$ to indicate derivations in $\lambda^{cp}_\rhd$.

In order to reason about the lock and location resolution rules in Figure 3.9, we reformulate them as inference rules, as shown in Figure A.9. Recall that our constraint resolution rules use $C \vdash_{cfl} \mathit{escapes}(\ell, \vec{l})$ (defined on page 22). In words, we have $\mathit{escapes}(\ell, \vec{l})$ if ℓ is connected in any way to $\vec{l}$, either via an instantiation 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 26. 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 $\varepsilon\tau = \emptyset$ 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 3.2.1) 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 $\lambda_\rhd$ is slightly stronger than the definition from $\lambda^{cp}_\rhd$. In particular, consider the program shown in Figure A.10. In $\lambda_\rhd$, this program will not type check. The problem is that x is used once with l' directly and

```
1  let l1 = newlock in
2  let x = ref 0 in
3  let f l = (x :=^l 0) in
4  f l1;
5  x :=^l1 0
```

Figure A.10: Program showing a difference between $\lambda_{\triangleright}^{cp}$ and $\lambda_{\triangleright}$

once in the body of f, where l' is represented by the name l. Thus $\lambda_{\triangleright}$ generates two correlation constraints for this example: $C \vdash_{cfl} x \triangleright l'$ from the outer use, and $C \vdash_{cfl} x \triangleright l$ from the use within f (because of [CORR-MATCH] and the self-loop on x because it is global at the definition of f). Thus it appears that x is inconsistently correlated by our definition. However, this program will type check in $\lambda_{\triangleright}^{cp}$, because in [LET] in Figure A.5 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.

The problem here was that l' was passed as a parameter to f but x was used as a global. This is an unusual program—typically either both l' and x would be passed as arguments to f or neither would be. It is possible to modify $\lambda_{\triangleright}$ to allow the program in Figure A.10 to type check. In particular, we could extend $\lambda_{\triangleright}$ to only check correlation with respect to concrete locks—in this case, we would see that both l and l' can only correspond to the call to `newlock` on the first line, and hence there is consistent correlation. LOCKSMITH implements this approach (taking care to model wrappers around `newlock` precisely), but we omit it from $\lambda_{\triangleright}$ for simplicity.

Now we prove that derivations in $\lambda_{\triangleright}$ reduce to derivations in $\lambda_{\triangleright}^{cp}$. Our approach closely follows Rehof et al [132], and we omit details where they are the same.

Definition A.2.1 *Every application of [*INST*]*

$$[\text{INST}]\frac{C \vdash_{cfl} \tau \preceq_{+}^{i} \tau' \qquad C \vdash_{cfl} \vec{l} \preceq_{\pm}^{i} \vec{l}}{C; \Gamma, f : (\forall.\tau, \vec{l}) \vdash_{cfl} f^{i} : \tau'; \emptyset}$$

defines an *instantiation context* $\langle C, \vec{l}, \tau, S \rangle$, where S is the substitution given by instantiation i. (Instantiation i is represented by the two constraints $\tau \preceq_{+}^{i} \tau'$ and $\vec{l} \preceq_{\pm}^{i} \vec{l}$.)

Definition A.2.2 (Closure) *Let C be a set of $\lambda_{\triangleright}$ constraints. Then we define $C^* = \{\rho \leq \rho' \mid C \vdash_{cfl} \rho \leq \rho'\} \cup \{\rho \triangleright \ell \mid C \vdash_{cfl} \rho \triangleright \ell\}$, i.e., C^* is the closure of C with respect to the rules in Figure A.9. Note that we treat C^* as a set of $\lambda_{\triangleright}^{cp}$ constraints.*

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

Definition A.2.3 (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 3.9 with respect to C:

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

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

Lemma A.2.4 *If $C \vdash_{cfl} \varepsilon \leq \varepsilon'$, then $\varepsilon^* \subseteq \varepsilon'^*$.*

Lemma A.2.5 *If $C \vdash_{cfl} \varepsilon \leq_{\vec{l}} \varepsilon'$, then $\varepsilon^* \cap \{\ell \mid \textit{escapes}(\ell, \vec{l})\} \subseteq \varepsilon'^*$.*

Lemma A.2.6 *If $C \vdash_{cfl} \varepsilon \preceq^i \varepsilon'$, then there is a substitution S_i such that $S_i(\varepsilon) \subseteq \varepsilon'$.*

Proof: These three lemmas can be proven by observing that the rules in Figure 3.9 compute a valid solution to the effect constraints. □

Next we prove a lemma that we can use during the reduction of [INST] or [FIX] from $\lambda_{\triangleright}$ to $\lambda_{\triangleright}^{cp}$. This lemma shows that we can extend a substitution S from an instantiation context into a substitution $\hat{S}$ such that C^* is closed with respect to $\hat{S}$. This substitution $\hat{S}$ is the substitution we will ultimately choose for the $\lambda_{\triangleright}^{cp}$ versions of [INST] and [FIX]. We introduce a new kind of label $\rho \sqcup \rho'$, which stands for the union of two labels; a detailed discussion can be found elsewhere [38, 127].

Lemma A.2.7 *Let $\langle C, \vec{l}, \tau, S \rangle$ be an instantiation context (i.e., an occurrence on [INST] or [FIX]). Then $C^* \vdash_{cp} \hat{S}(C^*)$, where*

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

and

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

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

Proof: The standard proof [38, 127] of this lemma holds. We show some of the cases for correlation constraints. Suppose $\hat{S}(C^*) \vdash_{cp} \rho' \triangleright \ell'$. Then let ρ, ℓ be such that $C^* \vdash_{cp} \rho \triangleright \ell$, i.e., $\hat{S}(\rho) = \rho'$ and $\hat{S}(\ell) = \ell'$. We need to show that $C^* \vdash_{cp} \rho' \triangleright \ell'$. There are a total of nine cases, depending on ρ and ℓ.

1. Suppose $\rho \in \vec{l}$. Then $\rho' = \hat{S}(\rho) = \rho$.

(a) Suppose $\ell \in \vec{l}$. Then $\ell' = \hat{S}(\ell) = \ell$. Thus by assumption $C^* \vdash_{cp} \rho' \rhd \ell'$.

(b) Suppose $\ell \in \mathit{fl}(\tau) - \vec{l}$. Then $\ell' = \hat{S}(\ell) = S(\ell)$, and $C \vdash_{cfl} \ell \preceq^i \ell'$. Then since $\rho \in \vec{l}$, by [INST] we have $C \vdash_{cfl} \rho \preceq^i_{\pm} \rho$ and $\rho = \rho'$. Then by [CORR-MATCH] we have $C^* \vdash_{cp} \rho' \rhd \ell'$.

(c) Otherwise, $\hat{S}(\ell) = \emptyset$, so there is nothing to show.

2. Suppose $\rho \in \mathit{fl}(\tau) - \vec{l}$. Then $\rho' = S(\rho)$ where $C \vdash_{cfl} \rho \preceq^i_p \rho'$ for some p.

(a) Suppose $\ell \in \vec{l}$. Then $\ell' = \hat{S}(\ell) = \ell$ and by [INST] $C \vdash_{cfl} \ell \preceq^i \ell'$. But then by [CORR-MATCH] we have $C^* \vdash_{cp} \rho' \rhd \ell'$.

(b) Suppose $\ell \in \mathit{fl}(\tau) - \vec{l}$. Then $\ell' = \hat{S}(\ell) = S(\ell)$, and $C \vdash_{cfl} \ell \preceq^i \ell'$. Then by [CORR-MATCH] we have $C^* \vdash_{cp} \rho' \rhd \ell'$.

(c) Otherwise, $\hat{S}(\ell) = \emptyset$, so there is nothing to show.

3. The last cases follow by the reasoning similar to above plus the standard reasoning about intermediate locations [127].

□

Definition A.2.8 *For a $\lambda_{\rhd}$ derivation $\mathcal{D}$, let the ith occurrence of [DOWN] be*

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

Let

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

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

Finally, define

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

Lemma A.2.9 *Let $\langle C, \vec{l}, \tau, S\rangle$ be an instantiation context. Then $C^{**} \vdash_{cp} \hat{S}(C^{**})$.*

Proof: By Lemma A.2.7 we have $C^* \vdash_{cp} \hat{S}(C^*)$, and all other constraint systems in C^{**} contain no free variables. □

Definition A.2.10 *Define* $(\forall.\tau, \vec{l})^* = \forall \vec{l}[C^{**}].\tau$ *where* $\vec{l} = (\mathit{fl}(\tau) \cup \mathit{fl}(C^{**})) - \vec{l}$*, 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 A.2.11 *If* $C \vdash_{cfl} \rho \leq \rho'$ *then* $C^* \vdash_{cp} \rho \leq \rho'$.

Lemma A.2.12 *If* $C \vdash_{cfl} \rho \rhd \ell$ *then* $C^* \vdash_{cp} \rho \rhd \ell$.

Lemma A.2.13 *If* $C \vdash_{cfl} \tau \leq \tau'$ *then* $C^* \vdash_{cp} \tau \leq \tau'$.

Proof: The proofs of all three statements are trivial. The proof of the last statement uses Lemma A.2.4 to show that the effect constraints from [SUB-FUN] in Figure 3.7 can be translated to $\subseteq$ conditions for [SUB-FUN] in Figure A.6. □

Lemma A.2.14 *Given a normal* $C; \Gamma \vdash_{cfl} e : \tau; \varepsilon$ *that is consistently correlated, we have* $\varepsilon^* \vdash_{ok} C^{**}$

Proof: We show that the rules in Figure A.8 apply. First, we can ignore [CON-LOCK], the $L \leq^1 \ell$ hypothesis of [CON-ENCAP], 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 A.2.8 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 A.2.8. And we need to show that $\varepsilon' \subseteq \vec{l}$, but that holds again by construction in Definition A.2.8.

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 $\mathit{strip}^*(C^{**})$, i.e., that

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

But since we assumed C was consistently correlated, $|S(C^*, \rho)| \leq 1$ for all ρ. Therefore for any i we have $|S(C^*|_{\vec{l_i}})| \leq 1$ 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 A.2.15 (Reduction) *If* $\mathcal{D}$ *is a normal derivation of* $C; \Gamma \vdash_{cfl} e : \tau; \varepsilon$*, then* $C^{**}; \Gamma^* \vdash_{cp} e : \tau; \varepsilon^*$.

Proof: By induction on the structure of the derivation $\mathcal{D}$. The cases for the monomorphic rules follow by induction and Lemmas A.2.11, A.2.12, and A.2.13.

Case [LET]. We have

$$[\textsc{Let}]\frac{C;\Gamma \vdash_{cfl} v_1 : \tau_1;\emptyset \qquad \vec{l} = \mathit{fl}(\Gamma) \qquad C;\Gamma, f : (\forall.\tau_1, \vec{l}) \vdash_{cfl} e_2 : \tau_2;\varepsilon}{C;\Gamma \vdash_{cfl} \mathsf{let}\ f = v_1\ \mathsf{in}\ e_2 : \tau_2;\varepsilon}$$

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

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

Case [INST]. We have

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

We want to show

$$[\textsc{Inst}]\frac{C^{**} \vdash_{cp} \hat{S}(C^{**}) \qquad \mathit{dom}(\hat{S}) = \vec{l}}{C^{**};\Gamma^*, f : \forall\vec{l}[C^{**}].\tau \vdash_{cp} f^i : \hat{S}(\tau);\emptyset}$$

We apply Lemma A.2.9 to show that $C^{**} \vdash_{cp} \hat{S}(C^{**})$, where S is the substitution defined by this instantiation. We have $\mathit{dom}(\hat{S}) = \vec{l}$ by construction of $\hat{S}$ and choice of $\vec{l}$. And $\hat{S}(\tau) = S(\tau)$, by definition of $\hat{S}$, so the type of f^i is what we expect.

Case [FIX]. We have

$$[\textsc{Fix}]\frac{\begin{array}{c}C;\Gamma, f : (\forall.\tau, \vec{l}) \vdash_{cfl} v : \tau';\emptyset \qquad \vec{l} = \mathit{fl}(\Gamma) \qquad C \vdash_{cfl} \tau' \leq \tau \qquad C \vdash_{cfl} \tau \preceq^i_+ \tau'' \\ C \vdash_{cfl} \vec{l} \preceq^i_\pm \vec{l} \qquad\qquad C \vdash_{cfl} \varepsilon\tau = \emptyset\end{array}}{C;\Gamma \vdash_{cfl} \mathsf{fix}^{:} f.v\tau'';\emptyset}$$

By induction, we have $C^{**};\Gamma^*, f : \forall\vec{l}[C^{**}].\tau \vdash_{cp} v : \tau';\emptyset$, where by construction $\vec{l} = (\mathit{fl}(\tau) \cup \mathit{fl}(C^{**})) - \vec{l}$. (Note that we have $\mathit{fl}(\Gamma) = \mathit{fl}(\Gamma^*)$ by construction of Γ^*.) Since $C \vdash_{cfl} \tau' \leq \tau$, by [SUB] and Lemma A.2.13 we have $C^{**};\Gamma^*, f : \forall\vec{l}[C^{**}].\tau \vdash_{cp} v : \tau;\emptyset$. By Lemma A.2.9 we have $C^{**} \vdash_{cp} \hat{S}(C^{**})$, where S is the instantiation defined by this substitution. We have $\mathit{dom}(\hat{S}) = \vec{l}$ by construction of $\hat{S}$ and choice of $\vec{l}$. Putting this together, we get

$$[\textsc{Fix}]\frac{\begin{array}{c}C^{**};\Gamma^*, f : f : \forall\vec{l}[C^{**}].\tau \vdash_{cp} v : \tau;\emptyset \\ \vec{l} \subseteq (\mathit{fl}(\tau) \cup \mathit{fl}(C^{**})) \setminus \mathit{fl}(\Gamma) \qquad C \vdash_{cp} S(C') \qquad \mathit{dom}(\hat{S}) = \vec{l}\end{array}}{C;\Gamma \vdash_{cp} \mathsf{fix}^{:} f.v\hat{S}(\tau);\emptyset}$$

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

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

$$[\text{DOWN}]\frac{C; \Gamma_i \vdash_{cfl} e : \tau_i; \varepsilon_i \qquad \vec{l} = \mathit{fl}(\Gamma_i) \cup \mathit{fl}(\tau_i) \qquad C \vdash_{cfl} \varepsilon_i \leq_{\vec{l_i}} \chi_i}{C; \Gamma_i \vdash_{cfl} e : \tau_i; \chi_i}$$

By induction, we have

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

Let $\vec{\ell_i}$, $\vec{l_i}$, and $\nu\vec{l_i}[C^*|_{\vec{l_i}}; \vec{\ell_i}]$ be as in Definition A.2.8. Let S be the alpha-renaming such that $S(\vec{l_i}) = \vec{l}$, where $\nu\vec{l}[C'; \varepsilon']$ is the constraint in C^{**}.

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

$$\varepsilon' = S(\vec{\ell_i}) \subseteq S(\vec{l_i}) = \vec{l} \tag{A.1}$$

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

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

or

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

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

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

Then by Lemma A.1.14 and the definition of C' (in Definition A.2.8) 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 A.1.3. Also construct $\phi_\alpha^{\vec{l'}}$ such that $\phi_\alpha^{\vec{l'}}(\vec{l}) \cap (\mathit{fl}(\Gamma_i) \cup \mathit{fl}(\tau_i)) = \emptyset$. 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 $S(\vec{l})$, which do not appear in Γ_i, τ_i, or χ_i^* by choice of the alpha renaming S in Definition A.2.8. By applying appropriate alpha conversions to the bound constraint systems in C', we get

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

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

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

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

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

And nothing that C^{**} includes $C_i = \nu\vec{l}[C';\varepsilon']$, and hence has the right shape. Thus we have shown the conclusion. □

Appendix B

Soundness proof for existential label flow analysis

B.1 Soundness proof for $\lambda_{\exists}^{cp}$

We prove soundness for $\lambda_{\exists}^{cp}$ using a standard subject-reduction style approach. We begin by proving a number of helpful lemmas. First, we need to show that it is sound in several places to weaken a constraint system C to a constraint system C' where $C' \vdash C$. Intuitively this holds because C' contains all the "flows" of C, hence all typing judgments are preserved.

Lemma B.1.1 *If $C' \vdash C$ then $C; D \vdash l \leq l'$ implies $C'; D \vdash l \leq l'$*

Proof: By definition, $C' \vdash C$ requires $C \subseteq C'$. There are two possible ways we could have shown $C; D \vdash l \leq l'$:

Case [SUB-LABEL-1].

$$\text{[SUB-LABEL-1]}\frac{D(l) = D(l') = 0 \qquad C \vdash l \leq l'}{C; D \vdash l \leq l'}$$

then $\{l \leq l'\} \subseteq C'$, and we hence we can apply [SUB-LABEL-1] to show $C'; D \vdash l \leq l'$.

Case [SUB-LABEL-2].

$$\text{[SUB-LABEL-2]}\frac{D(l) > 0}{C; D \vdash l \leq l}$$

Obviously, [SUB-LABEL-2] can be applied for any C, so we also have $C'; D \vdash l \leq l$. $\square$

Lemma B.1.2 (Constraint weakening in subtyping) *If $C; D \vdash \tau \leq \tau'$ then for any C' such that $C' \vdash C$ it holds that $C'; D \vdash \tau \leq \tau'$*

Proof: By induction on the proof derivation of $C; D \vdash \tau \leq \tau'$.

Case [SUB-INT]. By assumption, we have

$$[\text{SUB-INT}]\frac{C; D \vdash l \leq l'}{C; D \vdash intl \leq intl'}$$

Then from Lemma B.1.1 we get $C'; D \vdash l \leq l'$ so applying [SUB-INT] again we have $C'; D \vdash intl \leq intl'$.

Case [SUB-PAIR]. By assumption, we have

$$[\text{SUB-PAIR}]\frac{\begin{array}{c} C; D \vdash l \leq l' \\ C; D \vdash \tau_1 \leq \tau_1' \\ C; D \vdash \tau_2 \leq \tau_2' \end{array}}{C; D \vdash \tau_1 \times^l \tau_2 \leq \tau_1' \times^{l'} \tau_2'}$$

From Lemma B.1.1 we have $C'; D \vdash l \leq l'$ and by induction we have $C'; D \vdash \tau_1 \leq \tau_1'$ and $C'; D \vdash \tau_2 \leq \tau_2'$. Then applying [SUB-PAIR] again, we can show $C'; D \vdash \tau_1 \times^l \tau_2 \leq \tau_1' \times^{l'} \tau_2'$.

Case [SUB-FUN]. Similar to [SUB-PAIR].

Case [SUB-$\exists$]. By assumption, we have

$$[\text{SUB-}\exists]\frac{\begin{array}{c} C_1 \vdash C_2 \\ D' = D[l \mapsto D(l) + 1, \forall l \in \vec{\alpha}] \\ C; D' \vdash \tau_1 \leq \tau_2 \\ C; D \vdash l_1 \leq l_2 \end{array}}{C; D \vdash \exists^{l_1} \vec{\alpha}[C_1].\tau_1 \leq \exists^{l_2} \vec{\alpha}[C_2].\tau_2}$$

From Lemma B.1.1 we have $C'; D \vdash l_1 \leq l_2$. By the induction hypothesis, we have $C'; D \vdash \tau_1 \leq \phi(\tau_2)$ so we can apply [SUB-$\exists$] to prove $C'; D \vdash \exists \vec{\alpha}[C_1].\tau_1 \leq \exists \vec{\alpha}[C_2].\tau_2$

□

Lemma B.1.3 (Constraint weakening in judgment) *If $C; \Gamma \vdash e : \tau$ and $C' \vdash C$ then $C'; \Gamma \vdash e : \tau$*

Proof: By induction on the derivation of $C; \Gamma \vdash e : \tau$. First, observe that if $C \vdash l \leq l'$, then $C' \vdash l \leq l'$, by definition of $C' \vdash C$.

Case [ID], [INT], [APP], [LAM], [PAIR], [PROJ], and [COND]. The first case is trivial. For the others, apply induction and observe that $C' \vdash L \leq l$ or $C' \vdash l \leq L$, as appropriate.

Case [SUB]. We have

$$[\text{SUB}]\frac{C; \Gamma \vdash_{cp} e : \tau \qquad C; \emptyset \vdash \tau \leq \tau'}{C; \Gamma \vdash_{cp} e : \tau'}$$

Apply induction, and observe that by Lemma B.1.2, $C'; \emptyset \vdash \tau \leq \tau'$.

Case [LET]. We have

$$[\text{LET}]\frac{C''; \Gamma \vdash_{cp} e_1 : \tau_1 \qquad C; \Gamma, f : \forall \vec{\alpha}[C''].\tau_1 \vdash_{cp} e_2 : \tau_2 \quad \vec{\alpha} \subseteq (fl(\tau_1) \cup fl(C'')) \setminus fl(\Gamma)}{C; \Gamma \vdash_{cp} \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

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

Case [FIX]. We have

$$[\text{FIX}]\frac{C''; \Gamma, f : \forall \vec{\alpha}[C''].\tau_1 \vdash_{cp} e : \tau_1 \qquad C \vdash C''[\vec{\alpha} \mapsto \vec{l}] \quad \vec{\alpha} \subseteq (fl(\tau_1) \cup fl(C'')) \setminus fl(\Gamma)}{C; \Gamma \vdash_{cp} \mathsf{fix}^{\cdot} f.e\tau_1[\vec{\alpha} \mapsto \vec{l}]}$$

Then since $C' \vdash C$ and $C \vdash C''[\vec{\alpha} \mapsto \vec{l}]$, we have $C' \vdash C''[\vec{\alpha} \mapsto \vec{l}]$. Thus we can apply [FIX] to yield $C'; \Gamma \vdash_{cp} \mathsf{fix}\, f.e : \tau_1[\vec{\alpha} \mapsto \vec{l}]$.

Case [INST]. We have

$$[\text{INST}]\frac{C \vdash C''[\vec{\alpha} \mapsto \vec{l}]}{C; \Gamma, f : \forall \vec{\alpha}[C''].\tau \vdash_{cp} f^i : \tau[\vec{\alpha} \mapsto \vec{l}]}$$

Then $C' \vdash C''[\vec{\alpha} \mapsto \vec{l}]$, and thus by [INST], $C'; \Gamma, f : \forall \vec{\alpha}[C''].\tau \vdash_{cp} f^i : \tau[\vec{\alpha} \mapsto \vec{l}]$.

Case [PACK]. We have

$$[\text{PACK}]\frac{C; \Gamma \vdash_{cp} e : \tau[\vec{\alpha} \mapsto \vec{l}] \quad C \vdash C''[\vec{\alpha} \mapsto \vec{l}] \qquad C \vdash L \leq l}{C; \Gamma \vdash_{cp} \mathsf{pack}^i\ e\ L : \exists \vec{\alpha}[C''].\tau'}$$

By induction, $C'; \Gamma \vdash_{cp} e : \tau[\vec{\alpha} \mapsto \vec{l}]$. Then $C' \vdash C''[\vec{\alpha} \mapsto \vec{l}]$. As before, we also have $C' \vdash L \leq l$. Thus by [PACK], $C'; \Gamma \vdash_{cp} \mathsf{pack}^i\ e\ L : \exists \vec{\alpha}[C''].\tau'$.

Case [UNPACK]. We have

$$[\text{UNPACK}]\frac{\begin{array}{c} C; \Gamma \vdash_{cp} e_1 : \exists \vec{\alpha}[C''].\tau_1 \qquad\qquad C \vdash l \leq L \\ C \cup C''; \Gamma, x : \tau_1 \vdash_{cp} e_2 : \tau \\ \vec{\alpha} \subseteq (fl(\tau_1) \cup fl(C'')) \setminus (fl(\Gamma) \cup fl(\tau) \cup fl(C)) \end{array}}{C; \Gamma \vdash_{cp} \mathsf{unpack}\ x = e_1\ \mathsf{in}\ e_2 L : \tau}$$

By induction, $C'; \Gamma \vdash_{cp} e_1 : \exists \vec{\alpha}[C''].\tau_1$. Since $C' \vdash C$, we have $C' \vdash l \leq L$ and $C' \cup C'' \vdash C \cup C''$. Thus also by induction, $C' \cup C''; \Gamma, x : \tau_1 \vdash_{cp} e_2 : \tau$. We can always apply alpha renaming to $\vec{\alpha}$ in C'' and τ_1 so that $fl(C') \cap \vec{\alpha} = \emptyset$, and therefore we

have $\vec{\alpha} \subseteq (fl(\tau_1) \cup fl(C'')) \setminus (fl(\Gamma) \cup fl(C') \cup fl(\tau))$. Thus we can apply [UNPACK] to show $C'; \Gamma \vdash_{cp} \mathsf{unpack}\ x = e_1\ \mathsf{in}\ e_2 L : \tau$. $\square$

The following lemma is useful for proving soundness of `unpack`. Intuitively, we will use this lemma to reason about subtyping step $C; D \vdash \exists^{l_1} \vec{\alpha}[C_1].\tau_1 \leq \exists^{l_2} \vec{\alpha}[C_2].\tau_2$. Specifically, it will allow us to derive $C; \emptyset \vdash \psi(\tau_1) \leq \psi(\tau_2)$ for a substitution $\psi()$ on $\vec{\alpha}$, because [SUB-LABEL-2] requires that any labels in $\vec{\alpha}$ have identical occurrences in τ_1 and τ_2,

Lemma B.1.4 *Let $D = D' \circ [l \mapsto 1, \forall l \in \vec{\alpha}])$, where dom$(D') \cap \vec{\alpha} = \emptyset$. Then if $C; D \vdash \tau_1 \leq \tau_2$ and dom$(\psi) = \vec{\alpha}$, then $C; D' \vdash \psi(\tau_1) \leq \psi(\tau_2)$*

Proof: Proof by induction on the derivation $C; D \vdash \tau_1 \leq \tau_2$.

Case [SUB-LABEL-1]. We have

$$[\text{SUB-LABEL-1}]\frac{D(l) = D(l') = 0 \qquad C \vdash l \leq l'}{C; D \vdash l \leq l'}$$

But then $l \notin$ *dom*(ψ) and $l' \notin$ *dom*(ψ), so $\psi(l) = l$ and $\psi(l') = l'$. But then since $C \vdash l \leq l'$, clearly $C; \emptyset \vdash \psi(l) \leq \psi(l')$.

Case [SUB-LABEL-2]. We have

$$[\text{SUB-LABEL-2}]\frac{D(l) > 0}{C; D \vdash l \leq l}$$

This case is trivial, since $C; \emptyset \vdash \psi(l) \leq \psi(l)$.

Case [SUB-PAIR]. We have

$$[\text{SUB-PAIR}]\frac{\begin{array}{c} C; D \vdash l \leq l' \\ C; D \vdash \tau_1 \leq \tau_1' \\ C; D \vdash \tau_2 \leq \tau_2' \end{array}}{C; D \vdash \tau_1 \times^l \tau_2 \leq \tau_1' \times^{l'} \tau_2'}$$

By induction, $C; \emptyset \vdash \psi(l) \leq \psi(l')$. Also by induction, $C; \emptyset \vdash \psi(\tau_i) \leq \psi(\tau_i')$. Hence by [SUB-PAIR], $C; \emptyset \vdash \psi(\tau_1 \times^l \tau_2) \leq \psi(\tau_1' \times^{l'} \tau_2')$.

Case [SUB-INT], [SUB-FUN]. Similar to [SUB-PAIR].

Case [SUB-$\exists$]. We have

$$[\text{SUB-}\exists]\frac{\begin{array}{c} C_1 \vdash C_2 \\ D'' = D[l \mapsto D(l) + 1, \forall l \in \vec{\beta}] \\ C; D'' \vdash \tau_1 \leq \tau_2 \\ C; D \vdash l_1 \leq l_2 \end{array}}{C; D \vdash \exists^{l_1} \vec{\beta}[C_1].\tau_1 \leq \exists^{l_2} \vec{\beta}[C_2].\tau_2}$$

By alpha conversion, we can assume that $\vec{\beta} \cap \mathit{dom}(\psi) = \emptyset$, $\vec{\beta} \cap fl(\mathit{rng}(\psi)) = \emptyset$, and $\vec{\beta} \cap \mathit{dom}(D') = \emptyset$. Notice that $D'' = (D'[l \mapsto D(l) + 1, \forall l \in \vec{\beta}]) \circ [l \mapsto 1, \forall l \in \vec{\alpha}]$. (The order doesn't matter because the domains of the substitutions are all different.) Then by induction, $C; D'[l \mapsto D(l) + 1, \forall l \in \vec{\beta}] \vdash \psi(\tau_1) \leq \psi(\tau_2)$. Further, since $C_1 \vdash C_2$, we have $\psi(C_1) \vdash \psi(C_2)$. Then since we assumed ψ did not replace or capture any variables in $\vec{\beta}$, by [SUB-$\exists$], we have $C; D' \vdash \psi(\exists^{l_1}\vec{\beta}[C_1].\tau_1) \leq \psi(\exists^{l_2}\vec{\beta}[C_2].\tau_2)$. $\square$

The next lemmas show that in a polymorphically constrained type, we can safely weaken the bound constraints C into C' where $C' \vdash C$. Because existential types are first-class in our system, changing the bound constraints may also change types τ on the right-hand side of a typing judgment.

Let χ range over quantifiers, either $\forall$ or $\exists$. We define $\mathit{polytypes}(\tau)$ to be the set $\{\chi_i\vec{\alpha_i}[C_i].\tau_i\}$ of occurrences of quantified types in τ. As a shorthand, we write χ_i for the ith element of this set, and we define $\chi_i\langle C'\rangle = \chi_i\vec{\alpha_i}[C' \cup C_i].(\tau_i\langle C'\rangle)$, i.e., we union C' with any bound constraint systems. We implicitly alpha rename bound type variables as necessary to avoid capturing variables in C', i.e., we assume $\vec{\alpha_i} \cap C' = \emptyset$. Here $\tau\langle C\rangle$ is τ where each $\chi_i \in \mathit{polytypes}(\tau)$ is replaced by $\chi_i\langle C\rangle$. We define $\mathit{polytypes}(\Gamma)$ to be the set of occurrences of quantified types in the range of Γ, and we define $\Gamma\langle C\rangle$ to be Γ with $\langle C\rangle$ applied to the range of Γ.

Lemma B.1.5 *If $C; D \vdash \tau \leq \tau'$, then $C; D \vdash \tau\langle C'\rangle \leq \tau'\langle C'\rangle$.*

Proof: By induction on the derivation of $C \vdash \tau \leq \tau'$. The [SUB-PAIR], [SUB-FUN], and [SUB-INT] cases are straightforward.

Case [SUB-$\exists$]. We have

$$[\text{SUB-}\exists]\frac{\begin{array}{c} C_1 \vdash C_2 \\ D' = D[l \mapsto D(l) + 1, \forall l \in \vec{\alpha_1}] \\ C; D' \vdash \tau_1 \leq \tau_2 \\ C; D \vdash l_1 \leq l_2 \end{array}}{C; D_1; D_2 \vdash \exists^{l_1}\vec{\alpha}[C_1].\tau_1 \leq \exists^{l_2}\vec{\alpha}[C_2].\tau_2}$$

By induction, we have $C; D' \vdash \tau_1\langle C'\rangle \leq \tau_2\langle C'\rangle$. Further, since $C_1 \vdash C_2$, we have $C_1 \cup C' \vdash C_2 \cup C'$. Putting these together and applying [SUB-$\exists$] yields $C; D \vdash \exists^{l_1}\vec{\alpha}[C_1 \cup C'].(\tau_1\langle C'\rangle) \leq \exists^{l_2}\vec{\alpha}[C_2 \cup C'].(\tau_2\langle C'\rangle)$. $\square$

Lemma B.1.6 (Constraint weakening in polymorphic types) *If $C; \Gamma \vdash e : \tau$, then $C \cup C'; \Gamma\langle C'\rangle \vdash e : \tau\langle C'\rangle$.*

Proof: First, observe that by Lemma B.1.3, we may assume $C \cup C'; \Gamma \vdash e : \tau$. Then the proof proceeds by induction on the derivation of $C \cup C'; \Gamma \vdash e : \tau$.

Case [ID]. We have

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

Then trivially we have

$$[\text{ID}]\frac{}{C \cup C'; \Gamma\langle C'\rangle, x : \tau\langle C'\rangle \vdash_{cp} x : \tau\langle C'\rangle}$$

Case [INT]. Trivial.

Case [LAM]. We have

$$[\text{LAM}]\frac{C \cup C'; \Gamma, x : \tau_1 \vdash_{cp} e : \tau_2 \qquad C \cup C' \vdash L \leq l}{C \cup C'; \Gamma \vdash_{cp} \lambda x.eL : \tau_1 \rightarrow^l \tau_2}$$

By induction we have $C \cup C'; \Gamma\langle C'\rangle, x : \tau_1\langle C'\rangle \vdash_{cp} e : \tau_2\langle C'\rangle$. Then by [LAM], we have $C \cup C'; \Gamma\langle C'\rangle \vdash_{cp} \lambda x.eL : (\tau_1 \rightarrow^l \tau_2)\langle C'\rangle$.

Case [APP], [PAIR], [PROJ], and [COND]. Similar to [LAM].

Case [SUB]. We have

$$[\text{SUB}]\frac{\begin{array}{c} C \cup C'; \Gamma \vdash_{cp} e : \tau_1 \\ C \cup C'; \emptyset \vdash \tau_1 \leq \tau_2 \end{array}}{C \cup C'; \Gamma \vdash_{cp} e : \tau_2}$$

By induction, $C \cup C'; \Gamma\langle C'\rangle \vdash_{cp} e : \tau_1\langle C'\rangle$. Further, by Lemma B.1.5, we have $C \cup C'; \emptyset \vdash \tau_1\langle C'\rangle \leq \tau_2\langle C'\rangle$. Thus applying [SUB], we have $C \cup C'; \Gamma \vdash_{cp} e : \tau_2\langle C'\rangle$.

Case [LET]. We have

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

By induction, we have $C_f \cup C'; \Gamma\langle C'\rangle \vdash_{cp} e_1 : \tau_1\langle C'\rangle$. Also by induction, we have $C \cup C'; \Gamma\langle C'\rangle, f : (\forall\vec{\alpha}[C_f].\tau_1)\langle C'\rangle \vdash_{cp} e_2 : \tau_2\langle C'\rangle$ since $C \cup C' \cup C' = C \cup C'$. Since $fl(C') \cap \vec{\alpha} = \emptyset$, we have $(\forall\vec{\alpha}[C_f].\tau_1)\langle C'\rangle = \forall\vec{\alpha}[C_f \cup C'].(\tau_1\langle C'\rangle)$. Further, $fl(\Gamma\langle C'\rangle) = fl(\Gamma) \cup fl(C')$ and $fl(\tau_1\langle C'\rangle) = fl(\tau_1) \cup fl(C')$, hence we have $\vec{\alpha} \subseteq (fl(\tau_1\langle C'\rangle) \cup fl(C_f \cup C')) \setminus fl(\Gamma)\langle C'\rangle$. Thus we can apply [LET] to yield $C \cup C'; \Gamma\langle C'\rangle \vdash_{cp}$ $\mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 : \tau_2\langle C'\rangle$.

Case [FIX]. Similar to [LET].

Case [INST]. We have

$$[\text{INST}]\frac{C \cup C' \vdash C_f[\vec{\alpha} \mapsto \vec{l}]}{C \cup C'; \Gamma, f : \forall\vec{\alpha}[C_f].\tau \vdash_{cp} f^i : \tau[\vec{\alpha} \mapsto \vec{l}]}$$

By our alpha-renaming convention, $fl(C') \cap \vec{\alpha} = \emptyset$. Then $(\Gamma, f : \forall\vec{\alpha}[C_f].\tau)\langle C'\rangle = \Gamma\langle C'\rangle, f : \forall\vec{\alpha}[C_f \cup C'].(\tau\langle C'\rangle)$. Clearly $C \cup C' \vdash C_f[\vec{\alpha} \mapsto \vec{l}] \cup C'$, and by our alpha-renaming convention $C_f[\vec{\alpha} \mapsto \vec{l}] \cup C' = (C_f \cup C')[\vec{\alpha} \mapsto \vec{l}]$. Therefore applying [INST]

yields $C \cup C'; (\Gamma, f : \forall\vec{\alpha}[C_f].\tau)\langle C'\rangle \vdash_{cp} f^i : (\tau\langle C'\rangle)[\vec{\alpha} \mapsto \vec{l}]$. Then since by our alpha-renaming convention $(\tau\langle C'\rangle)[\vec{\alpha} \mapsto \vec{l}] = (\tau[\vec{\alpha} \mapsto \vec{l}])\langle C'\rangle$, we have shown the conclusion.

Case [PACK]. We have

$$[\text{PACK}]\frac{C \cup C'; \Gamma \vdash_{cp} e : \tau[\vec{\alpha} \mapsto \vec{l}] \qquad C \cup C' \vdash C_1[\vec{\alpha} \mapsto \vec{l}] \qquad C \cup C' \vdash L \leq l}{C \cup C'; \Gamma \vdash_{cp} \mathsf{pack}^i\ e\ L : \exists^l\vec{\alpha}[C_1].\tau}$$

By induction, $C \cup C'; \Gamma\langle C'\rangle \vdash_{cp} (\tau[\vec{\alpha} \mapsto \vec{l}])\langle C'\rangle$ since $C \cup C' \cup C' = C \cup C'$. By our alpha-renaming convention, $fl(C') \cap \vec{\alpha} = \emptyset$, so $(\tau[\vec{\alpha} \mapsto \vec{l}])\langle C'\rangle = (\tau\langle C'\rangle)[\vec{\alpha} \mapsto \vec{l}]$. Clearly $C \cup C' \vdash C_1[\vec{\alpha} \mapsto \vec{l}] \cup C'$, and also by our alpha-renaming convention $C_1[\vec{\alpha} \mapsto \vec{l}] \cup C' = (C_1 \cup C')[\vec{\alpha} \mapsto \vec{l}]$. Thus applying [PACK] we have $C \cup C'; \Gamma\langle C'\rangle \vdash \mathsf{pack}^i\ e\ L : \exists^l\vec{\alpha}[C_1 \cup C'].(\tau\langle C'\rangle)$. And by our alpha-renaming convention, $\exists^l\vec{\alpha}[C_1 \cup C'].(\tau\langle C'\rangle) = (\exists^l\vec{\alpha}[C_1].\tau)\langle C'\rangle$, so we have shown the conclusion.

Case [UNPACK]. We have

$$[\text{UNPACK}]\frac{\begin{array}{c} C \cup C'; \Gamma \vdash_{cp} e_1 : \exists^l\vec{\alpha}[C_1].\tau_1 \qquad C \cup C' \vdash l \leq L \\ C \cup C' \cup C_1; \Gamma, x : \tau_1 \vdash_{cp} e_2 : \tau \\ \vec{\alpha} \subseteq (fl(\tau_1) \cup fl(C_1)) \setminus (fl(\Gamma) \cup fl(\tau) \cup fl(C) \cup fl(C')) \end{array}}{C \cup C'; \Gamma \vdash_{cp} \mathsf{unpack}\ x = e_1\ \mathsf{in}\ e_2 L : \tau}$$

By induction, $C \cup C'; \Gamma\langle C'\rangle \vdash_{cp} e_1 : (\exists^l\vec{\alpha}[C_1].\tau_1)\langle C'\rangle$. By our alpha-renaming convention, $(\exists^l\vec{\alpha}[C_1].\tau_1)\langle C'\rangle = \exists^l\vec{\alpha}[C_1 \cup C'].(\tau_1\langle C'\rangle)$. Also by induction, $C \cup C' \cup C_1; \Gamma\langle C'\rangle, x : \tau_1\langle C'\rangle \vdash_{cp} e_2 : \tau\langle C'\rangle$. Finally,

$$\vec{\alpha} \subseteq (fl(\tau_1\langle C'\rangle) \cup fl(C_1 \cup C')) \setminus (fl(\Gamma\langle C'\rangle) \cup fl(\tau\langle C'\rangle) \cup fl(C) \cup fl(C'))$$

since $fl(\Gamma\langle C'\rangle) = fl(\Gamma) \cup fl(C')$, $fl(\tau_1\langle C'\rangle) = fl(\tau_1) \cup fl(C')$, $fl(\tau\langle C'\rangle) = fl(\tau) \cup fl(C')$, and we assume by alpha-renaming that $fl(C') \cap \vec{\alpha} = \emptyset$. Thus applying [UNPACK] yields $C \cup C'; \Gamma \vdash_{cp} \mathsf{unpack}\ x = e_1\ \mathsf{in}\ e_2 L : \tau\langle C'\rangle$. □

Next we prove the substitution lemma for monomorphic types. Because in rule [LET], we fixed the set of constraints in the quantified type to be exactly the constraints for e_1 and e_2, respectively, we need a slightly nonstandard lemma: When we replace a variable with an expression, we might need to add the constraints for that expression to quantified types in the environment and in the result type. Hence the definition of $\langle C'\rangle$ above. While we could have used a [LET] rule that is simpler to reason about for soundness, or changed the [UNPACK] rule to match [LET], this particular formulation turns out to be very helpful in proving correspondence between the $\lambda^{cfl}_{\exists}$ and $\lambda^{cp}_{\exists}$ in Appendix B.2.

Lemma B.1.7 (Substitution lemma) *If $C; \Gamma, x : \tau' \vdash_{cp} e : \tau$, $C \vdash C'$, and $C'; \Gamma \vdash_{cp} e' : \tau'$, then $C; \Gamma\langle C'\rangle \vdash_{cp} e[x \mapsto e'] : \tau\langle C'\rangle$.*

Proof: The proof proceeds by induction on the derivation of $C; \Gamma, x : \tau' \vdash_{cp} e : \tau$.

Case [ID]. There are two cases. First, if $e = x$, we have

$$\frac{}{C; \Gamma, x : \tau' \vdash_{cp} x : \tau'}$$

Then $\tau = \tau'$, and since $x[x \mapsto e'] = e'$, by our assumption $C'; \Gamma \vdash_{cp} e' : \tau'$ we have $C; \Gamma\langle C' \rangle \vdash_{cp} e' : \tau'\langle C' \rangle$ by Lemmas B.1.3 and B.1.6.

Otherwise, we have

$$\frac{}{C; \Gamma, x : \tau \vdash_{cp} y : \tau}$$

where $y \neq x$. Hence $y \in \mathit{dom}(\Gamma)$, and since $y[x \mapsto e'] = y$, by Lemma B.1.6 we have $C; \Gamma\langle C' \rangle, x : \tau\langle C' \rangle \vdash_{cp} y : \tau\langle C' \rangle$.

Case [INT]. Trivial.

Case [LAM]. We have

$$[\textsc{Lam}]\frac{C; \Gamma, x : \tau', y : \tau_1 \vdash_{cp} e_2 : \tau_2 \qquad C \vdash L \leq l}{C; \Gamma, x : \tau' \vdash_{cp} \lambda y.e_2 L : \tau_1 \to^l \tau_2}$$

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

Case [APP]. We have

$$[\textsc{App}]\frac{\begin{array}{c} C; \Gamma, x : \tau' \vdash_{cp} e_1 : \tau_2 \to^l \tau \\ C; \Gamma, x : \tau' \vdash_{cp} e_2 : \tau_2 \qquad C \vdash l \leq L \end{array}}{C; \Gamma, x : \tau' \vdash_{cp} e_1\ e_2 L : \tau}$$

Then by induction, we have $C; \Gamma\langle C' \rangle \vdash_{cp} e_1[x \mapsto e'] : (\tau_2 \to^l \tau)\langle C' \rangle$ and $C; \Gamma\langle C' \rangle \vdash_{cp} e_2[x \mapsto e'] : \tau_2\langle C' \rangle$. Therefore we can apply [APP] to yield $C; \Gamma\langle C' \rangle \vdash_{cp} (e_1\ e_2 L)[x \mapsto e'] : \tau\langle C' \rangle$.

Case [PAIR], [PROJ], [COND]. Similar to [APP].

Case [SUB]. We have

$$[\textsc{Sub}]\frac{\begin{array}{c} C; \Gamma, x : \tau' \vdash_{cp} e : \tau_1 \\ C; \emptyset \vdash \tau_1 \leq \tau_2 \end{array}}{C; \Gamma, x : \tau' \vdash_{cp} e : \tau_2}$$

By induction, we have $C; \Gamma\langle C' \rangle \vdash_{cp} e[x \mapsto e'] : \tau_1\langle C' \rangle$. By Lemma B.1.2, we have $C; \emptyset \vdash \tau_1\langle C' \rangle \leq \tau_2\langle C' \rangle$. Thus we can apply [SUB] to yield $C; \Gamma\langle C' \rangle \vdash_{cp} e[x \mapsto e'] : \tau_2\langle C' \rangle$.

Case [LET]. We have

$$[\textsc{Let}]\frac{\begin{array}{c} C''; \Gamma, x : \tau' \vdash_{cp} e_1 : \tau_1 \\ C; \Gamma, x : \tau', f : \forall \vec{\alpha}[C''].\tau_1 \vdash_{cp} e_2 : \tau_2 \\ \vec{\alpha} \subseteq (fl(\tau_1) \cup fl(C'')) \setminus (fl(\Gamma) \cup fl(\tau')) \end{array}}{C; \Gamma, x : \tau' \vdash_{cp} \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

By Lemma B.1.3 and induction, we have $C' \cup C''; \Gamma\langle C'\rangle \vdash_{cp} e_1[x \mapsto e'] : \tau_1\langle C'\rangle$. Then since $x \neq f$ (they are in different syntactic categories), by induction we also have $C; \Gamma\langle C'\rangle, f : (\forall\vec{\alpha}[C''].\tau_1)\langle C'\rangle \vdash e_2[x \mapsto e'] : \tau_2\langle C'\rangle$. By our alpha-renaming convention, $fl(C') \cap \vec{\alpha} = \emptyset$, so $(\forall\vec{\alpha}[C''].\tau_1)\langle C'\rangle = \forall\vec{\alpha}[C' \cup C''].\tau_1\langle C'\rangle$. Finally,

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

where the last step holds since we assume $fl(C') \cap \vec{\alpha} = \emptyset$. Hence we can apply [LET] to yield $C; \Gamma\langle C'\rangle \vdash_{cp} (\mathsf{let}\ f = e_1\ \mathsf{in}\ e_2)[x \mapsto e'] : \tau_2\langle C'\rangle$.

Case [FIX]. Similar to [LET] and [INST].

Case [INST]. By Lemma B.1.6 we have $C; \Gamma\langle C'\rangle, x : \tau'\langle C'\rangle, f : (\forall\vec{\alpha}[C'' \cup C'].(\tau))\langle C'\rangle \vdash_{cp} f^i : \tau\langle C'\rangle$. Then since $f_i[x \mapsto e'] = f_i$ (note we assume different syntactic forms for functions and local variables), we trivially have $C; \Gamma\langle C'\rangle, f : (\forall\vec{\alpha}[C'' \cup C'].(\tau))\langle C'\rangle \vdash_{cp} f^i[x \mapsto e'] : \tau\langle C'\rangle$

Case [PACK]. We have

$$\text{[PACK]}\frac{C; \Gamma, x : \tau' \vdash_{cp} e : \tau[\vec{\alpha} \mapsto \vec{l}] \qquad C \vdash C''[\vec{\alpha} \mapsto \vec{l}] \qquad C \vdash L \leq l}{C; \Gamma, x : \tau' \vdash_{cp} \mathsf{pack}^i\ e\ L : \exists^l\vec{\alpha}[C''].\tau}$$

By induction, $C; \Gamma\langle C'\rangle \vdash_{cp} e[x \mapsto e'] : (\tau[\vec{\alpha} \mapsto \vec{l}])\langle C'\rangle$. Since we assume by alpha renaming that $fl(C') \cap \vec{\alpha} = \emptyset$, we have $(\tau[\vec{\alpha} \mapsto \vec{l}])\langle C'\rangle = (\tau\langle C'\rangle)[\vec{\alpha} \mapsto \vec{l}]$. Further, since $C \vdash C'$, we have $C \vdash C' \cup C''[\vec{\alpha} \mapsto \vec{l}]$, and again since $fl(C') \cap \vec{\alpha} = \emptyset$ we have $C \vdash (C' \cup C'')[\vec{\alpha} \mapsto \vec{l}]$. Then applying [PACK] yields $C; \Gamma\langle C'\rangle \vdash_{cp} (\mathsf{pack}^i\ e\ L)[x \mapsto e'] : (\exists^l\vec{\alpha}[C''].\tau)\langle C'\rangle$.

Case [UNPACK]. We have

$$\text{[UNPACK]}\frac{\begin{array}{c} C; \Gamma, x : \tau' \vdash_{cp} e_1 : \exists^l\vec{\alpha}[C''].\tau_1 \qquad\qquad C \vdash l \leq L \\ C \cup C''; \Gamma, x : \tau', y : \tau_1 \vdash_{cp} e_2 : \tau \\ \vec{\alpha} \subseteq (fl(\tau_1) \cup fl(C'')) \setminus (fl(\Gamma) \cup fl(\tau') \cup fl(\tau) \cup fl(C)) \end{array}}{C; \Gamma, x : \tau' \vdash_{cp} \mathsf{unpack}\ y = e_1\ \mathsf{in}\ e_2 L : \tau}$$

Assume by alpha renaming that $x \neq y$. Then by induction, $C; \Gamma\langle C'\rangle \vdash_{cp} e_1[x \mapsto e'] : (\exists^l\vec{\alpha}[C''].\tau_1)\langle C'\rangle$. By our alpha renaming convention, we assume $fl(C') \cap \vec{\alpha} = \emptyset$, hence $(\exists^l\vec{\alpha}[C''].\tau_1)\langle C'\rangle = \exists^l\vec{\alpha}[C' \cup C''].(\tau_1\langle C'\rangle)$. Since $C \vdash C'$, we have $C \cup C' \cup C'' = C \cup C''$. Thus by Lemma B.1.3 and induction we have $C \cup C' \cup C''; \Gamma\langle C'\rangle, y : \tau_1\langle C'\rangle \vdash_{cp} e_2[x \mapsto e'] : \tau\langle C'\rangle$. Finally,

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

again because we assume $fl(C') \cap \vec{\alpha} = \emptyset$. Hence we can apply [UNPACK] to yield $C; \Gamma\langle C'\rangle \vdash_{cp} (\mathsf{unpack}\ y = e_1\ \mathsf{in}\ e_2 L)[x \mapsto e'] : \tau\langle C'\rangle$.

□

Lemma B.1.8 (Polymorphic substitution lemma) *If* $C; \Gamma, f : \forall\vec{\alpha}[C'].\tau' \vdash_{cp} e : \tau$ *and* $C'; \Gamma \vdash_{cp} e' : \tau'$ *where* $\vec{\alpha} \cap fl(\Gamma) = \emptyset$*, then* $C; \Gamma \vdash_{cp} e[f \mapsto e'] : \tau$.

Proof: By induction on the derivation of $C; \Gamma, f : \forall\vec{\alpha}[C'].\tau' \vdash_{cp} e : \tau$.

Case [ID]. Trivial, since $x[f \mapsto e'] = x$ (note we assume different syntactic forms for functions and local variables).

Case [INT]. Trivial.

Case [LAM]. We have

$$[\text{LAM}]\frac{C; \Gamma, f : \forall\vec{\alpha}[C'].\tau', x : \tau_1 \vdash_{cp} e : \tau_2 \qquad C \vdash L \leq l}{C; \Gamma, f : \forall\vec{\alpha}[C'].\tau' \vdash_{cp} \lambda x.eL : \tau_1 \rightarrow^l \tau_2}$$

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

Case [APP]. We have

$$[\text{APP}]\frac{\begin{array}{c} C; \Gamma, f : \forall\vec{\alpha}[C'].\tau' \vdash_{cp} e_1 : \tau_2 \rightarrow^l \tau_1 \\ C; \Gamma, f : \forall\vec{\alpha}[C'].\tau' \vdash_{cp} e_2 : \tau_2 \qquad C \vdash l \leq L \end{array}}{C; \Gamma, f : \forall\vec{\alpha}[C'].\tau' \vdash_{cp} e_1\ e_2 L : \tau_1}$$

By induction, we have $C; \Gamma \vdash_{cp} e_1[f \mapsto e'] : \tau_2 \rightarrow^l \tau_1$ and $C; \Gamma \vdash_{cp} e_2[f \mapsto e'] : \tau_2$. Then applying [APP] yields $C; \Gamma \vdash_{cp} (e_1\ e_2 L)[f \mapsto e'] : \tau_1$.

Case [PAIR], [PROJ], [COND], [SUB]. Analogous to [APP].

Case [LET]. We have

$$[\text{LET}]\frac{\begin{array}{c} C''; \Gamma, f : \forall\vec{\alpha}[C'].\tau' \vdash_{cp} e_1 : \tau_1 \\ C; \Gamma, f : \forall\vec{\alpha}[C'].\tau', g : \forall\vec{\beta}[C''].\tau_1 \vdash_{cp} e_2 : \tau_2 \\ \vec{\beta} \subseteq (fl(\tau_1) \cup fl(C'')) \setminus (fl(\Gamma) \cup fl(\forall\vec{\alpha}[C'].\tau')) \end{array}}{C; \Gamma, f : \forall\vec{\alpha}[C'].\tau' \vdash_{cp} \mathsf{let}\ g = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

By induction, $C''; \Gamma \vdash_{cp} e_1[f \mapsto e'] : \tau_1$. Assuming by alpha renaming that $f \neq g$, by induction we also have

$$C; \Gamma, g : \forall\vec{\beta}[C''].\tau_1 \vdash_{cp} e_2[f \mapsto e'] : \tau_2$$

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 $C; \Gamma \vdash_{cp} (\mathsf{let}\ g = e_1\ \mathsf{in}\ e_2)[f \mapsto e'] : \tau_2$.

Case [FIX]. Similar to [LET] and [INST].

Case [INST]. There are two cases. If $e \neq f$, then the conclusion holds trivially, since $e[f \mapsto e'] = e$. Otherwise, we have

$$[\text{INST}]\frac{C \vdash C'[\vec{\alpha} \mapsto \vec{l}]}{C; \Gamma, f : \forall\vec{\alpha}[C'].\tau \vdash_{cp} f^i : \tau[\vec{\alpha} \mapsto \vec{l}]}$$

By assumption, $C'; \Gamma \vdash_{cp} e' : \tau'$. Then $C'[\vec{\alpha} \mapsto \vec{l}]; \Gamma[\vec{\alpha} \mapsto \vec{l}] \vdash_{cp} e' : \tau[\vec{\alpha} \mapsto \vec{l}]$. But since by assumption $\vec{\alpha} \cap fl(\Gamma) = \emptyset$, we then have $C'[\vec{\alpha} \mapsto \vec{l}]; \Gamma \vdash_{cp} e' : \tau[\vec{\alpha} \mapsto \vec{l}]$. But $C \vdash C'[\vec{\alpha} \mapsto \vec{l}]$, and so by Lemma B.1.3, $C; \Gamma \vdash_{cp} e' : \tau[\vec{\alpha} \mapsto \vec{l}]$, and so we have shown the conclusion, since $f^i[f \mapsto e'] = e'$.

Case [PACK]. We have

$$[\text{PACK}]\frac{C; \Gamma, f : \forall\vec{\alpha}[C'].\tau' \vdash_{cp} e : \tau[\vec{\beta} \mapsto \vec{l}] \quad C \vdash C''[\vec{\beta} \mapsto \vec{l}] \quad C \vdash L \leq l}{C; \Gamma, f : \forall\vec{\alpha}[C'].\tau' \vdash_{cp} \mathsf{pack}^i\ e\ L : \exists^l\vec{\beta}[C''].\tau}$$

By induction, we have $C; \Gamma \vdash_{cp} e[f \mapsto e'] : \tau[\vec{\beta} \mapsto \vec{l}]$. But then we can apply [PACK] to show $C; \Gamma \vdash_{cp} (\mathsf{pack}^i\ e\ L)[f \mapsto e'] : \exists^l\vec{\beta}[C''].\tau$.

Case [UNPACK]. We have

$$[\text{UNPACK}]\frac{\begin{array}{c} C; \Gamma, f : \forall\vec{\alpha}[C'].\tau' \vdash_{cp} e_1 : \exists^l\vec{\beta}[C''].\tau_1 \qquad C \vdash l \leq L \\ C \cup C''; \Gamma, f : \forall\vec{\alpha}[C'].\tau', x : \tau_1 \vdash_{cp} e_2 : \tau \\ \vec{\beta} \subseteq (fl(\tau_1) \cup fl(C'')) \setminus (fl(\Gamma) \cup fl(\forall\vec{\alpha}[C'].\tau') \cup fl(\tau) \cup fl(C)) \end{array}}{C; \Gamma, f : \forall\vec{\alpha}[C'].\tau' \vdash_{cp} \mathsf{unpack}\ x = e_1\ \mathsf{in}\ e_2 L : \tau}$$

By induction, we have $C; \Gamma \vdash_{cp} e_1[f \mapsto e'] : \exists^l\vec{\beta}[C''].\tau_1$. Also by induction, assuming that $f \neq x$ (since functions are in a different syntactic category), we have $C \cup C''; \Gamma, x : \tau_1 \vdash_{cp} e_2[f \mapsto e'] : \tau$. Finally,

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

Thus we can apply [UNPACK] to show $C; \Gamma \vdash (\mathsf{unpack}\ x = e_1\ \mathsf{in}\ e_2 L)[f \mapsto e'] : \tau$. □

Finally, we can state and prove our soundness theorem. We assume that the program is well-typed with respect to the standard types. Hence, every program is either in normal form or can take a step. We wish to prove that, for any destructor that consumes a value, the actual constructor label that is consumed appears in the set of labels computed by the analysis. If the program is in normal form this is trivial, because there are no more evaluation steps. Hence we prove this statement below for the case when the program takes a single step.

Definition B.1.9 *Suppose $e \longrightarrow e'$ and in the (single step) reduction, the destructor* (`if0`, @, $.j$, `unpack`) *labeled L' consumes the constructor* (n, λ, $(\cdot,\cdot)$, `pack`) *labeled L. Then we write $C \vdash e \longrightarrow e'$ if $C \vdash L \leq L'$. We also write $C \vdash e \longrightarrow e'$ if no value is consumed during reduction (e.g., for* `let` *or* `fix`*).*

Notice that if $C \vdash e \longrightarrow e'$ and $\mathbb{E}[e] \longrightarrow \mathbb{E}[e']$, then $C \vdash \mathbb{E}[e] \longrightarrow \mathbb{E}[e']$, since reducing inside of a context does not change which destructor consumed which constructor. We will use this fact implicitly in the proof below.

Lemma B.1.10 (Preservation) *If $C;\Gamma \vdash_{cp} e : \tau$ and $e \longrightarrow e'$, then $C;\Gamma\langle C\rangle \vdash_{cp} e' : \tau\langle C\rangle$ and $C \vdash e \longrightarrow e'$.*

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

Case [ID], [INT]. These cases cannot happen, because we assume $e \longrightarrow e'$.

Case [LAM]. In this case, the tern is $\lambda x.eL$, and the only possible reduction is $\lambda x.eL \longrightarrow \lambda x.e'L$. By assumption, we have

$$[\textsc{Lam}]\frac{C;\Gamma, x:\tau \vdash_{cp} e : \tau' \qquad C \vdash L \leq l}{C;\Gamma \vdash_{cp} \lambda x.eL : \tau \to^{l} \tau'}$$

By induction, $C;\Gamma\langle C\rangle, x : \tau\langle C\rangle \vdash_{cp} e' : \tau'\langle C\rangle$ and $C \vdash e \longrightarrow e'$. Then applying [LAM] yields $C;\Gamma\langle C\rangle \vdash_{cp} \lambda x.e'L : (\tau \to^{l} \tau')\langle C\rangle$, and we also have $C \vdash \lambda x.eL \longrightarrow \lambda x.e'L$.

Case [APP]. In this case, the term is $e_1\ e_2L$, and there are three possible reductions. In the first case, when $e_1\ e_2L \longrightarrow e_1'\ e_2L$, we have

$$[\textsc{App}]\frac{\begin{array}{c}C;\Gamma \vdash_{cp} e_1 : \tau_2 \to^{l} \tau_1 \\ C;\Gamma \vdash_{cp} e_2 : \tau_2 \qquad C \vdash l \leq L\end{array}}{C;\Gamma \vdash_{cp} e_1\ e_2L : \tau_1}$$

Then by induction, $C;\Gamma\langle C\rangle \vdash_{cp} e_1' : (\tau_2 \to^{l} \tau_1)\langle C\rangle$. By Lemma B.1.6, $C;\Gamma\langle C\rangle \vdash_{cp} e_2 : \tau_2\langle C\rangle$. Thus we can apply [APP] to yield $C;\Gamma\langle C\rangle \vdash_{cp} e_1'\ e_2L : \tau_1\langle C\rangle$. Also by induction, $C \vdash e_1 \longrightarrow e_1'$, so $C \vdash e_1\ e_2L \longrightarrow e_1'\ e_2L$. The second case, when $e_1\ e_2L \longrightarrow e_1\ e_2'L$, is similar.

In the last case, we have $(\lambda x.e_1L)\ e_2L' \longrightarrow e_1[x \mapsto e_2]$. In this case, we have

$$\dfrac{\dfrac{[\textsc{Lam}]\ \dfrac{C;\Gamma, x:\tau_1 \vdash e_1 : \tau_2 \qquad C \vdash L \leq l'}{C;\Gamma \vdash_{cp} \lambda x.e_1L : \tau_1 \to^{l'} \tau_2} \qquad C;\emptyset \vdash (\tau_1 \to^{l'} \tau_2) \leq (\tau' \to^{l} \tau)}{C;\Gamma \vdash_{cp} \lambda x.e_1L : \tau' \to^{l} \tau} \qquad C \vdash l \leq L'}{C;\Gamma \vdash_{cp} e_2 : \tau'}$$
$$\dfrac{}{C;\Gamma \vdash (\lambda x.e_1L)\ e_2L' : \tau}$$

Then $C;\emptyset \vdash \tau' \leq \tau_1$, hence $C;\Gamma \vdash_{cp} e_2 : \tau_1$. Then by Lemma B.1.7, $C;\Gamma\langle C\rangle \vdash_{cp} e_1[x \mapsto e_2] : \tau_2\langle C\rangle$. By Lemma B.1.5, $C;\emptyset \vdash \tau_2\langle C\rangle \leq \tau\langle C\rangle$. Thus by [SUB] we have $C;\Gamma\langle C\rangle \vdash_{cp} e_1[x \mapsto e_2] : \tau\langle C\rangle$.

Finally, in this reduction step L' consumes L. But $C \vdash L \leq l'$, $C \vdash l' \leq l$, and $C \vdash l \leq L'$, hence $C \vdash L \leq L'$. Hence we have shown the conclusion.

Case [PAIR]. In this case, we have either $(e_1, e_2)\, L \longrightarrow (e_1', e_2)\, L$ or $(e_1, e_2)\, L \longrightarrow (e_1, e_2')\, L$. In either case the proof proceeds by induction, similar to the first case of [APP].

Case [PROJ]. In this case, the term is $e.jL$. There are two possible reductions. If the reduction is $e.jL \longrightarrow e'.jL$, then we apply induction as in the first case of [APP]. Otherwise, the reduction is $(e_1, e_2)\, L'.jL \longrightarrow e_j$. In this case, our typing proof is of the form

$$\text{[PROJ]}\frac{\text{[SUB]}\dfrac{\text{[PAIR]}\dfrac{C;\Gamma \vdash e_1 : \tau_1' \quad C;\Gamma \vdash e_2 : \tau_2' \quad C \vdash L' \leq l'}{C;\Gamma \vdash_{cp} (e_1, e_2)\, L' : \tau_1' \times^{l'} \tau_2'} \quad C;\emptyset \vdash \tau_1' \times^{l'} \tau_2' \leq \tau_1 \times^{l} \tau_2}{C;\Gamma \vdash_{cp} (e_1, e_2)\, L' : \tau_1 \times^{l} \tau_2} \quad C \vdash l \leq L \quad j = 1,2}{C;\Gamma \vdash_{cp} (e_1, e_2)\, L'.jL : \tau_j}$$

Then $C;\Gamma \vdash e_j : \tau_j'$, and since $C;\emptyset \vdash \tau_j' \leq \tau_j$, we have $C;\Gamma \vdash e_j : \tau_j$. Then by Lemma B.1.6, we have $C;\Gamma\langle C\rangle \vdash e_j : \tau_j\langle C\rangle$. Also, $C \vdash L' \leq l'$, $C \vdash l' \leq l$, and $C \vdash l \leq L$, hence $C \vdash L' \leq L$, and we have shown the conclusion.

Case [COND]. In this case, the term is $\mathsf{if0}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 L$, and there are four possible reductions. If the reduction occurs inside of e_0, e_1, or e_2, then we proceed by induction as usual. Otherwise, the reduction is either $\mathsf{if0}\ n^{L'}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 L \longrightarrow e_1$ or $\mathsf{if0}\ n^{L'}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 L \longrightarrow e_2$, depending on whether n is 0. In either case, our typing judgment looks like

$$\text{[COND]}\frac{C \vdash l \leq L \quad \text{[SUB]}\dfrac{\text{[INT]}\dfrac{C \vdash L' \leq l'}{C;\Gamma \vdash_{cp} n^{L'} : intl'} \quad C;\emptyset \vdash intl' \leq intl}{C;\Gamma \vdash_{cp} n^{L'} : intl} \quad C;\Gamma \vdash_{cp} e_1 : \tau \quad C;\Gamma \vdash_{cp} e_2 : \tau}{C;\Gamma \vdash_{cp} \mathsf{if0}\ n^{L'}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 L : \tau}$$

Then clearly $C;\Gamma \vdash_{cp} e_i : \tau$, and by Lemma B.1.6 we have $C;\Gamma\langle C\rangle \vdash_{cp} e_i : \tau\langle C\rangle$. And since $C \vdash L' \leq l'$, $C \vdash l' \leq l$, and $C \vdash l \leq L$, we have $C \vdash L' \leq L$.

Case [SUB]. In this case, the reduction is $e \longrightarrow e'$, and we have

$$\text{[SUB]}\frac{C;\Gamma \vdash_{cp} e : \tau_1 \quad C;\emptyset \vdash \tau_1 \leq \tau_2}{C;\Gamma \vdash_{cp} e : \tau_2}$$

But by induction, $C;\Gamma\langle C\rangle \vdash_{cp} e' : \tau_1\langle C\rangle$. By Lemma B.1.5, $C;\emptyset \vdash \tau_1\langle C\rangle \leq \tau_2\langle C\rangle$. Hence we can apply [SUB] to show $C;\Gamma\langle C\rangle \vdash_{cp} e' : \tau_2\langle C\rangle$.

Case [LET]. In this case, the term is $\mathsf{let}\ f = e_1\ \mathsf{in}\ e_2$. If the reduction occurs inside of

e_1 or e_2, then we proceed by induction as usual. Otherwise, the typing judgment is of the form

$$\text{[LET]}\frac{C';\Gamma \vdash_{cp} e_1 : \tau_1 \qquad C;\Gamma, f : \forall\vec{\alpha}[C'].\tau_1 \vdash_{cp} e_2 : \tau_2 \qquad \vec{\alpha} \subseteq (fl(\tau_1) \cup fl(C')) \setminus fl(\Gamma)}{C;\Gamma \vdash_{cp} \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

and the reduction is $\mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 \longrightarrow e_2[f \mapsto e_1]$. But then by Lemma B.1.8, $C;\Gamma \vdash e_2[f \mapsto e_1] : \tau_2$. Then by Lemma B.1.6, $C;\Gamma\langle C\rangle \vdash e_2[f \mapsto e_1] : \tau_2\langle C\rangle$. Since no labeled values are consumed by this reduction, $C \vdash \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 \longrightarrow e_2[f \mapsto e_1]$ trivially, and we are done.

Case [FIX]. Analogous to [LET].

Case [INST]. This case cannot happen, because we assume $e \longrightarrow e'$.

Case [PACK]. This case proceeds by induction as usual. In this case the reduction must be $\mathsf{pack}^i\ e\ L \longrightarrow \mathsf{pack}^i\ e'\ L$, and so we proceed by the usual induction. The typing proof is

$$\text{[PACK]}\frac{C;\Gamma \vdash_{cp} e : \tau[\vec{\alpha} \mapsto \vec{l}] \qquad C \vdash C'[\vec{\alpha} \mapsto \vec{l}] \qquad C \vdash L \leq l}{C;\Gamma \vdash_{cp} \mathsf{pack}^i\ e\ L : \exists^l\vec{\alpha}[C'].\tau}$$

Then by induction, $C;\Gamma\langle C\rangle \vdash_{cp} e' : (\tau[\vec{\alpha} \mapsto \vec{l}])\langle C\rangle$ and $C \vdash e \longrightarrow e'$. By our alpha renaming convention, $(\tau[\vec{\alpha} \mapsto \vec{l}])\langle C\rangle = (\tau\langle C\rangle)[\vec{\alpha} \mapsto \vec{l}]$. Further, $C \vdash C \cup C'[\vec{\alpha} \mapsto \vec{l}]$, and again by our alpha renaming convention $C \vdash (C \cup C')[\vec{\alpha} \mapsto \vec{l}]$. Hence by [PACK] we have $C;\Gamma\langle C\rangle \vdash_{cp} \mathsf{pack}^i\ e\ L : (\exists^l\vec{\alpha}[C'].\tau)\langle C\rangle$, and we also have $C \vdash \mathsf{pack}^i\ e\ L \longrightarrow \mathsf{pack}^i\ e'\ L$.

Case [UNPACK]. In this case the term is $\mathsf{unpack}\ x = e_1\ \mathsf{in}\ e_2 L$, and there are three possible reductions. If the reduction occurs inside e_1 or e_2 then apply induction as usual. Otherwise, the reduction is $\mathsf{unpack}\ x = (\mathsf{pack}^i\ e\ L')\ \mathsf{in}\ e_2 L \longrightarrow e_2[x \mapsto e]$, and the typing proof is

$$\text{[UNPACK]}\frac{\text{[SUB]}\ \dfrac{\text{[PACK]}\ \dfrac{C;\Gamma \vdash_{cp} e : \tau_1[\vec{\alpha} \mapsto \vec{l}] \qquad C \vdash C_1[\vec{\alpha} \mapsto \vec{l}] \qquad C \vdash L' \leq l_1}{C;\Gamma \vdash_{cp} \mathsf{pack}^i\ e\ L' : \exists^{l_1}\vec{\alpha}[C_1].\tau_1} \qquad C;\emptyset \vdash_{cp} \exists^{l_1}\vec{\alpha}[C_1].\tau_1 \leq \exists^{l_2}\vec{\alpha}[C_2].\tau_2}{C;\Gamma \vdash_{cp} \mathsf{pack}^i\ e\ L' : \exists^{l_2}\vec{\alpha}[C_2].\tau_2} \qquad C \vdash l_2 \leq L \qquad C \cup C_2;\Gamma, x : \tau_2 \vdash_{cp} e_2 : \tau \qquad \vec{\alpha} \subseteq (fl(\tau_2) \cup fl(C_2)) \setminus (fl(\Gamma) \cup fl(\tau) \cup fl(C))}{C;\Gamma \vdash_{cp} \mathsf{unpack}\ x = (\mathsf{pack}^i\ e\ L')\ \mathsf{in}\ e_2 L : \tau}$$

Further, the subtyping derivation is

$$\text{[SUB-}\exists\text{]}\frac{C_1 \vdash C_2 \qquad D' = [l \mapsto 1, \forall l \in \vec{\alpha}] \qquad C;D' \vdash \tau_1 \leq \tau_2 \qquad C;D \vdash l_1 \leq l_2}{C;\emptyset \vdash \exists^{l_1}\vec{\alpha}[C_1].\tau_1 \leq \exists^{l_2}\vec{\alpha}[C_2].\tau_2}$$

We show soundness by applying the Substitution Lemma. First, we have $C \cup C_2; \Gamma, x : \tau_2 \vdash_{cp} e_2 : \tau$. Let ψ be the substitution $[\vec{\alpha} \mapsto \vec{l}]$. Applying this to our judgment yields $\psi(C) \cup \psi(C_2); \psi(\Gamma), x : \psi(\tau_2) \vdash_{cp} e_2 : \psi(\tau)$. But since $\vec{\alpha} \cap (fl(\Gamma) \cup fl(\tau) \cup fl(C)) = \emptyset$, we have $C \cup \psi(C_2); \Gamma, x : \psi(\tau_2) \vdash_{cp} e_2 : \tau$.

Further, since $C_1 \vdash C_2$, we have $\psi(C_1) \vdash \psi(C_2)$. Then since $C \vdash \psi(C_1)$, we have $C \vdash \psi(C_2)$. Then by Lemma B.1.3, we have the following conclusion:

$$C; \Gamma, x : \psi(\tau_2) \vdash_{cp} e_2 : \tau$$

By assumption, we have $C; \Gamma \vdash_{cp} e : \psi(\tau_1)$. Also by assumption, we have $C; D' \vdash \tau_1 \leq \tau_2$. Then by Lemma B.1.4, we have $C; \emptyset \vdash \psi(\tau_1) \leq \psi(\tau_2)$. Therefore by [SUB], we have the following conclusion:

$$C; \Gamma \vdash_{cp} e : \psi(\tau_2)$$

Now we can apply Lemma B.1.7 to yield $C; \Gamma\langle C \rangle \vdash_{cp} e_2[x \mapsto e] : \tau\langle C \rangle$.

Finally, observe $C \vdash L' \leq l_1$, $C \vdash l_1 \leq l_2$, and $C \vdash l_2 \leq L$. Hence $C \vdash L' \leq L$, and therefore $C \vdash \mathsf{unpack}\ x = (\mathsf{pack}^i\ e\ L')\ \mathsf{in}\ e_2 L \longrightarrow e_2[x \mapsto e]$, so we are done. □

Theorem B.1.11 (Soundness) *If $C; \Gamma \vdash_{cp} e : \tau$ and $e \longrightarrow^* e'$, then $C \vdash e \longrightarrow^* e'$.*

Proof: By induction on the length of the reduction $e \longrightarrow^* e'$, using Lemma B.1.10. □

B.2 Reduction from $\lambda_{\exists}^{cfl}$ to $\lambda_{\exists}^{cp}$

In this section, we prove that typing proofs in $\lambda_{\exists}^{cfl}$ reduce to equivalent proofs in $\lambda_{\exists}^{cp}$. As mentioned earlier, $\lambda_{\exists}^{cfl}$ is actually more restrictive than $\lambda_{\exists}^{cp}$ in the programs it is able to check.

We proceed following the basic proof technique in [38], but due to higher-order polymorphic types, our proof is somewhat more complicated.

Definition B.2.1 (Polarity of label in type) *Let τ be a $\lambda_{\exists}^{cfl}$ type. We say that some label $l \in fl(\tau)$ has positive polarity (+) in τ if one of the following holds:*

1. $\tau = intl$
2. $\tau = \tau_1 \rightarrow^{l'} \tau_2$ *and* $l = l'$ *or* l *has* $+$ *polarity in* τ_2 *or* l *has* $-$ *polarity in* τ_1.
3. $\tau = \tau_1 \times^{l'} \tau_2$ *and* $l = l'$ *or* l *has* $+$ *polarity in* τ_1 *or in* τ_2.
4. $\tau = \exists^{l'} \vec{\alpha}.\tau'$ *and either* $l = l'$, *or* $l \notin \vec{\alpha}$ *and* l *has* $+$ *polarity in* τ'

Similarly, we say that some label $l \in fl(\tau)$ has negative polarity $-$ in τ if one of the following holds:

1. $\tau = \tau_1 \rightarrow^{l'} \tau_2$ *and* l *has* $-$ *polarity in* τ_2 *or* l *has* $+$ *polarity in* τ_1.

2. $\tau = \tau_1 \times^{l'} \tau_2$ *and* l *has* $-$ *polarity in* τ_1 *or in* τ_2.

3. $\tau = \tau = \exists^{l'} \vec{\alpha}.\tau'$ *and* $l \notin \vec{\alpha}$ *and* l *has* $-$ *polarity in* τ'.

Definition B.2.2 (Polarized constraint sets) *Let* C *be a set of flow constraints, let* τ *be a* $\lambda^{cfl}_{\exists}$ *type and let* $F \subseteq fl(\tau)$. *We say that* C is p-polarized with respect to τ and F, *written* $C \rhd^p_F \tau$, *if-f the following conditions hold for all* $l \in F$:

1. *whenever* $C \vdash l \leq l'$ *with* $l \neq l'$, *then* l *has polarity* $\bar{p}$ *in* τ.

2. *whenever* $C \vdash l' \leq l$ *with* $l \neq l'$, *then* l *has polarity* p *in* τ.

Lemma B.2.3 *If* l *has polarity* p *in* τ, $I; D \vdash \tau \preceq^i_{p'} \tau' : \phi$, *and* $l \notin D$, *then* $I \vdash l \preceq^i_{p \cdot p'} \phi(l)$, *where* $p \cdot p = +$ *and* $p \cdot \bar{p} = -$.

Proof: Induction over the instantiation $I; D \vdash \tau \preceq^i_{p'} \tau' : \phi$:

Case [INST-INT]. We have

$$[\text{INST-INT}]\frac{I; D \vdash l \preceq^i_{p'} l' : \phi}{I; D \vdash intl \preceq^i_{p'} intl' : \phi}$$

Since $l \notin D$, we conclude that $I; D \vdash l \preceq^i_p l' : \phi$ can only have been proved by [INST-INDEX-1]. Thus

$$[\text{INDEX-INDEX-1}]\frac{I \vdash l \preceq^i_{p'} l' \qquad \{(l, l')\} \in \phi}{I; \emptyset; \emptyset \vdash l \preceq^i_{p'} l' : \phi}$$

From this we get $l' = \phi(l)$ and $I \vdash l \preceq^i_{p'} \phi(l)$. But l has $+$ polarity in $intl$. , and by definition, $p' \cdot + = p'$, and thus we have $I \vdash l \preceq^i_{p' \cdot +} \phi(l)$.

Case [INST-PAIR]. We have

$$[\text{INST-PAIR}]\frac{\begin{array}{c} I; D \vdash l_1 \preceq^i_{p'} l_2 : \phi \\ I; D \vdash \tau_1 \preceq^i_{p'} \tau_1' : \phi \qquad I; D \vdash \tau_2 \preceq^i_{p'} \tau_2' : \phi \end{array}}{I; D \vdash \tau_1 \times^{l_1} \tau_2 \preceq^i_{p'} \tau_1' \times^{l_2} \tau_2' : \phi}$$

There are two cases:

- $l = l_1$. Then as in the previous case, since $l \notin D$ we get $l_2 = \phi(l_1)$ and $I \vdash l \preceq^i_{p'} \phi(l)$ from [INST-INDEX-1]. Then, since l has polarity $+$ in $\tau_1 \times^l \tau_2$, we have $I \vdash l \preceq^i_{p' \cdot +} \phi(l)$.

- $l \in fl(\tau_i)$ $(i = 1, 2)$. Then by Definition B.2.1, l has polarity p in τ_i. Then by induction we have $I \vdash l \preceq^i_{p \cdot p'} \phi(l)$

Case [INST-FUN]. We have

$$\text{[INST-FUN]}\frac{\begin{array}{c} I; D \vdash l_1 \preceq^i_{p'} l_2 : \phi \\ I; D \vdash \tau_1 \preceq^i_{\bar{p}'} \tau_1' : \phi \qquad I; D \vdash \tau_2 \preceq^i_{p'} \tau_2' : \phi \end{array}}{I; D \vdash \tau_1 \to^{l_1} \tau_2 \preceq^i_{p'} \tau_1' \to^{l_2} \tau_2' : \phi}$$

There are three cases:

- $l = l_1$. Then since $l \notin D$, as before by [INST-INDEX-1] we have $l_2 = \phi(l_1)$ and $I \vdash l \preceq^i_{p'} \phi(l)$. Since l has polarity $+$ in $\tau_1 \to^l \tau_2$, we then have $I \vdash l \preceq^i_{p'\cdot +} \phi(l)$.
- $l \in \mathit{fl}(\tau_1)$. By Definition B.2.1, l has polarity $\bar{p}$ in τ_1. Then by induction we have $I \vdash l \preceq^i_{\bar{p}\cdot\bar{p'}} \phi(l)$ But since $\bar{p} \cdot \bar{p'} = p \cdot p'$, this is equivalent to $I \vdash l \preceq^i_{p\cdot p'} \phi(l)$.
- $l \in \mathit{fl}(\tau_2)$. By Definition B.2.1, l has polarity p in τ_2. As before, by induction we then have $I \vdash l \preceq^i_{p\cdot p'} \phi(l)$.

Case [INST-$\exists$]. We have

$$\text{[INST-}\exists\text{]}\frac{\begin{array}{c} D' = D \oplus \vec{\alpha} \qquad I; D' \vdash \tau_1 \preceq^i_{p'} \tau_2 : \phi \\ I; D \vdash l_1 \preceq^i_{p'} l_2 : \phi \end{array}}{I; D \vdash \exists^{l_1}\vec{\alpha}.\tau_1 \preceq^i_{p'} \exists^{l_2}\vec{\alpha}.\tau_2 : \phi}$$

There are two cases:

- Case $l = l_1$. Then since $l \notin D$, by [INST-INDEX-1] we have $l_2 = \phi(l_1)$ and $I \vdash l \preceq^i_{p'} \phi(l)$. Since l has polarity $+$ in $\exists^{l_1}\vec{\alpha}.\tau_1$, we then have $I \vdash l \preceq^i_{p'\cdot +} \phi(l)$.
- Case $l \in \mathit{fl}(\tau_1)$. We may assume $l \notin \vec{\alpha}$, since otherwise $l \notin \mathit{fl}(\exists^{l_1}\vec{\alpha}.\tau_1)$. Therefore $l \notin D'$. Further, by Definition B.2.1 the polarity of l in τ_1 is p. Then by induction we have $I \vdash l \preceq^i_{p\cdot p'} \phi(l)$.

□

Definition B.2.4 (Instantiation context) *Every application of an [*INST*]*

$$\textit{[INST]}\frac{\begin{array}{c} I; \emptyset \vdash \tau \preceq^i_+ \tau' : \phi \\ \mathit{dom}(\phi) = \vec{\alpha} \qquad I \vdash \vec{l} \preceq^i_+ \vec{l} \qquad I \vdash \vec{l} \preceq^i_- \vec{l} \end{array}}{I; C; \Gamma, f : (\forall\vec{\alpha}.\tau, \vec{l}) \vdash_{CFL} f^i : \tau'}$$

defines a positive instantiation context $\langle C, I, \vec{\alpha}, \vec{l}, \tau, \phi, +, i\rangle$.

*Every application of [*PACK*]*

$$\textit{[PACK]}\frac{\begin{array}{cc} I; C; \Gamma \vdash_{CFL} e : \tau' \qquad I; \emptyset \vdash \tau \preceq^i_- \tau' : \phi \\ \mathit{dom}(\phi) = \vec{\alpha} \qquad\qquad C \vdash L \leq l \end{array}}{I; C; \Gamma \vdash_{CFL} \mathsf{pack}^i\ e\ L : \exists^l\vec{\alpha}.\tau}$$

$$\text{[LUBL]}\frac{C \vdash l_1 \leq l \quad \cdots \quad C \vdash l_n \leq l}{C \vdash (\bigsqcup_{i=1}^{n} l_i) \leq l}$$

$$\text{[LUBR]}\frac{j \in \{1, \ldots, n\}}{C \vdash l_j \leq (\bigsqcup_{i=1}^{n} l_i)}$$

Figure B.1: Extended Subtype Relation

defines a negative instantiation context $\langle C, I, \vec{\alpha}, \tau, \phi, -, i\rangle$

*Since there is a unique i for every [*INST*] or [*PACK*] rule, we define $InstCtx(i, D)$ to be the instantiation context defined at the rule identified by i in the CFL derivation $\mathcal{D}$.*

Definition B.2.5 (Closure) *Let C and I be CFL constraints. The we define the closure of the constraints as C^* under the rewriting rules in Figures 2.6 and 2.10.*

Definition B.2.6 *A set of instantiation constraints I is* normal *if whenever $I \vdash l_1 \preceq_p^i l_2$ and $I \vdash l_3 \preceq_{p'}^j l_4$ with $l_1 \neq l_2$ and $l_3 \neq l_4$, then $l_2 \neq l_3$.*

Definition B.2.7 *A positive instantiation context $\langle C, I, \vec{\alpha}, \vec{l}, \tau, \phi, +, i\rangle$ is* normal *if*

1. *$I; \emptyset \vdash \tau \preceq_p^i \tau' : \phi$*
2. *$\vec{\alpha} \cap \vec{l} = \emptyset$*
3. *$C \rhd_{\vec{\alpha}}^{+} \tau$*
4. *$I \vdash \vec{l} \preceq_{+}^{i} \vec{l}$ and*
5. *I is normal*

Notice that by definition of $C \rhd_{\vec{\alpha}}^{p} \tau$ we also have $\vec{\alpha} \subseteq fl(\tau)$. We will define normal negative instantiation contexts after proving some important lemmas.

In order to show how derivations in $\lambda_{\exists}^{cfl}$ relate to derivations in $\lambda_{\exists}^{cp}$, we will need to relate types $(\forall\vec{\alpha}.\tau, \vec{l})$ with types $\forall\vec{\alpha}[C].\tau$, and similarly for existential types. However, notice that these types may be quantified over different variables—in the $\lambda_{\exists}^{cp}$ type, we may quantify over variables appearing in τ and C, whereas in the $\lambda_{\exists}^{cfl}$ type we only explicitly quantify over variables appearing in τ. To make these match, we need to observe that if a variable appears in C and not in τ or $\vec{l}$, then it is an intermediate variable—the only thing that we really need to capture is how it induces constraints among variables that appear in τ. Hence we add to our system formal joins $(\bigsqcup_{i=1}^{n} l_i)$ of label variables. In the course of the proof, we will replace all intermediate variables in with joins among the variables in τ, the variables in $\vec{l}$, and constants. Figure B.1 gives additional rules we use when checking $C \vdash l_1 \leq l_2$ in addition to containment $\{l_1 \leq l_2\} \in C$. For the remainder of this section, we write $C \vdash C'$ if for all $\{l_1 \leq l_2\} \in C'$ we have $C \vdash l_1 \leq l_2$.

Formally, we define $\Phi_S(l) = \{l' \in S \cup L \mid C^* \vdash l' \leq l\}$. For a set of labels S, we then define a substitution

$$\psi_S(l) = \begin{cases} l & l \in S \cup L \\ \bigsqcup \Phi_S(l) & \textit{otherwise} \end{cases}$$

Finally, for a set of labels S, we define $C^*_S = \psi_S(C*)$, i.e., we replace labels in C^* that are not in S by the least-upper bound of labels in S that flow to it.

Lemma B.2.8 *If $S \subseteq S'$, then $C^*_{S'} \vdash C*_S$.*

Proof: Pick some $l \in S$. Then in C^*_S, there are two cases. Either l is mapped to itself, if $l \in S$, or l is mapped to $\bigsqcup \Phi_S(l)$. Now suppose $C^* \vdash l_1 \leq l_2$. If $l_1 \in S$ and $l_2 \in S$ then $C^*_S \vdash l_1 \leq l_2$ and $C*_{S'} \vdash l_1 \leq l_2$ by the above reasoning. If $l_1 \notin S$ and $l_2 \notin S$, then we have $C^*_S \vdash \bigsqcup \Phi_S(l_1) \leq \bigsqcup \Phi_S(l_2)$. Then by [LUBL] and [LUBR], there exists an $l'_2 \in \Phi_S(l_2)$ such that for all $l'_1 \in \Phi_S(l_1)$, we have $C^*_S \vdash l'_1 \leq l'_2$, and notice that $l'_1, l'_2 \in S \subseteq S'$. Thus we have $C^*_{S'} \vdash l'_1 \leq l'_2$. But since this holds for all l'_1 and some l'_2, by [LUBL] and [LUBR] we have $C^*_{S'} \vdash \bigsqcup \Phi_S(l_1) \leq \bigsqcup \Phi_S(l_2)$. The reasoning for the other possibilities for l_1 and l_2 is similar. □

Lemma B.2.9 *If $S \subseteq S'$, then $\psi_S(C^*_{S'}) = C*_S$.*

Lemma B.2.10 *If $\langle C, I, \vec{\alpha}, \vec{l}, \tau, \phi, +, i\rangle$ is a normal positive instantiation context, then $C^* \rhd^p_{\vec{\alpha}} \tau$.*

Proof: We will show one case; the other polarity is similar. Suppose $C^* \vdash l \leq l'$ with $l \neq l'$ and $l \in \vec{\alpha}$. Then we have $I; C \vdash l \rightsquigarrow_m l'$. We need to show that l has polarity $\bar{p}$ in τ. The proof is by induction on the derivation of $I; C \vdash l \rightsquigarrow_m l'$.
Case [LEVEL]. We have $C \vdash l \leq l'$. But then since the instantiation context is normal, $C \rhd^p_{\vec{\alpha}} \tau$, and thus l has polarity $\bar{p}$ in τ.

Case [TRANS]. We have $I; C \vdash l \rightsquigarrow_m l''$ and $I; C \vdash l'' \rightsquigarrow_m l'$. By induction, $I; C \vdash l \rightsquigarrow_m l''$ implies that l has polarity $\bar{p}$ in τ.

Case [CONSTANT]. This case cannot occur, because we assume $l \in \vec{\alpha}$.

Case [MATCH]. We have $I \vdash l_1 \preceq^i_- l$, $I; C \vdash l_1 \rightsquigarrow_m l_2$, and $I \vdash l_2 \preceq^i_+ l'$. Then suppose for a contradiction that $l_1 \neq l$. Since $l \in \vec{\alpha}$ and $\vec{\alpha} \cap \vec{l} = \emptyset$, we have $I \vdash l \preceq^i_{p'} \phi(l)$ with $\phi(l) \neq l$. Then since the instantiation context is normal, we have $l \neq l$, a contradiction. Thus $l_1 = l$. But then we have $I; C \vdash l \rightsquigarrow_m l_2$, and so by induction we have that l has polarity $\bar{p}$ in τ. □

Intuitively, the following lemma shows that subsets of C^* are closed with respect to substitutions ϕ that correspond to instantiation constraints. In order to show this, we extend a substitution ϕ to a substitution $\hat{\phi}$, which is the same as ϕ except that intermediate variables are replaced by joins. We will use this lemma in proving correspondence

between [INST] and [PACK] rules of the two systems. Below we write L for the set of constant labels.

Given a normal positive instantiation context $\langle C, I, \vec{\alpha}, \vec{l}, \tau, \phi, +, i\rangle$, we define $\Phi_i = \Phi_{S_i}$ where $S_i = \vec{\alpha} \cup \vec{l} \cup L$.

Lemma B.2.11 *Let $\langle C, I, \vec{\alpha}, \vec{l}, \tau, \phi, +, i\rangle$ be a normal positive instantiation context. If $\vec{\alpha} \cup \vec{l} \subseteq S'$ and $\phi(S') \subseteq S$, then $C^*_S \vdash \hat{\phi}(C*_{S'})$, where*

$$\hat{\phi}(l) = \begin{cases} \phi(l) & l \in \vec{\alpha} \\ l & l \in \vec{l} \cup L \\ \bigsqcup \hat{\phi}(\Phi_i(l)) & \textit{otherwise} \end{cases}$$

Proof: Suppose $\{l'_1 \leq l'_2\} \in \hat{\phi}(C^*_{S'})$. Then there are $l_1, l_2 \in S'$ such that $\{l_1 \leq l_2\} \in C^*_{S'}$, where $\hat{\phi}(l_1) = l'_1$ and $\hat{\phi}(l_2) = l'_2$, and thus $C^* \vdash l_1 \leq l_2$ by Lemma B.2.8. Notice we can assume $l_1 \neq l_2$, since otherwise the proof is trivial. We can also assume without loss of generality that neither l_1 nor l_2 is a join, because if it is, we can use [LUBL] and [LUBR] to reduce the inequality to a set of inequalities among labels, as in Lemma B.2.8. So then we need to show $C^*_S \vdash l'_1 \leq l'_2$. There are nine possible cases, depending on where each of $l \in \{l_1, l_2\}$ appears:

1. $l \in \vec{l} \cup L$, and so $\hat{\phi}(l) = l$
2. $l \in \vec{\alpha}$, and so $\hat{\phi}(l) = \phi(l)$ (this is disjoint from the first case since the instantiation context is normal)
3. otherwise $\hat{\phi}(l) = \bigsqcup \hat{\phi}(\Phi_i(l))$

We proceed by case analysis.

1. $l_1 \in \vec{l} \cup L$ and $l'_1 = \hat{\phi}(l_1) = l_1$. The cases for l_2 are:
   (a) $l_2 \in \vec{l} \cup L$ and $l'_2 = \hat{\phi}(l_2) = l_2$. Then since $C^* \vdash l_1 \leq l_2$ and $l'_i = l_i$, we have $C^* \vdash l'_1 \leq l_2$. And since $\phi(S') \subseteq S$, we have $l'_1, l'_2 \in S$. Thus $C^*_S \vdash l'_1 \leq l'_2$.
   (b) $l_2 \in \vec{\alpha}$ and $l'_2 = \hat{\phi}(l_2) = \phi(l_2)$. Then since the instantiation context is normal, we have $C \rhd^p_{\vec{\alpha}} \tau$. Then by Lemma B.2.10 we have $C^* \rhd^p_{\vec{\alpha}} \tau$. But then since $C^* \vdash l_1 \leq l_2$ and $l_2 \in \vec{\alpha}$, we know l_2 has polarity p in τ. Then since $l_2 \notin \vec{l}$, by Lemma B.2.3 we have $I \vdash l_2 \preceq^i_+ \phi(l_2)$, since $p \cdot p = +$ and in the instantiation $D = \emptyset$. Since the instantiation context is normal, we either have $l_1 \in \vec{l}$ or $l_1 \in L$, all of which imply $I \vdash l_1 \preceq^i_\pm l_1$ (the former by [INST] or [PACK], and the latter by [CONSTANT]). Then by [MATCH], we have

$$[\text{MATCH}]\frac{I \vdash l_1 \preceq^i_- \phi(l_1) \qquad I; C \vdash l_1 \rightsquigarrow_m l_2 \qquad I \vdash l_2 \preceq^i_+ \phi(l_2)}{I; C \vdash \phi(l_1) \rightsquigarrow_m \phi(l_2)}$$

   and so $C^* \vdash l'_1 \leq l'_2$. Since $\phi(S') \subseteq S$, we have $l'_1, l'_2 \in S$. Thus $C^*_S \vdash l'_1 \leq l'_2$.

(c) Otherwise $l_2' = \hat{\phi}(l_2) = \bigsqcup \hat{\phi}(\Phi_i(l_2))$ Then since $C^* \vdash l_1 \leq l_2$ and either $l_1 \in \vec{l}$, or $l_1 \in L$, we have $l_1 \in \Phi_i(l_2)$. Since $\hat{\phi}(l_1) = l_1$, we have $l_1 \in \hat{\phi}(\Phi_i(l_2))$. But then from [LUBR], we get $C^* \vdash l_1 \leq \hat{\phi}(l_2) = \bigsqcup \hat{\phi}(\Phi_i(l_2))$. And since $\phi(S') \subseteq S$, we have $l_1' \in S$ and thus $C_S^* \vdash l_1' \leq l_2'$.

2. $l_1 \in \vec{\alpha}$. Then $l_1' = \hat{\phi}(l_1) = \phi(l_1)$. The cases for l_2 are:

   (a) $l_2 \in \vec{l} \cup L$ and $\hat{\phi}(l_2) = l_2$. This is analogous to case 1(b). Since the instantiation context is normal, we have $C \rhd^p_{\vec{\alpha}} \tau$. Then by Lemma B.2.10, we have $C^* \rhd^p_{\vec{\alpha}} \tau$. But since $C^* \vdash l_1 \leq l_2$ and $l_1 \in \vec{\alpha}$, we know that l_1 has polarity $\bar{p}$ in τ. Then since $l_1 \notin \vec{l}$, Then by Lemma B.2.3, we have $I \vdash l_1 \preceq^i_- \phi(l_1)$, since $\bar{p} \cdot p = -$ and in the instantiation $D = \emptyset$. Since the instantiation context is normal and either $l_2 \in \vec{l}$ or $l_2 \in L$, we also have $I \vdash l_2 \preceq^i_\pm l_2$. Then by [MATCH], we have

   $$[\text{MATCH}]\frac{I \vdash l_1 \preceq^i_- \hat{\phi}(l_1) \qquad I; C \vdash l_1 \rightsquigarrow_m l_2 \qquad I \vdash l_2 \preceq^i_+ \hat{\phi}(l_2)}{I; C \vdash \hat{\phi}(l_1) \rightsquigarrow_m \hat{\phi}(l_2)}$$

   and so $C^* \vdash l_1' \leq l_2'$. And since $\phi(S') \subseteq S$, we have $l_1', l_2' \in S$. Thus $C_S^* \vdash l_1' \leq l_2'$.

   (b) $l_2 \in \vec{\alpha}$ and $\hat{\phi}(l_2) = \phi(l_2)$. As in 2(a) above, we have $C^* \rhd^p_{\vec{\alpha}} \tau$. Since $C^* \vdash l_1 \leq l_2$, $l_1 \in \vec{\alpha}$, $l_1 \notin \vec{l}$, and $l_2 \in \vec{\alpha}$, we know that l_1 has polarity $\bar{p}$ in τ and l_2 has polarity p in τ, and in the instantiation $D = \emptyset$. Then by Lemma B.2.3, we have $I \vdash l_1 \preceq^i_- l_1'$ and $I \vdash l_2 \preceq^i_+ l_2'$. Then we have

   $$[\text{MATCH}]\frac{I \vdash l_1 \preceq^i_- \hat{\phi}(l_1) \qquad I; C \vdash l_1 \rightsquigarrow_m l_2 \qquad I \vdash l_2 \preceq^i_+ \hat{\phi}(l_2)}{I; C \vdash \hat{\phi}(l_1) \rightsquigarrow_m \hat{\phi}(l_2)}$$

   so $C^* \vdash l_1' \leq l_2'$. And since $\phi(S') \subseteq S$, we have $l_1', l_2' \in S$. Thus $C_S^* \vdash l_1' \leq l_2'$.

   (c) Otherwise $l_2' = \hat{\phi}(l_2) = \bigsqcup \hat{\phi}(\Phi_i(l_2))$. Then since $C^* \vdash l_1 \leq l_2$ and $l_1 \in \vec{\alpha}$, we have $l_1 \in \Phi_i(l_2)$. So then $l_1' = \hat{\phi}(l_1) \in \hat{\phi}(\Phi_i(l_2))$. Then from [LUBR] we have $C^* \vdash l_1' \leq \bigsqcup \hat{\phi}(\Phi_i(l_2))$. And since $\phi(S') \subseteq S$, we have $l_1' \in S$. Therefore $C_S^* \vdash l_1' \leq l_2'$.

3. Otherwise, $l_1' = \hat{\phi}(l_1) = \bigsqcup \hat{\phi}(\Phi_i(l_1))$. The cases for l_2 are:

   (a) $l_2 \in \vec{l} \cup L$ and $\hat{\phi}(l_2) = \phi(l_2) = l_2$. Then since $C^* \vdash l_1 \leq l_2$, we have $C^* \vdash l' \leq l_2$ for all $l' \in \Phi_i(l_1)$ (and $l' \in S'$ by assumption) by [TRANS] and [LUBL]. Moreover, for each $l' \in \Phi_i(l_1)$, there are two cases.

      i. $l' \in \vec{l} \cup L$. Apply case 1(a) to show $C_S^* \vdash \hat{\phi}(l') \leq l_2'$.

      ii. $l' \in \vec{\alpha}$. Apply case 2(a) to show $C_S^* \vdash \hat{\phi}(l') \leq l_2'$.

   Thus for all $l' \in \Phi_i(l_1)$, we have $C_S^* \vdash \hat{\phi}(l') \leq l_2'$. Then by [LUBL], we have $C_S^* \vdash \bigsqcup \hat{\phi}(\Phi_i(l_1)) \leq \hat{\phi}(l_2)$, or $C_S^* \vdash l_1' \leq l_2'$.

(b) $l_2 \in \vec{\alpha}$ and $\hat{\phi}(l_2) = \phi(l_2)$. Since $C^* \vdash l_1 \leq l_2$, we have $C* \vdash l' \leq l_2$ for all $l' \in \Phi_i(l_1)$ by [TRANS] (and $l' \in S'$ by assumption). For each $l' \in \Phi_i(l_1)$, there are two cases. If $l' \in \vec{l} \cup L$, apply case 1(b) to show $C^*_S \vdash \hat{\phi}(l') \leq l'_2$. If $l' \in \vec{\alpha}$, apply case 2(b) to show $C^*_S \vdash \hat{\phi}(l') \leq l'_2$. Then by [LUBL] as before, $C^*_S \vdash l'_1 \leq l'_2$.

(c) Otherwise $l'_2 = \hat{\phi}(l_2) = \bigsqcup \hat{\phi}(\Phi_i(l_2))$. Then since $C^* \vdash l_1 \leq l_2$, we have $C^* \vdash l' \leq l_2$ for all $l' \in \Phi_i(l_1)$ by [TRANS]. But then $\Phi_i(l_1) \subseteq \Phi_i(l_2) \subseteq S'$. Therefore $\hat{\phi}(\Phi_i(l_1)) \subseteq \hat{\phi}(\Phi_i(l_2)) \subseteq S$. Then by [LUBR] we have $C^* \vdash l' \leq (\bigsqcup \hat{\phi}(\Phi_i(l_2)))$ for all $l' \in \hat{\phi}(\Phi_i(l_1))$, and so by [LUBL] we have $C^* \vdash (\bigsqcup \hat{\phi}(\Phi_i(l_1))) \leq (\bigsqcup \hat{\phi}(\Phi_i(l_2)))$. Since $\hat{\phi}(\Phi_i(l_1)) \subseteq \hat{\phi}(\Phi_i(l_2)) \subseteq S$, we have $C^*_S \vdash l'_1 \leq l'_2$.

□

Definition B.2.12 *A negative instantiation context $\langle C, I, \vec{\alpha}, \tau, \phi, -, i\rangle$ is* normal *if*

1. $I; \emptyset \vdash \tau \preceq^i_p \tau' : \phi$
2. $C \rhd^-_{\vec{\alpha}} \tau$
3. $\mathit{fl}(\tau') \cap \vec{\alpha} = \emptyset$
4. I *is normal*

Next we define a notion of a *normal* $\lambda^{\mathit{cfl}}_{\exists}$ *derivation*, which intuitively is one that corresponds directly to a derivation in $\lambda^{cp}_{\exists}$.

Definition B.2.13 (Normal $\lambda^{\mathit{cfl}}_{\exists}$ derivation) *A $\lambda^{\mathit{cfl}}_{\exists}$ derivation $\mathcal{D}$ is* normal *if*

1. *Every instantiation context $InstCtx(i, D)$ is normal*
2. *For all universal types $(\forall\vec{\alpha}.\tau, \vec{l})$, it is the case that $\vec{\alpha} = \mathit{fl}(\tau) \setminus \vec{l}$.*
3. *For all existential types $\exists^{\vec{l}}\vec{\alpha}.\tau l$, it is the case that $C^*_{\vec{\alpha}} \rhd^-_{\vec{\alpha}} \tau$, i.e., the constraint sets in translated existential types are always negatively polarized with respect to the base type.*
4. *All polymorphic types created in [*LET*] and [*PACK*] have distinct bound labels $\vec{\alpha}$.*
5. *For every two sub-derivations $\mathcal{D}_1$, $\mathcal{D}_2$ in $\mathcal{D}$, where $\mathcal{D}_1$ is not a part of $\mathcal{D}_2$ and conversely, the only common labels between $\mathcal{D}_1$ and $\mathcal{D}_2$ are in the Γ assumptions and concluding types of $\mathcal{D}_1$ and $\mathcal{D}_2$.*

Notice that every sub-derivation of a normal $\lambda^{\mathit{cfl}}_{\exists}$ derivation is normal.

Lemma B.2.14 *If $I; C; \Gamma \vdash_{CFL} e : \tau$, then there exists a normal CFL derivation $I'; C'; \Gamma \vdash_{CFL} e : \tau$.*

Proof: We walk through the conditions. Satisfying conditions 4 and 5 is a matter of picking fresh labels wherever possible. Condition 2 is satisfied by construction of [LET] and [FIX]. And condition 1 follows by reasoning similar to [38]. Observe that reasoning similar to Lemma B.2.19 below shows that $C \rhd^{p}_{\vec{\alpha}} \tau$ at uses of [PACK].

The only tricky condition to show is 3. We sketch the proof. Consider the constraints generated in the e_2 sub-derivation portion of [UNPACK]:

$$I; C; \Gamma, x : \tau \vdash_{CFL} e_2 : \tau'$$

Within the body of e_2, we can assume that [SUB] is always applied after x. Thus for any l appearing positively in τ, we will only generate constraints $l \leq l'$, and vice-versa for labels appearing negatively. Thus the constraints generated in this portion of the derivation are negatively polarized with respect to τ and $\vec{\alpha}$, and so far $C^{*}_{\vec{\alpha}} \rhd^{-}_{\vec{\alpha}} \tau$, since transitively closing these constraints does not affect polarity, and neither does restricting to $\vec{\alpha}$.

Otherwise, suppose we have an application of [SUB] with

$$C; \emptyset; \emptyset \vdash \exists^{l'_1} \vec{\alpha_1}.\tau_1 \leq \exists^{l'_2} \vec{\alpha_2}.\tau_2$$

Then let $l_i \in \vec{\alpha_i}$ be labels in the same positions in τ_i. Each occurs with the same polarity. Suppose the l_i appear positively. Then [SUB-INDEX-2] generates the constraint $C \vdash l_1 \leq l_2$. Clearly we have violated the polarity restriction for l_2 in C. However, observe that in $C^{*}_{\vec{\alpha_2}}$, we have that l_1 is the join of no elements, and this holds transitively even with more applications of [SUB], since they can only add lower bounds to l_2 that do not appear in $\vec{\alpha_2}$. (Only [UNPACK] can add constraints in the other direction, and once we unpack something we cannot re-pack it in the same scope). Thus this constraint is vacuous, and we ignore it for computing polarities. (If we did not ignore these constraints, then [LUBL] would allow us to put any label on the right hand side of a constraint, in any constraint system.) Similarly, if the l_i appear negatively, [SUB] generates the constraint $C \vdash l_2 \leq l_1$, but in $C^{*}_{\vec{\alpha_2}}$, we have that l_1 is the join of some elements including l_2, which is again vacuous, and more applications of [SUB] can only add upper bounds to l_2 that do not appear in $\vec{\alpha_2}$.

Otherwise, suppose we have an application of [INST] with

$$I; \emptyset \vdash \exists^{l'_1} \vec{\alpha}.\tau_1 \preceq^{i}_{+} \exists^{l'_2} \vec{\alpha}.\tau_2$$

Then [INST-INDEX-2] generates no constraints, and reasoning about the type $\exists^{l'_1} \vec{\alpha}.\tau_1$ shows that positively occurring labels in τ_1 can only have lower bounds and negatively occurring labels can only have upper bounds.

Finally, otherwise suppose we have an application of [PACK] with

$$I; \emptyset \vdash \exists^{l'_2} \vec{\alpha}.\tau_2 \preceq^{i}_{-} \exists^{l'_1} \vec{\alpha}.\tau_1$$

Then [INST-INDEX-2] generates no constraints, and reasoning about the type $\exists^{l'_1} \vec{\alpha}.\tau_1$ shows that positively occurring labels in τ_1 can only have upper bounds and negatively occurring labels can only have lower bounds.

The cases of uses of [SUB], [PACK], and [INST] deeper in a type are similar.

□

$$
\begin{array}{rcl}
\Psi_{C,I}(intl) & = & intl \\
\Psi_{C,I}(\tau_1 \rightarrow^l \tau_2) & = & \Psi_{C,I}(\tau_1) \rightarrow^l \Psi_{C,I}(\tau_2) \\
\Psi_{C,I}(\tau_1 \times^l \tau_2) & = & \Psi_{C,I}(\tau_1) \times^l \Psi_{C,I}(\tau_2) \\
\Psi_{C,I}(\exists^l \vec{\alpha}.\tau) & = & \exists^l \vec{\alpha}[C^*_{\vec{\alpha}}].(\Psi_{C,I}(\tau)) \\
\Psi_{C,I}(\Gamma, f : (\forall \vec{\alpha}.\tau, \vec{l})) & = & \Psi_{C,I}(\Gamma), f : \forall \vec{\alpha}[C^*_{(\vec{\alpha} \cup \vec{l})}].(\Psi_{C,I}(\tau)) \\
\Psi_{C,I}(\Gamma, x : \tau) & = & \Psi_{C,I}(\Gamma), (\Psi_{C,I}(\tau))
\end{array}
$$

Figure B.2: Translation from $\lambda^{cfl}_{\exists}$ types to $\lambda^{cp}_{\exists}$ types

Finally, we can prove that for every normal $\lambda^{cfl}_{\exists}$ derivation, there exists an equivalent $\lambda^{cp}_{\exists}$ derivation. Intuitively, a program type checks under $\lambda^{cfl}_{\exists}$ constraints I and C, then it should type check under $\lambda^{cp}_{\exists}$ with constraints C^* (this turns out not quite to work; see below). When translating the derivation, we also need to choose the constraint systems for polymorphic $\lambda^{cp}_{\exists}$ types, and these systems are implicit in $\lambda^{cfl}_{\exists}$. Rehof, Fähndrich, and Das [38] choose C^* as the constraint system for all polymorphic types. However, this does not work in our system, because existentials are higher-order. We could translate the type $(\forall \vec{\alpha}.(\exists^{l'} \vec{\alpha'}.\tau), \vec{l})$ to $\forall \vec{\beta}[C^*].(\exists^{l'} \vec{\beta'}[C^*].\tau)$, but when we instantiate the latter, the instantiation might cause substitutions on some of the variables in the C^* of the existential type. Instead, for existentials, we put in a subset of C^* that is restricted to the bound labels in the type. By construction of the $\lambda^{cfl}_{\exists}$ system, these bound labels can never mix with free labels. Similarly, for universal types, we plug in C^* restricted to the bound labels and the free labels of the universal; for universals, free labels do not cause problems, because they are not first-class. Figure B.2 defines a translation function $\Psi_{C,I}$ that takes $\lambda^{cp}_{\exists}$ types and transforms them to $\lambda^{cfl}_{\exists}$ types. For an existential $\exists^l \vec{\alpha}.\tau$, we choose as the $\lambda^{cp}_{\exists}$ constraints $C^*_{\vec{\alpha}}$. The strong hypothesis in [UNPACK] in Figure 4.8 guarantees that this is safe, because quantified labels can never mix with non-quantified labels. For universal types, on the other hand, we allow quantified types to be constrained by non-quantified types, and thus for a type $(\forall \vec{\alpha}.\tau, \vec{l}))$ we choose the constraints $C^*_{(\vec{\alpha} \cup \vec{l})}$. Intuitively, these are exactly the labels that "matter" to a caller of the quantified type—those that are bound in the type and those that may be free in the type. Any other labels (for example, intermediate labels constructed in the function body) are irrelevant except for their effects on $\vec{\alpha}$ and $\vec{l}$.

Lemma B.2.15 *For any substitution ϕ, we have $\phi(\Psi_{C,I}(\tau)) = \Psi_{C,I}(\phi(\tau))$.*

Proof: By induction on the definition of $\Psi_{C,I}$. The interesting cases are existentials and universals. Letting $\phi'(l) = l$ for $l \in \vec{\alpha}$ and $\phi'(l) = \phi(l)$ otherwise, we have

$$
\begin{array}{rcll}
\phi(\Psi_{C,I}(\exists^l \vec{\alpha}.\tau)) & = & \phi(\exists^l \vec{\alpha}[C^*_{\vec{\alpha}}].(\Psi_{C,I}(\tau))) & \\
 & = & \exists^{\phi(l)} \vec{\alpha}[\phi'(C^*_{\vec{\alpha}})].\phi'(\Psi_{C,I}(\tau))) & \\
 & = & \exists^{\phi(l)} \vec{\alpha}[C^*_{\vec{\alpha}}].\phi'(\Psi_{C,I}(\tau))) & \text{by definition of } \phi' \\
 & = & \exists^{\phi(l)} \vec{\alpha}[C^*_{\vec{\alpha}}].\Psi_{C,I}(\phi'(\tau))) & \text{by induction} \\
 & = & \Psi_{C,I}(\phi(\exists^l \vec{\alpha}.\tau)) &
\end{array}
$$

□

Lemma B.2.16 *For any set S, we have $\psi_{(fl(\Gamma)\cup S)}(\Psi_{C,I}(\Gamma)) = \Psi_{C,I}(\Gamma)$.*

Proof: The proof is by induction. Let $\psi = \psi_{(fl(\Gamma)\cup S)}$. For regular types τ in the range of Γ, we have $\psi(\Psi_{C,I}(\tau)) = \Psi_{C,I}(\psi(\tau))$ by Lemma B.2.15. But since τ is in the range of Γ, $\psi(\tau) = \tau$.

For universals, let $\psi'(l) = l$ for $l \in \vec{\alpha}$ and $\psi'(l) = \psi(l)$ otherwise, and then we have

$$\begin{array}{rcll}
\psi(\Psi_{C,I}((\forall\vec{\alpha}.\tau,\vec{l}))) & = & \psi(\forall\vec{\alpha}[C^*_{(\vec{\alpha}\cup\vec{l})}].(\Psi_{C,I}(\tau))) & \\
& = & \forall\vec{\alpha}[\psi'(C^*_{(\vec{\alpha}\cup\vec{l})})].\psi'(\Psi_{C,I}(\tau)) & \\
& = & \forall\vec{\alpha}[C^*_{(\vec{\alpha}\cup\vec{l})}].\psi'(\Psi_{C,I}(\tau)) & \text{Since } \vec{l} \in fl(\Gamma) \\
& = & \forall\vec{\alpha}[C^*_{(\vec{\alpha}\cup\vec{l})}].\Psi_{C,I}(\psi'(\tau)) & \text{by Lemma B.2.15} \\
& = & \Psi_{C,I}(\psi((\forall\vec{\alpha}.\tau,\vec{l}))) &
\end{array}$$

□

Lemma B.2.17 $fl(\Psi_{C,I}(\tau)) = fl(\tau)$

Proof: By induction on the definition of $\Psi_{C,I}$. The only interesting case is $\Psi_{C,I}(\exists^l\vec{\alpha}.\tau) = \exists^l\vec{\alpha}[C^*_{\vec{\alpha}}].(\Psi_{C,I}(\tau))$. By induction, we have $fl(\Psi_{C,I}(\tau)) = fl(\tau)$. Then observe that $fl(C^*_{\vec{\alpha}}) = \vec{\alpha}$. Thus $fl(\exists^l\vec{\alpha}[C^*_{\vec{\alpha}}].(\Psi_{C,I}(\tau))) = \{l\} \cup ((fl(\Psi_{C,I}(\tau)) \cup fl(C^*_{\vec{\alpha}})) \setminus \vec{\alpha}) = \{l\} \cup ((fl(\Psi_{C,I}(\tau)) \setminus \vec{\alpha}) = \{l\} \cup (fl(\tau) \setminus \vec{\alpha}) = fl(\exists^l\vec{\alpha}.\tau)$. □

Lemma B.2.18 *Given types from a normal derivation, $fl(\Psi_{C,I}(\Gamma)) \subseteq fl(\Gamma)$.*

Proof: The interesting case (ignoring the environment and focusing on the $\forall$ type) is $\Psi_{C,I}((\forall\vec{\alpha}.\tau,\vec{l})) = \forall\vec{\alpha}[C^*_{(\vec{\alpha}\cup\vec{l})}].(\Psi_{C,I}(\tau))$. Since the derivation is normal, $\vec{\alpha} = fl(\tau) \setminus \vec{l}$. Thus

$$\begin{array}{rcl}
fl(\forall\vec{\alpha}[C^*_{(\vec{\alpha}\cup\vec{l})}].(\Psi_{C,I}(\tau))) & = & \Big(fl(C^*_{(\vec{\alpha}\cup\vec{l})}) \cup fl(\Psi_{C,I}(\tau))\Big) \setminus \vec{\alpha} \\
& \subseteq & \Big(\vec{\alpha} \cup \vec{l} \cup fl(\tau)\Big) \setminus \vec{\alpha} \\
& = & \vec{l} \cup (fl(\tau) \setminus \vec{\alpha}) \\
& = & fl((\forall\vec{\alpha}.\tau,\vec{l}))
\end{array}$$

□

Lemma B.2.19 *Given a type $\exists^l\vec{\alpha}.\tau$ from a normal $\lambda^{cfl}_{\exists}$ derivation, we have $C^*_{\vec{\alpha}} \rhd^-_{\vec{\alpha}} \tau$, i.e., the constraint sets in translated existential types are always negatively polarized with respect to the base type.*

Proof: □

Lemma B.2.20 *Let $\langle C, I, \vec{\alpha}, \tau, \phi, -, i\rangle$ be a negative instantiation context in a normal $\lambda^{cfl}_{\exists}$ derivation. If $\vec{\alpha} \subseteq S'$ and $\phi(S') \subseteq S$, then $C^*_S \vdash \check{\phi}(C^*_{S'})$, where*

$$\check{\phi}(l) = \begin{cases} \phi(l) & l \in \vec{\alpha} \\ l & l \in L \\ \bigsqcup \check{\phi}(\Phi_i(l)) & \textit{otherwise} \end{cases}$$

Proof: The proof is the same as in Lemma B.2.11, observing that all $l \in \vec{\alpha}$ only appear on the left-hand side of an instantiation constraint, by [INST-INDEX-2] and assumption that the derivation is normal. □

In order to translate subtyping derivations, we also need to translate the D from Figure 4.9 into the D of Figure 4.6. We define $\Psi_{C,I}(\emptyset) = \emptyset$, and $\Psi_{C,I}(D \oplus \vec{\alpha}) = (\Psi_{C,I}(D))[l \mapsto \Psi_{C,I}(D) + 1, \forall l \in \vec{\alpha}]$.

Lemma B.2.21 *If* $C; D; D \vdash l \leq l'$ *then* $C^*_{(l,l')}; \Psi_{C,I}(D) \vdash_{cp} l \leq l'$.

Proof: By induction on $C; D; D \vdash l \leq l'$.

Case [SUB-INDEX-1]. We have

$$[\text{SUB-INDEX-1 } (\lambda^{cfl}_{\exists})]\frac{C \vdash l \leq l'}{C; \emptyset; \emptyset \vdash l \leq l'}$$

Then since $\Psi_{C,I}(\emptyset) = \emptyset$ and $\emptyset(l) = \emptyset(l') = 0$, we have

$$[\text{SUB-LABEL-1 } (\lambda^{cp}_{\exists})]\frac{\begin{array}{c}(\Psi_{C,I}(D))(l) = (\Psi_{C,I}(D))(l') = 0 \\ C^*_{(l,l')} \vdash l \leq l'\end{array}}{C^*_{(l,l')}; \Psi_{C,I}(D) \vdash_{cp} l \leq l'}$$

Case [SUB-INDEX-2]. We have

$$[\text{SUB-INDEX-2 } (\lambda^{cfl}_{\exists})]\frac{C \vdash l_j \leq l_j}{C; D \oplus \{l_1, \ldots, l_n\}; D \oplus \{l_1, \ldots, l_n\} \vdash l_j \leq l_j}$$

Notice that by assumption both D_i's must be the same. Also, notice that $\Psi_{C,I}(D \oplus \{l_1, \ldots, l_n\})(l_j) > 0$ by definition.

$$[\text{SUB-LABEL-2 } (\lambda^{cp}_{\exists})]\frac{(\Psi_{C,I}(D \oplus \{l_1, \ldots, l_n\}))(l) > 0}{C^*_{(l_j)}; \Psi_{C,I}(D \oplus \{l_1, \ldots, l_n\}) \vdash_{cp} l_j \leq l_j}$$

Case [SUB-INDEX-3]. We have

$$[\text{SUB-INDEX-3 } (\lambda^{cfl}_{\exists})]\frac{C; D; D \vdash l \leq l' \quad l \neq l_i \quad l' \neq l_j \quad \forall i, j \in [1..n]}{C; D \oplus \{l_1, \ldots, l_n\}; D \oplus \{l_1, \ldots, l_n\} \vdash l \leq l'}$$

By induction, $C^*_{(l,l')}; \Psi_{C,I}(D) \vdash_{cp} l \leq l'$. Let $\vec{\alpha} = \{l_1, \ldots, l_n\}$. Then since $l, l' \neq l_i$ for any i, we have $(\Psi_{C,I}(D \oplus \vec{\alpha}))(l) = (\Psi_{C,I}(D))(l)$ and $(\Psi_{C,I}(D \oplus \vec{\alpha}))(l') = (\Psi_{C,I}(D))(l')$. Then there are two cases:

1. If $C^*_{(l,l')}; \Psi_{C,I}(D) \vdash_{cp} l \leq l'$ by [SUB-LABEL-1 $(\lambda^{cp}_{\exists})$], then $(\Psi_{C,I}(D))(l) = (\Psi_{C,I}(D))(l') = 0$. But then $(\Psi_{C,I}(D \oplus \vec{\alpha})(l) = (\Psi_{C,I}(D \oplus \vec{\alpha})(l') = 0$, so we have

$$[\text{SUB-LABEL-1 } (\lambda^{cp}_{\exists})]\frac{\begin{array}{c}(\Psi_{C,I}(D \oplus \vec{\alpha}))(l) = (\Psi_{C,I}(D \oplus \vec{\alpha}))(l') = 0 \\ C^*_{(l,l')} \vdash l \leq l'\end{array}}{C^*_{(l,l')}; \Psi_{C,I}(D \oplus \vec{\alpha}) \vdash_{cp} l \leq l'}$$

2. If $C^*_{(l,l')}; \Psi_{C,I}(D) \vdash_{cp} l \leq l'$ by [SUB-LABEL-2 ($\lambda^{cp}_{\exists}$)], then $(\Psi_{C,I}(D))(l) > 0$ and $l = l'$, and therefore $(\Psi_{C,I}(D \oplus \vec{\alpha}))(l) > 0$, and so we have

$$[\text{SUB-LABEL-2 } (\lambda^{cp}_{\exists})]\frac{(\Psi_{C,I}(D \oplus \vec{\alpha}))(l) > 0}{C^*_{(l,l')}; \Psi_{C,I}(D \oplus \vec{\alpha}) \vdash_{cp} l \leq l'}$$

□

Lemma B.2.22 (Subtyping reduction from $\lambda^{cfl}_{\exists}$ to $\lambda^{cp}_{\exists}$) *Let $\mathcal{D}$ be a normal $\lambda^{cfl}_{\exists}$ derivation of $C; D; D \vdash \tau \leq \tau'$. Then $C^*_{(\tau,\tau')}; \Psi_{C,I}(D) \vdash_{cp} \Psi_{C,I}(\tau) \leq \Psi_{C,I}(\tau')$.*

Proof: By induction on the given $\lambda^{cfl}_{\exists}$ derivation.

Case [SUB-INT]. We have

$$[\text{SUB-INT } (\lambda^{cfl}_{\exists})]\frac{C; D; D \vdash l \leq l'}{C; D; D \vdash intl \leq intl'}$$

Then by Lemma B.2.21 we have $C^*_{(l,l')}; \Psi_{C,I}(D) \vdash_{cp} l \leq l'$. But then we have

$$[\text{SUB-INT } (\lambda^{cp}_{\exists})]\frac{C^*_{(l,l')}; \Psi_{C,I}(D) \vdash_{cp} l \leq l'}{C^*_{(l,l')}; \Psi_{C,I}(D) \vdash_{cp} intl \leq intl'}$$

Case [SUB-PAIR], [SUB-FUN]. By induction, using Lemma B.2.21 and the definition of $\Psi_{C,I}$.

Case [SUB-$\exists$]. We have

$$[\text{SUB-}\exists]\frac{\begin{array}{cc} D_1 = D \oplus \vec{\alpha_1} & D_2 = D \oplus \vec{\alpha_2} \\ C; D_1; D_2 \vdash \tau_1 \leq \tau_2 & C; D; D \vdash l_1 \leq l_2 \end{array}}{C; D; D \vdash \exists^{l_1}\vec{\alpha_1}.\tau_1 \leq \exists^{l_2}\vec{\alpha_2}.\tau_2}$$

Let $T = \{l_1, l_2\} \cup (\mathit{fl}(\tau_1) \setminus \vec{\alpha_1}) \cup (\mathit{fl}(\tau_2) \setminus \vec{\alpha_2})$. Let ϕ be an alpha-renaming such that $\phi(\vec{\alpha_2}) = \vec{\alpha_1}$. This is always well-defined by the assumption that the derivation is normal and by the subtyping rules of Figure 4.9. Then $\phi(D_2) = D_1$, and thus since $C; D_1; D_2 \vdash \tau_1 \leq \tau_2$, we have $C; D_1; D_1 \vdash \tau_1 \leq \phi(\tau_2)$. Then by induction we have $C^*_{(\tau_1,\phi(\tau_2))}; \Psi_{C,I}(D_1) \vdash_{cp} \Psi_{C,I}(\tau_1) \leq \Psi_{C,I}(\phi(\tau_2))$. But notice that by [SUB-LABEL-2] in Figure 4.6, no (nontrivial) constraints between variables in $\Psi_{C,I}(D_1)$ are ever generated. Thus we have $C^*_{((\tau_1,\phi(\tau_2))\setminus\vec{\alpha_1})}; \Psi_{C,I}(D_1) \vdash_{cp} \Psi_{C,I}(\tau_1) \leq \Psi_{C,I}(\phi(\tau_2))$. Notice that by definition of ϕ, we have

$$((\mathit{fl}(\tau_1) \setminus \vec{\alpha_1}) \cup (\mathit{fl}(\tau_2) \setminus \vec{\alpha_2})) = ((\mathit{fl}(\tau_1) \setminus \vec{\alpha_1}) \cup (\phi(\mathit{fl}(\tau_2)) \setminus \vec{\alpha_1}))$$

And thus by Lemmas B.2.8 and B.2.15 we have $C^*_T; \Psi_{C,I}(D_1) \vdash_{cp} \Psi_{C,I}(\tau_1) \leq \phi(\Psi_{C,I}(\tau_2))$.

Also by Lemmas B.2.21 and B.2.8 we have $C^*_T, \Psi_{C,I}(D) \vdash l_1 \leq l_2$.

We need to show that $C^*_{\vec{\alpha_1}} \vdash \phi(C^*_{\vec{\alpha_2}})$. Since the derivation is normal, we have $C^*_{\vec{\alpha_2}} \rhd^-_{\vec{\alpha_2}} \tau_2$. Observe that by the subtyping rules in Figure 4.9, for label $l \in \vec{\alpha_2}$, if l has polarity $+$ in τ_2 then $\phi(l) \leq l$, and if l has polarity $-$ in τ_2 then $l \leq \phi(l)$.

Pick a label $l \in \vec{\alpha_2}$. Suppose that $C^*_{\vec{\alpha_2}} \vdash l' \leq l$. Then by definition, l has polarity $-$ in τ_2. Thus $l \leq \phi(l)$. By construction, l' is a join of the constants and labels in $\vec{\alpha_2}$, and by [LUBL], we have that for all labels $l'' \in l'$ we have $C^*_{\vec{\alpha_2}} \vdash l'' \leq l'$. Then l'' has polarity $+$ in τ_2, and thus $\phi(l'') \leq l''$. But then $C^* \vdash \phi(l'') \leq \phi(l)$. And since this holds for all $l'' \in l'$, by [LUBL] we have $C^*_{\vec{\alpha_1}} \vdash \phi(l'') \leq \phi(l)$, since $\phi(l''), \phi(l) \in \vec{\alpha_1}$.

Similarly, Suppose that $C^*_{\vec{\alpha_2}} \vdash l \leq l'$. Then by definition, l has polarity $+$ in τ_2, and hence $\phi(l) \leq l$. By construction, l' is a join of the constants and labels in $\vec{\alpha_2}$. There are two cases. If $C^*_{\vec{\alpha_2}} \vdash l \leq (l \sqcup S)$ by [LUBR] for some set S, there is nothing to prove, since by [LUBR] we have $C^*_{\vec{\alpha_1}} \vdash \phi(l) \leq (\phi(l) \sqcup \phi(S))$. Otherwise, we have $C^*_{\vec{\alpha_2}} \vdash l \leq l''$ for some $l' \in \vec{\alpha_2}$. Then l'' has polarity $-$ in τ_2, and thus $l'' \leq \phi(l'')$. Then $C^* \vdash \phi(l) \leq \phi(l'')$, and thus $C^*_{\vec{\alpha_1}} \vdash \phi(l) \leq \phi(l'')$, since $\phi(l), \phi(l'') \in \vec{\alpha_1}$.

Thus we have $C^*_{\vec{\alpha_1}} \vdash \phi(C{*}_{\vec{\alpha_2}})$. Notice that there is not requirement that these constraints are part of C^*_T, which follows the $\lambda^{cp}_{\exists}$ system pattern that constraints on existential types do not "leak" out to the outer constraint context upon subtyping them.

Finally, by alpha-conversion we have $\exists^{l_2}\vec{\alpha_2}[C^*_{\vec{\alpha_2}}].\tau_2 = \exists^{l_2}\phi(\vec{\alpha_2})[\phi(C^*_{\vec{\alpha_2}})].\phi(\tau_2)$

Thus we have

$$[\text{SUB-}\exists]\frac{\begin{array}{c} C^*_{\vec{\alpha_1}} \vdash \phi(C{*}_{\vec{\alpha_2}}) \\ D_1 = (\Psi_{C,I}(D))[l \mapsto (\Psi_{C,I}(D))(l) + 1, \forall l \in \vec{\alpha_1}] \\ C^*_T; D_1 \vdash \Psi_{C,I}(\tau_1) \leq \phi(\Psi_{C,I}(\tau_2)) \\ C^*_T; \Psi_{C,I}(D) \vdash l_1 \leq l_2 \end{array}}{C^*_T; \Psi_{C,I}(D) \vdash \exists^{l_1}\vec{\alpha_1}[C{*}_{\vec{\alpha_1}}].\Psi_{C,I}(\tau_1) \leq \exists^{l_2}\vec{\alpha_2}[C^*_{\vec{\alpha_2}}].\Psi_{C,I}(\tau_2)}$$

□

Theorem B.2.23 (Reduction from $\lambda^{cfl}_{\exists}$ to $\lambda^{cp}_{\exists}$) *Let $\mathcal{D}$ be a normal $\lambda^{cfl}_{\exists}$ derivation of $I; C; \Gamma \vdash_{CFL} e : \tau$. Then $C^*_{(fl(\Gamma) \cup fl(\tau))}; \Psi_{C,I}(\Gamma) \vdash_{cp} e : \Psi_{C,I}(\tau)$.*

Proof: By induction on the given $\lambda^{cfl}_{\exists}$ derivation. As a shorthand notation in the proof, we define C^*_Γ as $C^*_{fl(\Gamma)}$, $C{*}_\tau$ as $C{*}_{fl(\tau)}$, and we use commas in place of unions when subscripting.

Case [ID]. We have

$$[\text{ID } (\lambda^{cfl}_{\exists})]\frac{}{I; C; \Gamma, x : \tau \vdash_{CFL} x : \tau}$$

Thus trivially

$$[\text{ID } (\lambda^{cp}_{\exists})]\frac{}{C^*_{(\Gamma,\tau)}; \Psi_{C,I}(\Gamma), x : \Psi_{C,I}(\tau) \vdash_{cp} x : \Psi_{C,I}(\tau)}$$

Case [INT]. We have

$$[\text{INT } (\lambda^{cfl}_{\exists})]\frac{C \vdash L \leq l}{I; C; \Gamma \vdash_{CFL} n^L : intl}$$

Then since $C \vdash L \leq l$ and $l \in \mathit{fl}(intl) = \{l\}$ we have $C^*_{(\Gamma,l)} \vdash L \leq l$. Thus

$$[\textsc{Int}\ (\lambda^{cp}_{\exists})]\frac{C^*_{(\Gamma,l)} \vdash L \leq l}{C^*_{(\Gamma,l)}; \Psi_{C,I}(\Gamma) \vdash_{cp} n^L : intl}$$

and $\Psi_{C,I}(intl) = intl$.

Case [LAM]. We have

$$[\textsc{Lam}\ (\lambda^{cfl}_{\exists})]\frac{I; C; \Gamma, x : \tau \vdash_{CFL} e : \tau' \qquad C \vdash L \leq l}{I; C; \Gamma \vdash_{CFL} \lambda x.eL : \tau \rightarrow^l \tau'}$$

Then since $l \in \mathit{fl}(\tau \rightarrow^l \tau')$ we have $C^*_{(\Gamma,\tau,\tau',l)} \vdash L \leq l$. By induction, $C^*_{(\Gamma,\tau,\tau')}; \Psi_{C,I}(\Gamma), x : \Psi_{C,I}(\tau) \vdash_{cp} e : \Psi_{C,I}(\tau')$. Then by Lemmas B.2.8 and B.1.3, we have $C^*_{(\Gamma,\tau,\tau',l)}; \Psi_{C,I}(\Gamma), x : \Psi_{C,I}(\tau) \vdash_{cp} e : \Psi_{C,I}(\tau')$. Thus we have

$$[\textsc{Lam}\ (\lambda^{cp}_{\exists})]\frac{C^*_{(\Gamma,\tau,\tau',l)}; \Psi_{C,I}(\Gamma), x : \Psi_{C,I}(\tau) \vdash_{cp} e : \Psi_{C,I}(\tau') \qquad C^*_{(\Gamma,\tau,\tau',l)} \vdash L \leq l}{C^*_{(\Gamma,\tau,\tau',l)}; \Psi_{C,I}(\Gamma) \vdash_{cp} \lambda x.eL : \Psi_{C,I}(\tau) \rightarrow^l \Psi_{C,I}(\tau')}$$

and $\Psi_{C,I}(\tau \rightarrow^l \tau') = \Psi_{C,I}(\tau) \rightarrow^l \Psi_{C,I}(\tau')$.

Case [APP]. We have

$$[\textsc{App}\ (\lambda^{cfl}_{\exists})]\frac{\begin{array}{c} I; C; \Gamma \vdash_{CFL} e_1 : \tau \rightarrow^l \tau' \\ I; C; \Gamma \vdash_{CFL} e_2 : \tau \\ C \vdash l \leq L \end{array}}{I; C; \Gamma \vdash_{CFL} e_1\ e_2 L : \tau'}$$

Let $\psi = \psi_{(\Gamma,\tau')}$. By induction, $C^*_{(\Gamma,\tau,\tau',l)}; \Psi_{C,I}(\Gamma) \vdash_{cp} e_1 : \Psi_{C,I}(\tau) \rightarrow^l \Psi_{C,I}(\tau')$. Then

$$\psi(C^*_{(\Gamma,\tau,\tau',l)}); \psi(\Psi_{C,I}(\Gamma)) \vdash_{cp} e_1 : \psi(\Psi_{C,I}(\tau) \rightarrow^l \Psi_{C,I}(\tau'))$$

But $\psi(C^*_{(\Gamma,\tau,\tau',l)}) = C*_{(\Gamma,\tau')}$ and $\psi(\Psi_{C,I}(\Gamma)) = \Psi_{C,I}(\Gamma)$ by Lemma B.2.16. Similarly, $\psi(\Psi_{C,I}(\tau')) = \Psi_{C,I}(\tau')$. Thus

$$C^*_{(\Gamma,\tau')}, \Psi_{C,I}(\Gamma) \vdash_{cp} e_1 : \psi(\Psi_{C,I}(\tau)) \rightarrow^{\psi(l)} \Psi_{C,I}(\tau')$$

Also by induction, $C^*_{(\Gamma,\tau)}; \Psi_{C,I}(\Gamma) \vdash_{cp} e_2 : \Psi_{C,I}(\tau)$, and by similar reasoning and Lemma B.2.8 we get $C^*_{(\Gamma,\tau')}; \Psi_{C,I}(\Gamma) \vdash_{cp} e_2 : \psi(\Psi_{C,I}(\tau))$.

Finally, since $C \vdash l \leq L$, we have $C^*_{(\Gamma,\tau')} \vdash \psi(l) \leq \psi(L)$ or $C^*_{(\Gamma,\tau')} \vdash \psi(l) \leq L$.

But then we have

$$[\textsc{App}\ (\lambda^{cp}_{\exists})]\frac{\begin{array}{c} C^*_{(\Gamma,\tau')}; \Psi_{C,I}(\Gamma) \vdash_{cp} e_1 : \psi(\Psi_{C,I}(\tau)) \rightarrow^{\psi(l)} \Psi_{C,I}(\tau') \\ C^*_{(\Gamma,\tau')}; \Psi_{C,I}(\Gamma) \vdash_{cp} e_2 : \psi(\Psi_{C,I}(\tau)) \qquad C^*_{(\Gamma,\tau')} \vdash \psi(l) \leq L \end{array}}{C^*_{(\Gamma,\tau')}; \Psi_{C,I}(\Gamma) \vdash_{cp} e_1\ e_2 L : \Psi_{C,I}(\tau')}$$

Case [PAIR], [PROJ], [COND]. Similar to [APP].

Case [SUB]. We have

$$[\text{SUB } (\lambda_{\exists}^{cfl})]\frac{I; C; \Gamma \vdash_{CFL} e : \tau \qquad C; \emptyset \vdash \tau \leq \tau'}{I; C; \Gamma \vdash_{CFL} e : \tau'}$$

By induction and Lemma B.2.8, we have

$$C^*_{(\Gamma,\tau,\tau')}; \Psi_{C,I}(\Gamma) \vdash_{cp} e : \Psi_{C,I}(\tau)$$

Let $\psi = \psi_{(\Gamma,\tau')}$. Then

$$\psi(C^*_{(\Gamma,\tau,\tau')}); \psi(\Psi_{C,I}(\Gamma)) \vdash_{cp} e : \psi(\Psi_{C,I}(\tau))$$

But by Lemma B.2.16 we have $\psi(\Psi_{C,I}(\Gamma)) = \Psi_{C,I}(\Gamma)$. And $\psi(C^*_{(\Gamma,\tau,\tau')}) = C^*_{(\Gamma,\tau')}$. Thus we have

$$C^*_{(\Gamma,\tau')}; \Psi_{C,I}(\Gamma) \vdash_{cp} e : \psi(\Psi_{C,I}(\tau))$$

Next, by Lemma B.2.22, we have $C^*_{(\tau,\tau')} \vdash_{cp} \Psi_{C,I}(\tau) \leq \Psi_{C,I}(\tau')$. Thus by Lemma B.2.8 we have $C^*_{(\Gamma,\tau,\tau')} \vdash_{cp} \Psi_{C,I}(\tau) \leq \Psi_{C,I}(\tau')$ Then

$$\psi(C^*_{(\Gamma,\tau,\tau')}) \vdash_{cp} \psi(\Psi_{C,I}(\tau)) \leq \psi(\Psi_{C,I}(\tau'))$$

But $\psi(\Psi_{C,I}(\tau')) = \Psi_{C,I}(\psi(\tau'))$ by Lemma B.2.15, and $\Psi_{C,I}(\psi(\tau')) = \Psi_{C,I}(\tau')$ by definition of ψ. And $\psi(C^*_{(\Gamma,\tau,\tau')}) = C*_{(\Gamma,\tau')}$. Thus by Lemma B.2.8 we have

$$C^*_{(\Gamma,\tau')} \vdash_{cp} \psi(\Psi_{C,I}(\tau)) \leq \Psi_{C,I}(\tau')$$

Then we have

$$[\text{SUB } (\lambda_{\exists}^{cp})]\frac{\begin{array}{c} C^*_{(\Gamma,\tau')}; \Psi_{C,I}(\Gamma) \vdash_{cp} e : \psi(\Psi_{C,I}(\tau)) \\ C^*_{(\Gamma,\tau')}; \emptyset \vdash_{cp} \psi(\Psi_{C,I}(\tau)) \leq \Psi_{C,I}(\tau') \end{array}}{C^*_{(\Gamma,\tau')}; \Psi_{C,I}(\Gamma) \vdash_{cp} e : \Psi_{C,I}(\tau')}$$

Case [LET]. We have

$$[\text{LET } (\lambda_{\exists}^{cfl})]\frac{\begin{array}{cc} I; C; \Gamma \vdash_{CFL} e_1 : \tau_1 & I; C; \Gamma, f : (\forall \vec{\alpha}.\tau_1, \vec{l}) \vdash_{CFL} e_2 : \tau_2 \\ \vec{\alpha} = \mathit{fl}(\tau_1) \setminus \vec{l} & \vec{l} = \mathit{fl}(\Gamma) \end{array}}{I; C; \Gamma \vdash_{CFL} \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

By induction, we have $C^*_{(\Gamma,\tau_1)}; \Psi_{C,I}(\Gamma) \vdash_{cp} e_1 : \Psi_{C,I}(\tau_1)$. But $\vec{l} = \mathit{fl}(\Gamma)$, and $\vec{\alpha} = \mathit{fl}(\tau_1) \backslash \vec{l}$. Thus $\mathit{fl}(\Gamma) \cup \mathit{fl}(\tau_1) = \vec{\alpha} \cup \vec{l}$. Therefore $C^*_{(\vec{\alpha},\vec{l})}; \Psi_{C,I}(\Gamma) \vdash_{cp} e_1 : \Psi_{C,I}(\tau_1)$.

Then since $\vec{l} = \mathit{fl}(\Gamma)$, by induction we also have

$$C^*_{(\Gamma,\tau_2)}; \Psi_{C,I}(\Gamma_{cp}), f : \forall \vec{\alpha}[C^*_{(\vec{\alpha},\vec{l})}].(\Psi_{C,I}(\tau_1)) \vdash_{cp} e_2 : \Psi_{C,I}(\tau_2)$$

Finally, by Lemma B.2.17 we have $fl(\Psi_{C,I}(\tau_1)) = fl(\tau_1)$, and by Lemma B.2.18 we have $fl(\Psi_{C,I}(\Gamma)) \subseteq fl(\Gamma)$. Thus

$$\vec{\alpha} = fl(\tau_1) \setminus \vec{l} \subseteq \left(fl(\Psi_{C,I}(\tau_1)) \cup fl(C^*_{(\vec{\alpha},\vec{l})})\right) \setminus fl(\Psi_{C,I}(\Gamma))$$

Therefore we can apply [LET] rule of the $\lambda^{cp}_{\exists}$ system to prove

$$[\text{LET } (\lambda^{cp}_{\exists})]\frac{\begin{array}{c} C^*_{(\vec{\alpha},\vec{l})}; \Psi_{C,I}(\Gamma) \vdash_{cp} e_1 : \Psi_{C,I}(\tau_1) \\ C^*_{(\Gamma,\tau_2)}; \Psi_{C,I}(\Gamma_{cp}), f : \forall\vec{\alpha}[C^*_{(\vec{\alpha},\vec{l})}].(\Psi_{C,I}(\tau_1)) \vdash_{cp} e_2 : \Psi_{C,I}(\tau_2) \\ \vec{\alpha} \subseteq \left(fl(\Psi_{C,I}(\tau_1)) \cup fl(C^*_{(\vec{\alpha},\vec{l})})\right) \setminus fl(\Psi_{C,I}(\Gamma)) \end{array}}{C^*_{(\Gamma,\tau_2)}; \Psi_{C,I}(\Gamma) \vdash_{cp} \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 : \Psi_{C,I}(\tau_2)}$$

Case [FIX]. Similar to [LET] and [INST]

Case [INST]. We have

$$[\text{INST } (\lambda^{cfl}_{\exists})]\frac{\begin{array}{c} I; C; \emptyset \vdash \tau \preceq^i_+ \tau' : \phi \\ dom(\phi) = \vec{\alpha} \qquad I \vdash \vec{l} \preceq^i_+ \vec{l} \qquad I \vdash \vec{l} \preceq^i_- \vec{l} \end{array}}{I; C; \Gamma, f : (\forall\vec{\alpha}.\tau, \vec{l}) \vdash_{CFL} f^i : \tau'}$$

By definition $\Psi_{C,I}((\forall\vec{\alpha}.\tau, \vec{l})) = \forall\vec{\alpha}[C^*_{(\vec{\alpha},\vec{l})}].(\Psi_{C,I}(\tau))$. Notice that $\vec{l}$ is the set of free labels at the point where f was bound by [LET] or [FIX], and that this use of [INST] is nested inside that derivation. Thus $\vec{l} \subseteq fl(\Gamma)$, and by [INST $(\lambda^{cfl}_{\exists})$] we have $\phi(\vec{l}) = \vec{l}$. Further, since $\phi(\tau) = \tau'$ we have $\phi(\vec{\alpha}) \subseteq fl(\tau')$. Thus $\phi(\vec{\alpha} \cup \vec{l}) \subseteq fl(\tau') \cup fl(\Gamma)$. Then by Lemma B.2.11, we have $C^*_{(\Gamma,\tau')} \vdash \hat{\phi}(C*_{(\vec{\alpha},\vec{l})})$.

Thus we can apply the [INST] rule of $\lambda^{cp}_{\exists}$:

$$[\text{INST } (\lambda^{cp}_{\exists})]\frac{C^*_{(\Gamma,\tau')} \vdash \hat{\phi}(C*_{(\vec{\alpha},\vec{l})})}{C^*_{(\Gamma,\tau')}; \Psi_{C,I}(\Gamma), f : \forall\vec{\alpha}[C^*_{(\vec{\alpha},\vec{l})}].(\Psi_{C,I}(\tau)) \vdash_{cp} f^i : \hat{\phi}(\Psi_{C,I}(\tau))}$$

Finally, by Lemma B.2.15 we have $\hat{\phi}(\Psi_{C,I}(\tau)) = \Psi_{C,I}(\hat{\phi}(\tau)) = \Psi_{C,I}(\tau')$,

Case [PACK]. We have

$$[\text{PACK } (\lambda^{cfl}_{\exists})]\frac{\begin{array}{cc} I; C; \Gamma \vdash_{CFL} e : \tau' & I; C; \emptyset \vdash \tau \preceq^i_- \tau' : \phi \\ dom(\phi) = \vec{\alpha} & C \vdash L \leq l \end{array}}{I; C; \Gamma \vdash_{CFL} \mathsf{pack}^i\ e\ L : \exists^l\vec{\alpha}.\tau}$$

Since $C \vdash L \leq l$ we have $C^*_{(\Gamma,l,\tau')} \vdash L \leq l$. By definition $\Psi_{C,I}(\exists^l\vec{\alpha}.\tau) = \exists^l\vec{\alpha}[C^*_{\vec{\alpha}}].(\Psi_{C,I}(\tau))$. By induction and Lemma B.2.8 we have $C^*_{(\Gamma,l,\tau')}, \Psi_{C,I}(\Gamma) \vdash e : \Psi_{C,I}(\tau')$. But $\phi(\tau) = \tau'$, so by Lemma B.2.15 we have $\Psi_{C,I}(\tau') = \check{\phi}(\Psi_{C,I}(\tau))$. Also, since the instantiation context is normal, and since $\phi(\vec{\alpha}) \subseteq fl(\tau')$, by Lemma B.2.20 we have $C^*_{(\Gamma,l,\tau')} \vdash \check{\phi}(C*_{\vec{\alpha}})$.

Putting this together yields

$$[\textsc{Pack}\ (\lambda^{cp}_{\exists})]\frac{\begin{array}{c} C^*_{(\Gamma,l,\tau')}; \Psi_{C,I}(\Gamma) \vdash_{cp} e : \check{\phi}(\Psi_{C,I}(\tau)) \qquad C^*_{(\Gamma,l,\tau')} \vdash \check{\phi}(C^*_{\vec{\alpha}}) \\ C^*_{(\Gamma,l,\tau')} \vdash L \leq l \end{array}}{C^*_{(\Gamma,l,\tau')}; \Psi_{C,I}(\Gamma) \vdash_{cp} \mathsf{pack}^i\ e\ L : \exists^l \vec{\alpha}[C^*_{\vec{\alpha}}].(\Psi_{C,I}(\tau))}$$

Let $\psi = \psi_{(\Gamma,l,(\tau \setminus \vec{\alpha}))}$. Then we have

$$\psi(C^*_{(\Gamma,l,\tau')}); \psi(\Psi_{C,I}(\Gamma)) \vdash_{cp} \mathsf{pack}^i\ e\ L : \psi(\exists^l \vec{\alpha}[C^*_{\vec{\alpha}}].(\Psi_{C,I}(\tau)))$$

Notice that $\mathit{fl}(\tau) \setminus \vec{\alpha} \subseteq \mathit{fl}(\tau')$. Then $\psi(C^*_{(\Gamma,l,\tau')}) = C*_{(\Gamma,l,(\tau \setminus \vec{\alpha}))}$. And by Lemma B.2.16 we have $\psi(\Psi_{C,I}(\Gamma)) = \Psi_{C,I}(\Gamma)$. Finally, $\psi(\exists^l \vec{\alpha}[C^*_{\vec{\alpha}}].(\Psi_{C,I}(\tau))) = \exists^l \vec{\alpha}[C^*_{\vec{\alpha}}].(\Psi_{C,I}(\tau))$, since $\psi(l) = l$ by definition, all the labels in $C^*_{\vec{\alpha}}$ are bound, and since $\mathit{fl}(\Psi_{C,I}(\tau)) = \mathit{fl}(\tau)$ by Lemma B.2.17 and the only unbound labels of τ are those in $\mathit{fl}(\tau) \setminus \vec{\alpha}$, which ψ does not affect by definition. Thus

$$C^*_{(\Gamma,l,(\tau \setminus \vec{\alpha}))}; \Psi_{C,I}(\Gamma) \vdash_{cp} \mathsf{pack}^i\ e\ L : \exists^l \vec{\alpha}[C^*_{\vec{\alpha}}].(\Psi_{C,I}(\tau))$$

Case [UNPACK]. We have

$$[\textsc{Unpack}\ (\lambda^{cfl}_{\exists})]\frac{\begin{array}{c} I; C; \Gamma \vdash_{CFL} e_1 : \exists^l \vec{\alpha}.\tau \qquad I; C; \Gamma, x : \tau \vdash_{CFL} e_2 : \tau' \\ \vec{l} = \mathit{fl}(\Gamma) \cup (\mathit{fl}(\tau) \setminus \vec{\alpha}) \cup \mathit{fl}(\tau') \cup L \qquad \vec{\alpha} \subseteq \mathit{fl}(\tau) \setminus \vec{l} \qquad C \vdash l \leq L \\ \forall l \in \vec{\alpha}, l' \in \vec{l}.(I; C \not\vdash l \rightsquigarrow_m l' \text{ and } I; C \not\vdash l' \rightsquigarrow_m l) \end{array}}{I; C; \Gamma \vdash_{CFL} \mathsf{unpack}\ x = e_1\ \mathsf{in}\ e_2 L : \tau'}$$

By induction and Lemma B.1.2, we have

$$C^*_{(\Gamma,l,(\tau \setminus \vec{\alpha}),\tau')}; \Psi_{C,I}(\Gamma) \vdash e_1 : \exists^l \vec{\alpha}[C^*_{\vec{\alpha}}].\Psi_{C,I}(\tau)$$

Let $\psi = \psi_{(\Gamma,\tau')}$. Then we have

$$\psi(C^*_{(\Gamma,l,(\tau \setminus \vec{\alpha}),\tau')}); \psi(\Psi_{C,I}(\Gamma)) \vdash e_1 : \psi(\exists^l \vec{\alpha}[C^*_{\vec{\alpha}}].\Psi_{C,I}(\tau))$$

Then by Lemma B.2.16 we have $\psi(\Psi_{C,I}(\Gamma)) = \Psi_{C,I}(\Gamma)$. Also, we have $\psi(C^*_{(\Gamma,l,(\tau \setminus \vec{\alpha}))}) = C^*_{(\Gamma,\tau')}$. Finally, notice that $\psi(\exists^l \vec{\alpha}[C^*_{\vec{\alpha}}].\Psi_{C,I}(\tau)) = \exists^{\psi(l)} \vec{\alpha}[C^*_{\vec{\alpha}}].\psi'(\Psi_{C,I}(\tau))$, where $\psi'(l) = l$ if $l \in \vec{\alpha}$ and $\psi'(l) = \psi(l)$ otherwise. Further, all labels in $C^*_{\vec{\alpha}}$ are bound. Thus we have

$$C^*_{(\Gamma,\tau')}; \Psi_{C,I}(\Gamma) \vdash e_1 : \exists^{\psi(l)} \vec{\alpha}[C^*_{\vec{\alpha}}].\psi'(\Psi_{C,I}(\tau))$$

Also by induction $C^*_{(\Gamma,\tau,\tau')}; \Psi_{C,I}(\Gamma), x : \Psi_{C,I}(\tau) \vdash e_2 : \Psi_{C,I}(\tau')$. Then we claim

$$C^*_{(\Gamma,(\tau \setminus \vec{\alpha}),\tau')} \cup C^*_{\vec{\alpha}} \vdash C^*_{(\Gamma,(\tau \setminus \vec{\alpha}),\vec{\alpha},\tau')} = C^*_{(\Gamma,\tau,\tau')}$$

To see why, suppose $C^*_{(\Gamma,\tau,(\tau \setminus \vec{\alpha}),\tau')} \vdash l \leq l'$. Then without loss of generality, assume l and l' are labels rather than joins. If l or l' is in L, then the result holds trivially. Also, if $l, l' \in \mathit{fl}(\Gamma) \cup (\mathit{fl}(\tau) \setminus \vec{\alpha}) \cup \mathit{fl}(\tau')$ or $l, l' \in \vec{\alpha}$, then the result holds trivially. Otherwise, we

have $I; C \vdash l \rightsquigarrow_m l'$ with one of l, l' in $\mathit{fl}(\Gamma) \cup (\mathit{fl}(\tau) \setminus \vec{\alpha}) \cup \mathit{fl}(\tau')$ and one in $\vec{\alpha}$, which is impossible by the last hypothesis of [UNPACK]. Thus by Lemma B.1.2 we have

$$C^*_{(\Gamma,(\tau\setminus\vec{\alpha}),\tau')} \cup C^*_{\vec{\alpha}}; \Psi_{C,I}(\Gamma), x : \Psi_{C,I}(\tau) \vdash e_2 : \Psi_{C,I}(\tau')$$

But then we have

$$\psi'(C^*_{(\Gamma,(\tau\setminus\vec{\alpha}),\tau')} \cup C^*_{\vec{\alpha}}); \psi'(\Psi_{C,I}(\Gamma)), x : \psi'(\Psi_{C,I}(\tau)) \vdash e_2 : \psi'(\Psi_{C,I}(\tau'))$$

ψ' and ψ only differ on $\vec{\alpha}$, and by assumption $\vec{\alpha} \cap \vec{l} = \emptyset$. Thus by Lemma B.2.16 we have $\psi'(\Psi_{C,I}(\Gamma)) = \Gamma$ and $\psi'(\Psi_{C,I}(\tau')) = \Psi_{C,I}(\tau')$. Further, ψ' does not affect $\vec{\alpha}$, so $\psi'(C^*_{(\Gamma,(\tau\setminus\vec{\alpha}),\tau')} \cup C^*_{\vec{\alpha}}) = \psi'(C*_{(\Gamma,(\tau\setminus\vec{\alpha}),\tau')}) \cup C^*_{\vec{\alpha}}$. And $\psi'(C*_{(\Gamma,(\tau\setminus\vec{\alpha}),\tau')}) = C^*_{(\Gamma,\tau')}$, again since $\vec{\alpha} \cap \vec{l} = \emptyset$. Putting this all together, we have

$$C^*_{(\Gamma,\tau')} \cup C^*_{\vec{\alpha}}; \Psi_{C,I}(\Gamma), x : \psi'(\Psi_{C,I}(\tau)) \vdash e_2 : \Psi_{C,I}(\tau')$$

Finally, since $C \vdash l \leq L$, we have $C^*_{(\Gamma,\tau')} \vdash \psi(l) \leq \psi(L)$ or $C^*_{(\Gamma,\tau')} \vdash \psi(l) \leq L$. Also, $\vec{\alpha} \subseteq \mathit{fl}(\tau) \setminus \vec{l}$. By Lemma B.2.18 we have $\mathit{fl}(\Psi_{C,I}(\Gamma)) \subseteq \mathit{fl}(\Gamma)$. By Lemma B.2.17 we have $\mathit{fl}(\Psi_{C,I}(\tau')) = \mathit{fl}(\tau')$. And $\mathit{fl}(C^*_{(\Gamma,\tau')}) \subseteq \mathit{fl}(\Gamma) \cup \mathit{fl}(\tau')$. And since $\vec{l} \supseteq \mathit{fl}(\Gamma) \cup \mathit{fl}(\tau')$ we have

$$\vec{\alpha} \subseteq (\mathit{fl}(\Psi_{C,I}(\tau)) \cup \mathit{fl}(C^*_{\vec{\alpha}})) \setminus \left(fl(\Psi_{C,I}(\Gamma)) \cup fl(C^*_{(\Gamma,\tau')}) \cup fl(\Psi_{C,I}(\tau'))\right)$$

Putting these all together, we get

$$\text{[UNPACK } (\lambda^{cp}_{\exists})\text{]} \frac{\begin{array}{c} C^*_{(\Gamma,\tau')}; \Psi_{C,I}(\Gamma) \vdash_{cp} e_1 : \exists^{\psi(l)}\vec{\alpha}[C*_{\vec{\alpha}}].\psi'(\Psi_{C,I}(\tau)) \qquad C^*_{(\Gamma,\tau')} \vdash \psi(l) \leq L \\ C^*_{(\Gamma,\tau')} \cup C*_{\vec{\alpha}}; \Psi_{C,I}(\Gamma), x : \psi'(\Psi_{C,I}(\tau)) \vdash_{cp} e_2 : \Psi_{C,I}(\tau') \\ \vec{\alpha} \subseteq (\mathit{fl}(\Psi_{C,I}(\tau)) \cup \mathit{fl}(C^*_{\vec{\alpha}})) \setminus \left(fl(\Psi_{C,I}(\Gamma)) \cup fl(C^*_{(\Gamma,\tau')}) \cup fl(\Psi_{C,I}(\tau'))\right) \end{array}}{C^*_{(\Gamma,\tau')}; \Psi_{C,I}(\Gamma) \vdash_{cp} \mathsf{unpack}\ x = e_1\ \mathsf{in}\ e_2 L : \Psi_{C,I}(\tau')}$$

□

Appendix C

Soundness proof for contextual effects

C.1 Additional definitions

We add a typing rule for heap locations:

$$[\text{TLoc}]\frac{\Gamma(r) = \tau}{\Phi_\emptyset; \Gamma \vdash r_L : ref^{\{L\}}(\tau)}$$

We also add a list of evaluation rules that define when a program goes wrong:

$$\text{[CALL-W]}\ \frac{\langle \alpha, \omega, H, e_1\rangle \xrightarrow{\varepsilon_1} \langle \alpha', \omega', H', v\rangle \quad v \neq \lambda x.e}{\langle \alpha, \omega, H, e_1\ e_2\rangle \xrightarrow{\emptyset} \langle \alpha, \omega, H, \mathbf{err}\rangle}$$

$$\text{[IF-W]}\ \frac{\langle \alpha, \omega, H, e_1\rangle \xrightarrow{\varepsilon_1} \langle \alpha_1, \omega_1, H_1, v_1\rangle \quad v_1 \neq n}{\langle \alpha, \omega, H, \mathsf{if0}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3\rangle \xrightarrow{\emptyset} \langle \alpha, \omega, H, \mathbf{err}\rangle}$$

$$\text{[DEREF-H-W]}\ \frac{\langle \alpha, \omega, H, e\rangle \xrightarrow{\varepsilon} \langle \alpha', \omega', H', r_L\rangle \quad r \notin \mathit{dom}(H')}{\langle \alpha, \omega, H, !\, e\rangle \xrightarrow{\emptyset} \langle \alpha, \omega, H, \mathbf{err}\rangle}$$

$$\text{[DEREF-L-W]}\ \frac{\langle \alpha, \omega, H, e\rangle \xrightarrow{\varepsilon} \langle \alpha', \omega', H', r_L\rangle \quad r \in \mathit{dom}(H') \quad L \notin \omega'}{\langle \alpha, \omega, H, !\, e\rangle \xrightarrow{\emptyset} \langle \alpha, \omega, H, \mathbf{err}\rangle}$$

$$\text{[ASSIGN-H-W]}\ \frac{\langle \alpha, \omega, H, e_1\rangle \xrightarrow{\varepsilon_1} \langle \alpha_1, \omega_1, H_1, r_L\rangle \quad \langle \alpha_1, \omega_1, H_1, e_2\rangle \xrightarrow{\varepsilon_2} \langle \alpha_2, \omega_2, H_2, v\rangle \quad r \notin \mathit{dom}(H_2)}{\langle \alpha, \omega, H, e_1 := e_2\rangle \xrightarrow{\emptyset} \langle \alpha, \omega, H, \mathbf{err}\rangle}$$

$$\text{[ASSIGN-L-W]}\ \frac{\langle \alpha, \omega, H, e_1\rangle \xrightarrow{\varepsilon_1} \langle \alpha_1, \omega_1, H_1, r_L\rangle \quad \langle \alpha_1, \omega_1, H_1, e_2\rangle \xrightarrow{\varepsilon_2} \langle \alpha_2, \omega_2, (H_2, r \mapsto v'), v\rangle \quad L \notin \omega_2}{\langle \alpha, \omega, H, e_1 := e_2\rangle \xrightarrow{\emptyset} \langle \alpha, \omega, H, \mathbf{err}\rangle}$$

Definition C.1.1 (Heap Typing) *We say heap H is well-typed under Γ, written $\Gamma \vdash H$, if*

1. *$\mathit{dom}(\Gamma) = \mathit{dom}(H)$ and*
2. *for every $r \in \mathit{dom}(H)$, we have $\Phi_\emptyset; \Gamma \vdash H(r) : \Gamma(r)$.*

C.2 Soundness for standard effects

Theorem C.2.1 (Standard Effect Soundness) *If*

1. *$\Phi; \Gamma \vdash e : \tau$,*
2. *$\Gamma \vdash H$ and*

3. $\langle 1, 1, H, e\rangle \xrightarrow{\varepsilon} \langle 1, 1, H', R\rangle$

then there is a $\Gamma' \supseteq \Gamma$ *such that:*

1. *R is a value* v *for which* $\Phi_\emptyset; \Gamma' \vdash v : \tau$,
2. $\Gamma' \vdash H'$ *and*
3. $\varepsilon \subseteq \Phi^\varepsilon$.

Proof: Proof by induction on the evaluation derivation.

case [ID] **:**

$$[\text{ID}]\frac{}{\langle 1, 1, H, v\rangle \xrightarrow{\emptyset} \langle 1, 1, H, v\rangle}$$

Then obviously v is a value and $\Phi; \Gamma \vdash v : \tau$ is given from hypothesis.

case [CALL] **:**

From the assumptions, we have an evaluation derivation:

$$[\text{CALL}]\frac{\begin{array}{c}\langle 1, 1, H, e_1\rangle \xrightarrow{\varepsilon_1} \langle 1, 1, H_1, \lambda x.e\rangle \\ \langle 1, 1, H_1, e_2\rangle \xrightarrow{\varepsilon_2} \langle 1, 1, H_2, v_2\rangle \\ \langle 1, 1, H_2, [v_2 \mapsto e]x\rangle \xrightarrow{\varepsilon_3} \langle 1, 1, H', v\rangle\end{array}}{\langle 1, 1, H, e_1\ e_2\rangle \xrightarrow{\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3} \langle 1, 1, H', v\rangle}$$

and also a typing derivation $\Phi; \Gamma \vdash e_1\ e_2 : \tau_2$. The typing derivation could be produced either by [TSUB] or [TAPP]. We show how to handle [TSUB] now and ignore it for the other cases:

$$[\text{TSUB}]\frac{\begin{array}{c}\Phi'; \Gamma \vdash e_1\ e_2 : \tau' \\ \tau' \leq \tau \\ \Phi' \leq \Phi\end{array}}{\Phi; \Gamma \vdash e_1\ e_2 : \tau}$$

Assuming the theorem holds for the premise:

$$\Gamma' \supseteq \Gamma \tag{C.1}$$
$$\Gamma' \vdash H_1 \tag{C.2}$$
$$\varepsilon \subseteq \Phi'^\varepsilon \tag{C.3}$$

Then from $\Phi' \leq \Phi$ we get that $\varepsilon \subseteq \Phi'^\varepsilon \subseteq \Phi^\varepsilon$.

In the case that the last rule of the typing derivation is [TAPP]:

$$[\text{TAPP}]\frac{\begin{array}{c}\Phi_1; \Gamma \vdash e_1 : \tau_1 \rightarrow^{\Phi_f} \tau_2 \\ \Phi_2; \Gamma \vdash e_2 : \tau_1 \\ \Phi_1 \rhd \Phi_2 \rhd \Phi_f \hookrightarrow \Phi\end{array}}{\Phi; \Gamma \vdash e_1\ e_2 : \tau_2}$$

We inductively apply the theorem on the first premise:

$$\Gamma_1 \supseteq \Gamma \tag{C.4}$$
$$\Phi_\emptyset; \Gamma_1 \vdash \lambda x.e : \tau_1 \rightarrow^{\Phi_f} \tau_2 \tag{C.5}$$
$$\Gamma_1 \vdash H_1 \tag{C.6}$$
$$\varepsilon_1 \subseteq \Phi_1^\varepsilon \tag{C.7}$$

From the second premise of [TAPP], $\Gamma_1 \supseteq \Gamma$ and Lemma C.3.5 we get $\Phi_2; \Gamma_1 \vdash e_2 : \tau_1$. From this, $\Gamma_1 \vdash H_1$ and the second premise of the [CALL] rule, we apply the theorem inductively and get:

$$\Gamma_2 \supseteq \Gamma_1 \tag{C.8}$$
$$\Phi_\emptyset; \Gamma_2 \vdash v_2 : \tau_1 \tag{C.9}$$
$$\Gamma_2 \vdash H_2 \tag{C.10}$$
$$\varepsilon_2 \subseteq \Phi_2^\varepsilon \tag{C.11}$$

Finally, we apply Lemma C.3.5 to get:

$$\Phi_\emptyset; \Gamma_2 \vdash \lambda x.e : \tau_1 \rightarrow^{\Phi_f} \tau_2 \tag{C.12}$$
$$\Phi_\emptyset; \Gamma_2 \vdash v_2 : \tau_1 \tag{C.13}$$
$$\tag{C.14}$$

From the first we get:

$$[\text{TLAM}]\frac{\Phi_f; \Gamma_2, x : \tau_1 \vdash e : \tau_2}{\Phi_\emptyset; \Gamma_2 \vdash \lambda x.e : \tau_1 \rightarrow^{\Phi_f} \tau_2}$$

From the premise and (C.13) it follows from Lemma C.3.4 that $\Phi_f; \Gamma_2 \vdash [v_2 \mapsto e]x : \tau_2$. Inductively applying the theorem then gives:

$$\Gamma_3 \supseteq \Gamma_2 \tag{C.15}$$
$$\Gamma_3 \vdash H' \tag{C.16}$$
$$\Phi_\emptyset; \Gamma_3 \vdash v : \tau_2 \tag{C.17}$$
$$\varepsilon_3 \subseteq \Phi_f^\varepsilon \tag{C.18}$$

From $\Phi_1 \rhd \Phi_2 \rhd \Phi_f \hookrightarrow \Phi$ we get $\Phi_1^\varepsilon \cup \Phi_2^\varepsilon \cup \Phi_f^\varepsilon = \Phi^\varepsilon$.

Finally, we have shown that:

$$\Gamma_3 \supseteq \Gamma_2 \supseteq \Gamma_1 \supseteq \Gamma \tag{C.19}$$
$$\Gamma_3 \vdash H' \tag{C.20}$$
$$\Phi_\emptyset; \Gamma_3 \vdash v : \tau_2 \tag{C.21}$$
$$\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3 \subseteq \Phi_1^\varepsilon \cup \Phi_2^\varepsilon \cup \Phi_f^\varepsilon = \Phi^\varepsilon \tag{C.22}$$

case [REF] **:**

From assumptions:

$$[\text{TREF}]\frac{\Phi;\Gamma \vdash e : \tau}{\Phi;\Gamma \vdash \mathsf{ref}^{L}\ e : \mathit{ref}^{\{L\}}(\tau)} \tag{C.23}$$

$$\Gamma \vdash H \tag{C.24}$$

$$[\text{REF}]\frac{\begin{array}{c}\langle 1,1,H,e\rangle \xrightarrow{\varepsilon} \langle 1,1,H',v\rangle \\ r \notin \mathit{dom}(H')\end{array}}{\langle 1,1,H,\mathsf{ref}^{L}\ e\rangle \xrightarrow{\varepsilon} \langle 1,1,(H',r \mapsto v),r_L\rangle} \tag{C.25}$$

Let $H'' = (H', r \mapsto v)$.

From the premises of the above derivations we can apply the theorem inductively and get

$$\Gamma' \supseteq \Gamma \tag{C.26}$$

$$\Phi_\emptyset;\Gamma' \vdash v : \tau \tag{C.27}$$

$$\Gamma' \vdash H' \tag{C.28}$$

$$\varepsilon \subseteq \Phi^{\varepsilon} \tag{C.29}$$

From $\Gamma' \vdash H'$ and $r \notin \mathit{dom}(H')$ we have $r \notin \mathit{dom}(\Gamma')$. So, we select $\Gamma'' = \Gamma', r \mapsto \tau$. Obviously $\Gamma'' \supseteq \Gamma' \supseteq \Gamma$, and $\Gamma''(r) = \tau$, from which we get

$$[\text{TLOC}]\frac{\Gamma''(r) = \tau}{\Phi_\emptyset;\Gamma'' \vdash r_L : \mathit{ref}^{\{L\}}(\tau)}$$

Moreover, $H''(r) = v$. Also, $\mathit{dom}(\Gamma'') = \mathit{dom}(\Gamma') \cup \{r\} = \mathit{dom}(H') \cup \{r\} = \mathit{dom}(H'')$. From Lemma C.3.5 we get $\Phi_\emptyset;\Gamma'' \vdash v : \tau$, which means $\Phi_\emptyset;\Gamma'' \vdash H''(r) : \Gamma''(r)$. It follows that $\Gamma'' \vdash H''$.

case [DEREF] **:**

From assumptions:

$$[\text{TDEREF}]\frac{\begin{array}{c}\Phi_1;\Gamma \vdash e : \mathit{ref}^{\varepsilon_1}(\tau) \\ \Phi_2^{\varepsilon} = \varepsilon_1 \\ \Phi_1 \rhd \Phi_2 \hookrightarrow \Phi\end{array}}{\Phi;\Gamma \vdash !\,e : \tau} \tag{C.30}$$

$$\Gamma \vdash H \tag{C.31}$$

$$[\text{DEREF}]\frac{\begin{array}{c}\langle 1,1,H,e\rangle \xrightarrow{\varepsilon} \langle 1,1\cup\{L\},H',r_L\rangle \\ r \in \mathit{dom}(H')\end{array}}{\langle 1,1,H,!\,e\rangle \xrightarrow{\varepsilon\cup\{L\}} \langle 1\cup\{L\},1,H',H'(r)\rangle} \tag{C.32}$$

By applying the theorem inductively we get

$$\Gamma' \supseteq \Gamma \tag{C.33}$$

$$\Gamma' \vdash H' \tag{C.34}$$

$$\Phi_\emptyset;\Gamma' \vdash r_L : \mathit{ref}^{\varepsilon}(\tau) \tag{C.35}$$

$$\varepsilon \subseteq \Phi_1^{\varepsilon} \tag{C.36}$$

From (C.35) and [TLOC] we have:

$$[\text{TLOC}]\frac{\Gamma'(r) = \tau}{\Phi_\emptyset; \Gamma' \vdash r_L : \mathit{ref}^{\{L\}}(\tau)}$$

which means $\{L\} = \varepsilon_1$ and $\Gamma'(r) = \tau$. From $\Phi_2^\varepsilon = \varepsilon_1 = \{L\}$ and $\Phi_1 \rhd \Phi_2 \hookrightarrow \Phi$ we have $\Phi^\varepsilon = \Phi_1^\varepsilon \cup \Phi_2^\varepsilon = \Phi_1^\varepsilon \cup \{L\}$. Also, from $\Gamma' \vdash H'$ and $r \in \mathit{dom}(H')$ we have $\Phi_\emptyset; \Gamma' \vdash H'(r) : \Gamma(r)$.

Then, for Γ' it is the case that:

$$\Gamma' \supseteq \Gamma \tag{C.37}$$
$$\Gamma' \vdash H' \tag{C.38}$$
$$\Phi_\emptyset; \Gamma' \vdash H'(r) : \tau \tag{C.39}$$
$$\varepsilon \cup \{L\} \subseteq \Phi_1^\varepsilon \cup \{L\} = \Phi^\varepsilon \tag{C.40}$$

case [ASSIGN] **:**

From assumption

$$[\text{TASSIGN}]\frac{\begin{array}{c}\Phi_1; \Gamma \vdash e_1 : \mathit{ref}^\varepsilon(\tau) \\ \Phi_2; \Gamma \vdash e_2 : \tau \\ \Phi_3^\varepsilon = \varepsilon \\ \Phi_1 \rhd \Phi_2 \rhd \Phi_3 \hookrightarrow \Phi\end{array}}{\Phi; \Gamma \vdash e_1 := e_2 : \tau} \tag{C.41}$$

$$\Gamma \vdash H \tag{C.42}$$

$$[\text{ASSIGN}]\frac{\begin{array}{c}\langle 1, 1, H, e_1\rangle \xrightarrow{\varepsilon_1} \langle 1, 1, H_1, r_L\rangle \\ \langle 1, 1, H_1, e_2\rangle \xrightarrow{\varepsilon_2} \langle 1, 1 \cup \{L\}, (H_2, r \mapsto v'), v\rangle\end{array}}{\langle 1, 1, H, e_1 := e_2\rangle \xrightarrow{\varepsilon_1 \cup \varepsilon_2 \cup \{L\}} \langle 1 \cup \{L\}, 1, (H_2, r \mapsto v), v\rangle} \tag{C.43}$$

We apply the theorem inductively on the first premise:

$$\Gamma_1 \supseteq \Gamma \tag{C.44}$$
$$\Gamma_1 \vdash H_1 \tag{C.45}$$
$$\Phi_\emptyset; \Gamma_1 \vdash r_L : \mathit{ref}^\varepsilon(\tau) \tag{C.46}$$
$$\varepsilon_1 \subseteq \Phi_1^\varepsilon \tag{C.47}$$

From (C.46) and [TLOC] we have

$$[\text{TLOC}]\frac{\Gamma_1(r) = \tau}{\Phi_\emptyset; \Gamma_1 \vdash r_L : \mathit{ref}^{\{L\}}(\tau)}$$

So $\Phi_3^\varepsilon = \varepsilon = \{L\}$ and $\Gamma_1(r) = \tau$. From $\Gamma_1 \vdash H_1$ we have $\Phi_\emptyset; \Gamma_1 \vdash H_1(r) : \tau$.

We then apply the theorem inductively to the second premise:

$$\Gamma_2 \supseteq \Gamma_1 \tag{C.48}$$
$$\Gamma_2 \vdash (H_2, r \mapsto v') \tag{C.49}$$
$$\Phi_\emptyset; \Gamma_2 \vdash v : \tau \tag{C.50}$$
$$\varepsilon_2 \subseteq \Phi_2^\varepsilon \tag{C.51}$$

Using Lemma C.3.5 we get $\Phi_1; \Gamma_2 \vdash r_L : \mathit{ref}^{\{L\}}(\tau)$. From (C.49) we get $\Phi_\emptyset; \Gamma_2 \vdash v' : \tau$. Therefore, $\mathit{dom}(\Gamma_2) = \mathit{dom}(H_2, r \mapsto v') = \mathit{dom}(H_2, r \mapsto v)$. and for all $r' \in \mathit{dom}(H_2) \cup \{r\}.\Phi_\emptyset; \Gamma_2 \vdash (H_2, r \mapsto v)(r') : \Gamma_2(r)$.

Finally:

$$\Gamma_2 \supseteq \Gamma_1 \supseteq \Gamma \tag{C.52}$$
$$\Gamma_2 \vdash (H_2, r \mapsto v) \tag{C.53}$$
$$\Phi_\emptyset; \Gamma_2 \vdash v : \tau \tag{C.54}$$
$$\varepsilon_1 \cup \varepsilon_2 \cup \{L\} \subseteq \Phi_1^\varepsilon \cup \Phi_2^\varepsilon \cup \Phi_3^\varepsilon = \Phi^\varepsilon \tag{C.55}$$

case [IF-T] **:**

From assumption

$$[\text{TIF}]\frac{\begin{array}{c}\Phi_1; \Gamma \vdash e_1 : \mathit{int} \\ \Phi_2; \Gamma \vdash e_2 : \tau \\ \Phi_2; \Gamma \vdash e_3 : \tau \\ \Phi_1 \rhd \Phi_2 \hookrightarrow \Phi\end{array}}{\Phi; \Gamma \vdash \mathsf{if0}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau} \tag{C.56}$$
$$\Gamma \vdash H \tag{C.57}$$
$$[\text{IF-T}]\frac{\begin{array}{c}\langle 1, 1, H, e_1\rangle \xrightarrow{\varepsilon_1} \langle 1, 1, H_1, v_1\rangle \\ v_1 = 0 \\ \langle 1, 1, H_1, e_2\rangle \xrightarrow{\varepsilon_2} \langle 1, 1, H_2, v\rangle\end{array}}{\langle 1, 1, H, \mathsf{if0}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3\rangle \xrightarrow{\varepsilon_1 \cup \varepsilon_2} \langle 1, 1, H_2, v\rangle} \tag{C.58}$$

We apply the theorem inductively to the first premise:

$$\Gamma_1 \supseteq \Gamma \tag{C.59}$$
$$\Gamma_1 \vdash H_1 \tag{C.60}$$
$$\Phi_\emptyset; \Gamma_1 \vdash v_1 : \mathit{int} \tag{C.61}$$
$$\varepsilon_1 \subseteq \Phi_1^\varepsilon \tag{C.62}$$

By Lemma C.3.5 and the second premise of [TIF] we have $\Phi_2; \Gamma_1 \vdash e_2 : \tau$. So, we can apply the theorem inductively to get:

$$\Gamma_2 \supseteq \Gamma_1 \tag{C.63}$$
$$\Gamma_2 \vdash H_2 \tag{C.64}$$
$$\Phi_\emptyset; \Gamma_2 \vdash v : \tau \tag{C.65}$$
$$\varepsilon_2 \subseteq \Phi_2^\varepsilon \tag{C.66}$$

From $\Phi_1 \rhd \Phi_2 \hookrightarrow \Phi$ we have $\Phi^\varepsilon = \Phi_1^\varepsilon \cup \Phi_2^\varepsilon$

Finally we show

$$\Gamma_2 \supseteq \Gamma_1 \supseteq \Gamma \tag{C.67}$$
$$\Gamma_2 \vdash H_2 \tag{C.68}$$
$$\Phi_\emptyset; \Gamma_2 \vdash v : \tau \tag{C.69}$$
$$\varepsilon_1 \cup \varepsilon_2 \subseteq \Phi_1^\varepsilon \cup \Phi_2^\varepsilon = \Phi^\varepsilon \tag{C.70}$$

case [IF-F] **:**

Similar to the above.

case [LET] **:**

Similar to [CALL].

case [CALL-W] **:**

Assume:

$$\text{[TAPP]}\frac{\begin{array}{c}\Phi_1;\Gamma \vdash e_1 : \tau_1 \rightarrow^{\Phi_f} \tau_2 \\ \Phi_2;\Gamma \vdash e_2 : \tau_1 \\ \Phi_1 \rhd \Phi_2 \rhd \Phi_f \hookrightarrow \Phi\end{array}}{\Phi;\Gamma \vdash e_1\ e_2 : \tau_2} \tag{C.71}$$

$$\Gamma \vdash H \tag{C.72}$$

$$\text{[CALL-W]}\frac{\begin{array}{c}\langle 1,1,H,e_1\rangle \xrightarrow{\varepsilon_1} \langle 1,1,H',v\rangle \\ v \neq \lambda x.e\end{array}}{\langle 1,1,H,e_1\ e_2\rangle \xrightarrow{\emptyset} \langle 1,1,H,\mathbf{err}\rangle} \tag{C.73}$$

We apply the theorem recursively to the first premise:

$$\Gamma' \supseteq \Gamma \tag{C.74}$$

$$\Gamma' \vdash H' \tag{C.75}$$

$$\Phi_\emptyset;\Gamma' \vdash v : \tau_1 \rightarrow^{\Phi_f} \tau_2 \tag{C.76}$$

$$\varepsilon_1 \subseteq \Phi_1^\varepsilon \tag{C.77}$$

Then the only rule that can create a derivation for (C.76) is [TLAM], meaning $v = \lambda x.e$. But we have $v \neq \lambda x.e$, a contradiction. Therefore, there is no derivation for $\langle 1,1,H,e_1\ e_2\rangle \xrightarrow{\emptyset} \langle 1,1,H,\mathbf{err}\rangle$ when $\Gamma \vdash H$ and $\Phi;\Gamma \vdash e_1\ e_2 : \tau_2$.

case [IF-W] **:**

Assume:

$$\text{[TIF]}\frac{\begin{array}{c}\Phi_1;\Gamma \vdash e_1 : \mathit{int} \\ \Phi_2;\Gamma \vdash e_2 : \tau \\ \Phi_2;\Gamma \vdash e_3 : \tau \\ \Phi_1 \rhd \Phi_2 \hookrightarrow \Phi\end{array}}{\Phi;\Gamma \vdash \mathsf{if0}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau} \tag{C.78}$$

$$\Gamma \vdash H \tag{C.79}$$

$$\text{[IF-W]}\frac{\begin{array}{c}\langle 1,1,H,e_1\rangle \xrightarrow{\varepsilon_1} \langle 1,1,H_1,v_1\rangle \\ v_1 \neq n\end{array}}{\langle 1,1,H,\mathsf{if0}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3\rangle \xrightarrow{\emptyset} \langle 1,1,H,\mathbf{err}\rangle} \tag{C.80}$$

We apply the theorem recursively to the first premise:

$$\Gamma' \supseteq \Gamma \tag{C.81}$$
$$\Gamma' \vdash H' \tag{C.82}$$
$$\Phi_\emptyset; \Gamma' \vdash v_1 : int \tag{C.83}$$
$$\varepsilon_1 \subseteq \Phi_1^\varepsilon \tag{C.84}$$

Then the only rule that can create a derivation for (C.83) is [TINT], meaning $v_1 = n$. But we have $v_1 \neq n$, a contradiction.

case [DEREF-H-W] **:**

Proof by contradiction, similar to the previous case. Assume there is a derivation that evaluates to $\mathbf{err}$, under the assumptions:

$$\text{[TDEREF]}\frac{\begin{array}{c}\Phi_1; \Gamma \vdash e : \mathit{ref}^{\varepsilon_1}(\tau) \\ \Phi_2^\varepsilon = \varepsilon_1 \\ \Phi_1 \rhd \Phi_2 \hookrightarrow \Phi\end{array}}{\Phi; \Gamma \vdash\, !\, e : \tau} \tag{C.85}$$
$$\Gamma \vdash H \tag{C.86}$$
$$\text{[DEREF-H-W]}\frac{\begin{array}{c}\langle 1, 1, H, e\rangle \xrightarrow{\varepsilon} \langle 1, 1, H', r_L\rangle \\ r \notin \mathit{dom}(H')\end{array}}{\langle 1, 1, H, !\, e\rangle \xrightarrow{\emptyset} \langle 1, 1, H, \mathbf{err}\rangle} \tag{C.87}$$

We apply the theorem inductively to the premise:

$$\Gamma' \supseteq \Gamma \tag{C.88}$$
$$\Gamma' \vdash H' \tag{C.89}$$
$$\Phi_\emptyset; \Gamma' \vdash r_L : \mathit{ref}^{\varepsilon_1}(\tau) \tag{C.90}$$
$$\varepsilon \subseteq \Phi_1^\varepsilon \tag{C.91}$$
$$\tag{C.92}$$

From (C.89) and [TLOC] we have $r \in \mathit{dom}(\Gamma')$. Then (C.89) gives $r \in \mathit{dom}(H')$. But we have that $r \notin \mathit{dom}(H')$, a contradiction. Therefore, there is no derivation for $\langle 1, 1, H, !\, e\rangle \xrightarrow{\emptyset} \langle 1, 1, H, \mathbf{err}\rangle$ that ends by [DEREF-H-W] when $\Gamma \vdash H$ and $\Phi; \Gamma \vdash e_1\ e_2 : \tau_2$.

case [DEREF-L-W] **:**

$$\text{[DEREF-L-W]}\frac{\begin{array}{c}\langle 1, 1, H, e\rangle \xrightarrow{\varepsilon} \langle 1, 1, H', r_L\rangle \\ r \in \mathit{dom}(H') \\ L \notin 1\end{array}}{\langle 1, 1, H, !\, e\rangle \xrightarrow{\emptyset} \langle 1, 1, H, \mathbf{err}\rangle}$$

It is obvious that this rule cannot be applied, as $L \notin 1$ is tautologically false.

case [ASSIGN-H-W] **:**

$$\text{[TASSIGN]}\frac{\begin{array}{c}\Phi_1;\Gamma \vdash e_1 : \mathit{ref}^{\varepsilon}(\tau)\\ \Phi_2;\Gamma \vdash e_2 : \tau\\ \Phi_3^{\varepsilon} = \varepsilon\\ \Phi_1 \rhd \Phi_2 \rhd \Phi_3 \hookrightarrow \Phi\end{array}}{\Phi;\Gamma \vdash e_1 := e_2 : \tau} \tag{C.93}$$

$$\Gamma \vdash H \tag{C.94}$$

$$\text{[ASSIGN-H-W]}\frac{\begin{array}{c}\langle 1,1,H,e_1\rangle \xrightarrow{\varepsilon_1} \langle 1,1,H_1,r_L\rangle\\ \langle 1,1,H_1,e_2\rangle \xrightarrow{\varepsilon_2} \langle 1,1,H_2,v\rangle\\ r \notin \mathit{dom}(H_2)\end{array}}{\langle 1,1,H,e_1 := e_2\rangle \xrightarrow{\emptyset} \langle 1,1,H,\mathbf{err}\rangle} \tag{C.95}$$

Similarly to the case for [DEREF-H-W], we apply the theorem inductively on the first premise:

$$\Gamma_1 \supseteq \Gamma \tag{C.96}$$

$$\Gamma_1 \vdash H_1 \tag{C.97}$$

$$\Phi_{\emptyset};\Gamma_1 \vdash r_L : \mathit{ref}^{\varepsilon}(\tau) \tag{C.98}$$

$$\varepsilon_1 \subseteq \Phi_1^{\varepsilon} \tag{C.99}$$

From Lemma C.3.5 we get $\Phi_2;\Gamma_1 \vdash e_2 : \tau$ and apply the theorem on the second premise:

$$\Gamma_2 \supseteq \Gamma_1 \tag{C.100}$$

$$\Gamma_2 \vdash H_2 \tag{C.101}$$

$$\Phi_{\emptyset};\Gamma_2 \vdash v : \tau \tag{C.102}$$

$$\varepsilon_2 \subseteq \Phi_2^{\varepsilon} \tag{C.103}$$

From (C.98) and [TLOC] we have $r \in \mathit{dom}(\Gamma_1)$. Then from (C.100) we have that $\mathit{dom}(\Gamma_1) \subseteq \mathit{dom}(\Gamma_2)$ and from (C.101) we get $\mathit{dom}(\Gamma_2) = \mathit{dom}(H_2)$. It follows that $r \in \mathit{dom}(H_2)$. But we have $r \notin \mathit{dom}(H_2)$ from hypothesis, a contradiction. Therefore, there is no derivation for $\langle 1,1,H,e_1 := e_2\rangle \xrightarrow{\emptyset} \langle 1,1,H,\mathbf{err}\rangle$ that ends by [ASSIGN-H-W] when $\Gamma \vdash H$ and $\Phi;\Gamma \vdash e_1 := e_2 : \tau$.

case [ASSIGN-L-W] **:**

$$\text{[ASSIGN-L-W]}\frac{\begin{array}{c}\langle 1,1,H,e_1\rangle \xrightarrow{\varepsilon_1} \langle 1,1,H_1,r_L\rangle\\ \langle 1,1,H_1,e_2\rangle \xrightarrow{\varepsilon_2} \langle 1,1,(H_2,r \mapsto v'),v\rangle\\ L \notin 1\end{array}}{\langle 1,1,H,e_1 := e_2\rangle \xrightarrow{\emptyset} \langle 1,1,H,\mathbf{err}\rangle}$$

It is obvious that this rule cannot be applied, as $L \notin 1$ is tautologically false.

□

C.3 Auxiliarry lemmas and definitions

Lemma C.3.1 (Weakening of evaluation sub-derivations) *Given a derivation* $\langle \alpha, \omega, H, e\rangle \xrightarrow{\varepsilon} \langle \alpha', \omega', H', v\rangle$*, there exists a derivation* $\langle \alpha \cup \alpha_w, \omega \cup \omega_w, H, e\rangle \xrightarrow{\varepsilon} \langle \alpha' \cup \alpha_w, \omega' \cup \omega_w, H', v\rangle$*,*

Proof: Proof by induction on the derivation.

case [ID] **:**

Given

$$[\text{ID}]\frac{}{\langle \alpha, \omega, H, v\rangle \xrightarrow{\emptyset} \langle \alpha, \omega, H, v\rangle}$$

then we can apply [ID] again:

$$[\text{ID}]\frac{}{\langle \alpha \cup \alpha_w, \omega \cup \omega_w, H, v\rangle \xrightarrow{\emptyset} \langle \alpha \cup \alpha_w, \omega \cup \omega_w, H, v\rangle}$$

case [CALL] **:**

$$[\text{CALL}]\frac{\begin{array}{c}\langle \alpha, \omega, H, e_1\rangle \xrightarrow{\varepsilon_1} \langle \alpha_1, \omega_1, H_1, \lambda x.e\rangle \\ \langle \alpha_1, \omega_1, H_1, e_2\rangle \xrightarrow{\varepsilon_2} \langle \alpha_2, \omega_2, H_2, v_2\rangle \\ \langle \alpha_2, \omega_2, H_2, [v_2 \mapsto e]x\rangle \xrightarrow{\varepsilon_3} \langle \alpha', \omega', H', v\rangle\end{array}}{\langle \alpha, \omega, H, e_1\ e_2\rangle \xrightarrow{\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3} \langle \alpha', \omega', H', v\rangle}$$

Assuming the lemma holds for the premises:

$$\langle \alpha \cup \alpha_w, \omega \cup \omega_w, H, e_1\rangle \xrightarrow{\varepsilon_1} \langle \alpha_1 \cup \alpha_w, \omega_1 \cup \omega_w, H_1, \lambda x.e\rangle \quad \text{(C.104)}$$

$$\langle \alpha_1 \cup \alpha_w, \omega_1 \cup \omega_w, H_1, e_2\rangle \xrightarrow{\varepsilon_2} \langle \alpha_2 \cup \alpha_w, \omega_2 \cup \omega_w, H_2, v_2\rangle \quad \text{(C.105)}$$

$$\langle \alpha_2 \cup \alpha_w, \omega_2 \cup \omega_w, H_2, [v_2 \mapsto e]x\rangle \xrightarrow{\varepsilon_3} \langle \alpha' \cup \alpha_w, \omega' \cup \omega_w, H', v\rangle \quad \text{(C.106)}$$

From these, we can apply [CALL] again to get the wanted

$$\langle \alpha \cup \alpha_w, \omega \cup \omega_w, H, e_1\ e_2\rangle \xrightarrow{\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3} \langle \alpha' \cup \alpha_w, \omega' \cup \omega_w, H', v\rangle$$

case [REF] **:**

$$[\text{REF}]\frac{\begin{array}{c}\langle \alpha, \omega, H, e\rangle \xrightarrow{\varepsilon} \langle \alpha', \omega', H', v\rangle \\ r \notin \mathit{dom}(H')\end{array}}{\langle \alpha, \omega, H, \mathsf{ref}^L\ e\rangle \xrightarrow{\varepsilon} \langle \alpha', \omega', (H', r \mapsto v), r_L\rangle}$$

Assuming the lemma holds for the premise:

$$\langle \alpha \cup \alpha_w, \omega \cup \alpha_w, H, e\rangle \xrightarrow{\varepsilon} \langle \alpha' \cup \alpha_w, \omega' \cup \alpha_w, H', v\rangle$$

we can then apply [REF] again to get

$$\langle \alpha \cup \alpha_w, \omega \cup \omega_w, H, \mathsf{ref}^L\ e\rangle \xrightarrow{\varepsilon} \langle \alpha' \cup \alpha_w, \omega' \cup \omega_w, (H, r \mapsto v), r_L\rangle$$

case [DEREF] **:**

$$[\textsc{Deref}]\frac{\begin{array}{c}\langle \alpha, \omega, H, e\rangle \xrightarrow{\varepsilon} \langle \alpha', \omega' \cup \{L\}, H', r_L\rangle \\ r \in \mathit{dom}(H')\end{array}}{\langle \alpha, \omega, H, !\, e\rangle \xrightarrow{\varepsilon \cup \{L\}} \langle \alpha' \cup \{L\}, \omega', H', H'(r)\rangle}$$

Assuming the lemma holds for the premise we have:

$$\langle \alpha \cup \alpha_w, \omega \cup \omega_w, H, e\rangle \xrightarrow{\varepsilon} \langle \alpha' \cup \alpha_w, \omega' \cup \{L\} \cup \omega_w, H', r_L\rangle$$

Then we can apply [DEREF] again to get the weakened derivation.

case [ASSIGN] **:**

case [IF-T] **:**

case [IF-F] **:**

case [LET] **:**

These cases are similar.

□

Lemma C.3.2 (Canonical Derivation) *If and only if* $\langle 1, 1, H, e\rangle \xrightarrow{\varepsilon} \langle 1, 1, H', v\rangle$ *then there exists a derivation* $\langle \emptyset, \omega, H, e\rangle \xrightarrow{\varepsilon} \langle \alpha, \emptyset, H', v\rangle$*, and also* $\omega = \alpha = \varepsilon$*.*

Proof: The only-if case trivially follows from C.3.1 by adding 1 to both α and ω in the given derivation. We prove the if case by induction on the derivation:

case [ID] **:**

Given

$$[\textsc{Id}]\frac{}{\langle 1, 1, H, v\rangle \xrightarrow{\emptyset} \langle 1, 1, H, v\rangle}$$

then it is also the case that

$$[\textsc{Id}]\frac{}{\langle \emptyset, \emptyset, H, v\rangle \xrightarrow{\emptyset} \langle \emptyset, \emptyset, H, v\rangle}$$

case [CALL] **:**

$$[\textsc{Call}]\frac{\begin{array}{c}\langle 1, 1, H, e_1\rangle \xrightarrow{\varepsilon_1} \langle 1, 1, H_1, \lambda x.e\rangle \\ \langle 1, 1, H_1, e_2\rangle \xrightarrow{\varepsilon_2} \langle 1, 1, H_2, v_2\rangle \\ \langle 1, 1, H_2, [v_2 \mapsto e]x\rangle \xrightarrow{\varepsilon_3} \langle 1, 1, H', v\rangle\end{array}}{\langle 1, 1, H, e_1\ e_2\rangle \xrightarrow{\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3} \langle 1, 1, H', v\rangle}$$

Assuming the lemma holds for the premises:

$$\langle \emptyset, \varepsilon_1, H, e_1 \rangle \xrightarrow{\varepsilon_1} \langle \varepsilon_1, \emptyset, H_1, \lambda x.e \rangle \tag{C.107}$$
$$\langle \emptyset, \varepsilon_2, H_1, e_2 \rangle \xrightarrow{\varepsilon_2} \langle \varepsilon_2, \emptyset, H_2, v_2 \rangle \tag{C.108}$$
$$\langle \emptyset, \varepsilon_3, H_2, [v_2 \mapsto e]x \rangle \xrightarrow{\varepsilon_3} \langle \varepsilon_3, \emptyset, H', v \rangle \tag{C.109}$$

Then from Lemma C.3.1 we get:

$$\langle \emptyset, \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3, H, e_1 \rangle \xrightarrow{\varepsilon_1} \langle \varepsilon_1, \varepsilon_2 \cup \varepsilon_3, H_1, \lambda x.e \rangle \tag{C.110}$$
$$\langle \varepsilon_1, \varepsilon_2 \cup \varepsilon_3, H_1, e_2 \rangle \xrightarrow{\varepsilon_2} \langle \varepsilon_1 \cup \varepsilon_2, \varepsilon_3, H_2, v_2 \rangle \tag{C.111}$$
$$\langle \varepsilon_1 \cup \varepsilon_2, \varepsilon_3, H_2, [v_2 \mapsto e]x \rangle \xrightarrow{\varepsilon_3} \langle \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3, \emptyset, H', v \rangle \tag{C.112}$$

Then we can apply [CALL] again to get

$$\langle \emptyset, \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3, H, e_1\ e_2 \rangle \xrightarrow{\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3} \langle \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3, \emptyset, H', v \rangle$$

case [REF] **:**

$$\text{[REF]}\frac{\begin{array}{c}\langle 1, 1, H, e \rangle \xrightarrow{\varepsilon} \langle 1, 1, H', v \rangle \\ r \notin \mathit{dom}(H')\end{array}}{\langle 1, 1, H, \mathsf{ref}^L\ e \rangle \xrightarrow{\varepsilon} \langle 1, 1, (H', r \mapsto v), r_L \rangle}$$

Assuming the lemma holds for the premise, we can apply [REF] again and get

$$\langle \emptyset, \varepsilon, H, \mathsf{ref}^L\ e \rangle \xrightarrow{\varepsilon} \langle \varepsilon, \emptyset, (H, r \mapsto v), r_L \rangle$$

case [DEREF] **:**

$$\text{[DEREF]}\frac{\begin{array}{c}\langle 1, 1, H, e \rangle \xrightarrow{\varepsilon} \langle 1, 1 \cup \{L\}, H', r_L \rangle \\ r \in \mathit{dom}(H')\end{array}}{\langle 1, 1, H, !\, e \rangle \xrightarrow{\varepsilon \cup \{L\}} \langle 1 \cup \{L\}, 1, H', H'(r) \rangle}$$

We write $1 \cup \{L\}$ for clarity in the premise, however $L \in 1$ so $1 \cup \{L\} = 1$. Assuming the lemma holds for the premise we have:

$$\langle \emptyset, \varepsilon, H, e \rangle \xrightarrow{\varepsilon} \langle \varepsilon, \emptyset, H', r_L \rangle$$

We apply Lemma C.3.1 to get

$$\langle \emptyset, \varepsilon \cup \{L\}, H, e \rangle \xrightarrow{\varepsilon} \langle \varepsilon, \emptyset \cup \{L\}, H', r_L \rangle$$

Then, we can apply [DEREF] again to get:

$$\langle \emptyset, \varepsilon \cup \{L\}, H, !\, e \rangle \xrightarrow{\varepsilon \cup \{L\}} \langle \varepsilon \cup \{L\}, \emptyset, H', H'(r) \rangle$$

case [ASSIGN] **:**

$$[\textsc{Assign}]\frac{\begin{array}{c}\langle 1,1,H,e_1\rangle \xrightarrow{\varepsilon_1} \langle 1,1,H_1,r_L\rangle \\ \langle 1,1,H_1,e_2\rangle \xrightarrow{\varepsilon_2} \langle 1,1\cup\{L\},(H_2,r\mapsto v'),v\rangle\end{array}}{\langle 1,1,H,e_1:=e_2\rangle \xrightarrow{\varepsilon_1\cup\varepsilon_2\cup\{L\}} \langle 1\cup\{L\},1,(H_2,r\mapsto v),v\rangle}$$

where obviously $1 \cup \{L\} = 1$, since $L \in 1$. Assuming the lemma holds for the premises we get:

$$\langle \emptyset,\varepsilon_1,H,e_1\rangle \xrightarrow{\varepsilon_1} \langle \varepsilon_1,\emptyset,H_1,r_L\rangle \tag{C.113}$$
$$\langle \emptyset,\varepsilon_2,H_1,e_2\rangle \xrightarrow{\varepsilon_2} \langle \varepsilon_2,\emptyset,(H_2,r\mapsto v'),v\rangle \tag{C.114}$$

Applying Lemma C.3.1 we get:

$$\langle \emptyset,\varepsilon_1\cup\varepsilon_2\cup\{L\},H,e_1\rangle \xrightarrow{\varepsilon_1} \langle \varepsilon_1,\varepsilon_2\cup\{L\},H_1,r_L\rangle \tag{C.115}$$
$$\langle \varepsilon_1,\varepsilon_2\cup\{L\},H_1,e_2\rangle \xrightarrow{\varepsilon_2} \langle \varepsilon_1\cup\varepsilon_2,\{L\},(H_2,r\mapsto v'),v\rangle \tag{C.116}$$

Then we can apply [ASSIGN] again to get:

$$\langle \emptyset,\varepsilon_1\cup\varepsilon_2\cup\{L\},H,e_1:=e_2\rangle \xrightarrow{\varepsilon_1\cup\varepsilon_2\cup\{L\}} \langle \varepsilon_1\cup\varepsilon_2\cup\{L\},\emptyset,(H_2,r\mapsto v),v\rangle$$

case [IF-T] **:**

$$[\textsc{If-T}]\frac{\begin{array}{c}\langle 1,1,H,e_1\rangle \xrightarrow{\varepsilon_1} \langle 1,1,H_1,v_1\rangle \\ v_1 = 0 \\ \langle 1,1,H_1,e_2\rangle \xrightarrow{\varepsilon_2} \langle 1,1,H_2,v\rangle\end{array}}{\langle 1,1,H,\mathsf{if0}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3\rangle \xrightarrow{\varepsilon_1\cup\varepsilon_2} \langle 1,1,H_2,v\rangle}$$

Assuming the lemma holds for the premises we have:

$$\langle \emptyset,\varepsilon_1,H,e_1\rangle \xrightarrow{\varepsilon_1} \langle \varepsilon_1,\emptyset,H_1,v_1\rangle \tag{C.117}$$
$$\langle \emptyset,\varepsilon_2,H_1,e_2\rangle \xrightarrow{\varepsilon_2} \langle \varepsilon_2,\emptyset,H_2,v\rangle \tag{C.118}$$

We can transform these with Lemma C.3.1 to:

$$\langle \emptyset,\varepsilon_1\cup\varepsilon_2,H,e_1\rangle \xrightarrow{\varepsilon_1} \langle \varepsilon_1,\varepsilon_2,H_1,v_1\rangle \tag{C.119}$$
$$\langle \varepsilon_1,\varepsilon_2,H_1,e_2\rangle \xrightarrow{\varepsilon_2} \langle \varepsilon_1\cup\varepsilon_2,\emptyset,H_2,v\rangle \tag{C.120}$$

then we apply [IF-T] again to get:

$$\langle \emptyset,\varepsilon_1\cup\varepsilon_2,H,\mathsf{if0}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3\rangle \xrightarrow{\varepsilon_1\cup\varepsilon_2} \langle \varepsilon_1\cup\varepsilon_2,\emptyset,H_2,v\rangle$$

case [IF-F] **:**

Similar to the above.

case [LET] **:**

Similar to [CALL].

□

Lemma C.3.3 (Adequacy of Semantics) *If*

$$\langle \alpha, \omega, H, e \rangle \xrightarrow{\varepsilon} \langle \alpha', \omega', H', v \rangle$$

then

1. $\alpha' = \alpha \cup \varepsilon$
2. $\omega = \omega' \cup \varepsilon$

Proof: Proof by induction on the evaluation derivation.

case [ID] **:**

$$[\text{ID}]\frac{}{\langle \alpha, \omega, H, v \rangle \xrightarrow{\emptyset} \langle \alpha, \omega, H, v \rangle}$$

Obviously, $\alpha = \alpha \cup \emptyset$, $\omega = \omega \cup \emptyset$.

case [CALL] **:**

$$[\text{CALL}]\frac{\begin{array}{c}\langle \alpha, \omega, H, e_1 \rangle \xrightarrow{\varepsilon_1} \langle \alpha_1, \omega_1, H_1, \lambda x.e \rangle \\ \langle \alpha_1, \omega_1, H_1, e_2 \rangle \xrightarrow{\varepsilon_2} \langle \alpha_2, \omega_2, H_2, v_2 \rangle \\ \langle \alpha_2, \omega_2, H_2, [v_2 \mapsto e]x \rangle \xrightarrow{\varepsilon_3} \langle \alpha', \omega', H', v \rangle\end{array}}{\langle \alpha, \omega, H, e_1\ e_2 \rangle \xrightarrow{\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3} \langle \alpha', \omega', H', v \rangle}$$

Assuming the lemma holds for the premises, we have:

$$\alpha_1 = \alpha \cup \varepsilon_1 \tag{C.121}$$
$$\alpha_2 = \alpha_1 \cup \varepsilon_2 \tag{C.122}$$
$$\alpha' = \alpha_2 \cup \varepsilon_3 \tag{C.123}$$
$$\omega = \omega_1 \cup \varepsilon_1 \tag{C.124}$$
$$\omega_1 = \omega_2 \cup \varepsilon_2 \tag{C.125}$$
$$\omega_2 = \omega' \cup \varepsilon_3 \tag{C.126}$$

From (C.123), (C.122), (C.121) we have $\alpha' = \alpha_2 \cup \varepsilon_3 = (\alpha_1 \cup \varepsilon_2) \cup \varepsilon_3 = \alpha \cup \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3$.

Similarly, from (C.124), (C.125), (C.126) we get: $\omega = \omega' \cup \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3$.

case [REF] **:**

$$[\text{REF}]\frac{\begin{array}{c}\langle \alpha, \omega, H, e\rangle \xrightarrow{\varepsilon} \langle \alpha', \omega', H', v\rangle \\ r \notin \mathit{dom}(H')\end{array}}{\langle \alpha, \omega, H, \mathsf{ref}^{L}\ e\rangle \xrightarrow{\varepsilon} \langle \alpha', \omega', (H', r \mapsto v), r_L\rangle}$$

This case is trivially proven by induction hypothesis for the premise.

case [DEREF] **:**

$$[\text{DEREF}]\frac{\begin{array}{c}\langle \alpha, \omega, H, e\rangle \xrightarrow{\varepsilon} \langle \alpha', \omega' \cup \{L\}, H', r_L\rangle \\ r \in \mathit{dom}(H')\end{array}}{\langle \alpha, \omega, H, !\, e\rangle \xrightarrow{\varepsilon \cup \{L\}} \langle \alpha' \cup \{L\}, \omega', H', H'(r)\rangle}$$

From induction hypothesis we have:

$$\alpha' = \alpha \cup \varepsilon \tag{C.127}$$
$$\omega = (\omega' \cup \{L\}) \cup \varepsilon \tag{C.128}$$

By adding $\{L\}$ to both sides of the above equations and parenthesizing for readability, we get

$$\alpha' \cup \{L\} = \alpha \cup (\varepsilon \cup \{L\}) \tag{C.129}$$
$$\omega = \omega' \cup (\varepsilon \cup \{L\}) \tag{C.130}$$

which proves the case.

case [ASSIGN] **:**

$$[\text{ASSIGN}]\frac{\begin{array}{c}\langle \alpha, \omega, H, e_1\rangle \xrightarrow{\varepsilon_1} \langle \alpha_1, \omega_1, H_1, r_L\rangle \\ \langle \alpha_1, \omega_1, H_1, e_2\rangle \xrightarrow{\varepsilon_2} \langle \alpha_2, \omega_2 \cup \{L\}, (H_2, r \mapsto v'), v\rangle\end{array}}{\langle \alpha, \omega, H, e_1 := e_2\rangle \xrightarrow{\varepsilon_1 \cup \varepsilon_2 \cup \{L\}} \langle \alpha_2 \cup \{L\}, \omega_2, (H_2, r \mapsto v), v\rangle}$$

From the induction hypothesis we have:

$$\alpha_1 = \alpha \cup \varepsilon_1 \tag{C.131}$$
$$\alpha_2 = \alpha_1 \cup \varepsilon_2 \tag{C.132}$$
$$\omega = \omega_1 \cup \varepsilon_1 \tag{C.133}$$
$$\omega_1 = (\omega_2 \cup \{L\}) \cup \varepsilon_2 \tag{C.134}$$

From the first two we have $\alpha_2 = \alpha \cup \varepsilon_1 \cup \varepsilon_2$ therefore $\alpha_2 \cup \{L\} = \alpha \cup \varepsilon_1 \cup \varepsilon_2 \cup \{L\}$.
From the second two we have $\omega = \omega_2 \cup \varepsilon_1 \cup \varepsilon_2 \cup \{L\}$.

case [IF-T] **:**

$$\text{[IF-T]}\frac{\langle \alpha, \omega, H, e_1\rangle \xrightarrow{\varepsilon_1} \langle \alpha_1, \omega_1, H_1, v_1\rangle \qquad v_1 = 0 \qquad \langle \alpha_1, \omega_1, H_1, e_2\rangle \xrightarrow{\varepsilon_2} \langle \alpha_2, \omega_2, H_2, v\rangle}{\langle \alpha, \omega, H, \mathsf{if0}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3\rangle \xrightarrow{\varepsilon_1 \cup \varepsilon_2} \langle \alpha_2, \omega_2, H_2, v\rangle}$$

From induction on the premises we have

$$\alpha_1 = \alpha \cup \varepsilon_1 \tag{C.135}$$

$$\alpha_2 = \alpha_1 \cup \varepsilon_2 \tag{C.136}$$

$$\omega = \omega_1 \cup \varepsilon_1 \tag{C.137}$$

$$\omega_1 = \omega_2 \cup \varepsilon_2 \tag{C.138}$$

The first two give $\alpha_2 = \alpha \cup \varepsilon_1 \cup \varepsilon_2$

The last two give $\omega = \omega_2 \cup \varepsilon_1 \cup \varepsilon_2$

case [IF-F] **:**

Similar to the above.

case [LET] **:**

Similar to [CALL].

□

Lemma C.3.4 (Substitution) *If*

$$\Phi; \Gamma, x : \tau \vdash e : \tau'$$

$$\Phi_\emptyset; \Gamma \vdash v : \tau$$

then

$$\Phi; \Gamma \vdash [v \mapsto e]x : \tau'$$

Proof: Proof is straightforward by induction on the typing derivation. □

Lemma C.3.5 (Weakening of environment) *If* $\Phi; \Gamma \vdash e : \tau$ *and* $\Gamma' \supseteq \Gamma$ *then* $\Phi; \Gamma' \vdash e : \tau$.

Proof: Proof is straightforward by induction on the typing derivation. □

Definition C.3.6 (Canonical typing) *We say that a typing derivation is canonical when*

1. *It ends with [TSub] and the rule above [TSub] is not [TSub] again*
2. *All sub-derivations of the rule above [TSub] are canonical.*

Lemma C.3.7 (Construct canonical typing) *If there exists a typing derivation proving the judgment* $\Phi; \Gamma \vdash e : \tau$ *then there exists a canonical typing derivation that proves it as well.*

Proof: Trivial. □

Definition C.3.8 (Substitution) *Given* $T_e :: [\Phi_e; \Gamma, x : \tau \vdash e : \tau']$ *and a canonical typing derivation* $T_v :: [\Phi_\emptyset; \Gamma \vdash v : \tau]$*, we define a substitution algorithm*

$$\textsc{Subst}\left([T_e], [T_v]\right)$$

that constructs a canonical typing derivation for

$$\Phi_e; \Gamma \vdash [v \mapsto e]x : \tau'$$

based on the substitution lemma.

$$\textsc{Subst}\left([\Phi; \Gamma, x : \tau \vdash n : int], [\Phi_\emptyset; \Gamma \vdash v : \tau]\right) = \left(\begin{array}{c} \textit{[TInt]}\ \dfrac{}{\Phi_\emptyset; \Gamma \vdash n : int} \\ \Phi_\emptyset \leq \Phi \\ \textit{[TSub]}\ \dfrac{int \leq int}{\Phi; \Gamma \vdash n : int} \end{array}\right)$$

$$\textsc{Subst}\left([\Phi; \Gamma, x : \tau \vdash x : \tau], \left[\begin{array}{c} T'_v :: \Phi_\emptyset; \Gamma \vdash v : \tau' \\ \Phi_\emptyset \leq \Phi_\emptyset \\ \textit{[TSub]}\ \dfrac{\tau' \leq \tau}{T_v :: \Phi_\emptyset; \Gamma \vdash v : \tau} \end{array}\right]\right) = \left(\begin{array}{c} T'_v :: \Phi_\emptyset; \Gamma \vdash v : \tau' \\ \Phi_\emptyset \leq \Phi \\ \textit{[TSub]}\ \dfrac{\tau' \leq \tau}{\Phi; \Gamma \vdash v : \tau} \end{array}\right)$$

$$\textsc{Subst}\left(\left[\textit{[TVar]}\ \frac{(\Gamma, x : \tau)(y) = \tau'}{\Phi_\emptyset; \Gamma, x : \tau \vdash y : \tau'}\right], [\Phi_\emptyset; \Gamma \vdash v : \tau]\right) \textit{ where } x \neq y = \left(\begin{array}{c} \textit{[TVar]}\ \dfrac{\Gamma(y) = \tau'}{\Phi_\emptyset; \Gamma \vdash y : \tau'} \\ \Phi_\emptyset \leq \Phi_\emptyset \\ \textit{[TSub]}\ \dfrac{\tau' \leq \tau'}{\Phi_\emptyset; \Gamma \vdash y : \tau'} \end{array}\right)$$

$$\textsc{Subst}\left(\left[\begin{array}{c} T_1 :: \Phi_1; \Gamma, x : \tau \vdash e_1 : \tau_1 \rightarrow^{\Phi_f} \tau_2 \\ T_2 :: \Phi_2; \Gamma, x : \tau \vdash e_2 : \tau_1 \\ \Phi_1 \rhd \Phi_2 \rhd \Phi_f \hookrightarrow \Phi \\ \hline \Phi; \Gamma, x : \tau \vdash e_1\ e_2 : \tau_2 \end{array}\ [\textsc{TApp}]\right], [T_v :: \Phi_\emptyset; \Gamma \vdash v : \tau]\right) =$$

$$\left(\begin{array}{c} \textsc{Subst}\left([T_1], [T_v]\right) \\ \textsc{Subst}\left([T_2], [T_v]\right) \\ \Phi_1 \rhd \Phi_2 \rhd \Phi_f \hookrightarrow \Phi \\ \hline \Phi; \Gamma \vdash [v \mapsto e_1\ e_2]x : \tau_2 \end{array}\ [\textsc{TApp}]\right)$$

The other cases are similar and follow the substitution lemma.

Definition C.3.9 (Weakening) *Similarly to substitution, we define a weakening function, that constructs a weakened typing derivation using a given Γ' (following the structure of the proof of the weakening lemma):*

$$\textsc{Weaken}\left([\Phi; \Gamma \vdash e : \tau], \Gamma'\right) = (\Phi; \Gamma' \vdash e : \tau)$$

Given $\Gamma' \supseteq \Gamma$, we define:

$$\textsc{Weaken}\left([\Phi; \Gamma \vdash n : int], \Gamma'\right) =$$
$$\left(\frac{}{\Phi; \Gamma' \vdash n : int}\ [\textsc{TInt}]\right)$$

$$\textsc{Weaken}\left(\left[\frac{\Gamma(x) = \tau}{\Phi; \Gamma \vdash x : \tau}\ [\textsc{TVar}]\right], \Gamma'\right) =$$
$$\left(\frac{\Gamma'(x) = \tau}{\Phi; \Gamma' \vdash x : \tau}\ [\textsc{TVar}]\right)$$

$$\textsc{Weaken}\left(\left[\begin{array}{c} T_1 :: \Phi_1; \Gamma \vdash e_1 : \tau_1 \rightarrow^{\Phi_f} \tau_2 \\ T_2 :: \Phi_2; \Gamma \vdash e_2 : \tau_1 \\ \Phi_1 \rhd \Phi_2 \rhd \Phi_f \hookrightarrow \Phi \\ \hline \Phi; \Gamma \vdash e_1\ e_2 : \tau_2 \end{array}\ [\textsc{TApp}]\right], \Gamma'\right) =$$

$$\left(\begin{array}{c} \textsc{Weaken}\left([T_1], \Gamma'\right) \\ \textsc{Weaken}\left([T_2], \Gamma'\right) \\ \Phi_1 \rhd \Phi_2 \rhd \Phi_f \hookrightarrow \Phi \\ \hline \Phi; \Gamma' \vdash e_1\ e_2 : \tau_2 \end{array}\ [\textsc{TApp}]\right)$$

The other cases are similar.

C.4 Soundness proof strategy

To prove the soundness of the contextual effect system, we must show that the effect Φ of a term e approximates the trace α and promise ω of its evaluation. Note, however, that as the program reduces to a value, individual subterms might change through substitutions. It is therefore not always obvious which Φ in the typing derivation for the original term corresponds to a subterm produced during evaluation. To make this connection explicit, we define a *typed operational semantics* that annotates each state in the evaluation with a typing derivation. Our semantics is "natural," in the sense that as subterms are modified by substitutions, our semantics "preserves" the Φ associated with them.

Note that since the terms might change during evaluation, the typing derivations that we use to annotate the evaluation need not be parts of the original typing—but the Φ's that show up in the new typings always are. By defining this new semantics, we can easily express soundness for contextual effects: the Φ assigned to an evaluated term by our semantics always over-approximates the α and ω for the term at runtime. To show the soundness property is not vacuous, we also need to show that we can always construct such a typed operational semantics derivation, given any ordinary evaluation derivation and typing derivation.

C.5 Typed operational semantics

Typed evaluations have the form:

$$\langle T, \alpha, \omega, H, e\rangle \xrightarrow{\emptyset} \langle T', \alpha', \omega', H', v\rangle$$

where T is a canonical typing derivation for the expression e that is evaluated:

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

and T' is a canonical typing derivation for the result of the evaluation:

$$\Phi_\emptyset; \Gamma' \vdash v : \tau$$

Since T is a canonical derivation, it must end with an application of [TSUB] which follows the "normal" typing of the value v. Since v is a value, T' can always use $\Phi_\emptyset$ to type it, which simplifies the rules. The new environment Γ' is not in general the same as Γ, because it might contain extra typings for pointers r that are created during the evaluation of e, but we will show that it is always a superset of Γ. The type of the value is always the same as the type τ of e.

Fig. C.1 presents the typed evaluation rule for values. As in the untyped operational semantics, a value v evaluates to itself without changing the state of the heap, or the trace or promise sets. We have added a typing T in the input state and a typing T' in the output state. Note that we list the constraints on typing derivations T and T' after the rule, even though they are actually premises of the rule, to improve readability and reduce the complexity of the presentation. We follow this practice for the rest of the annotated

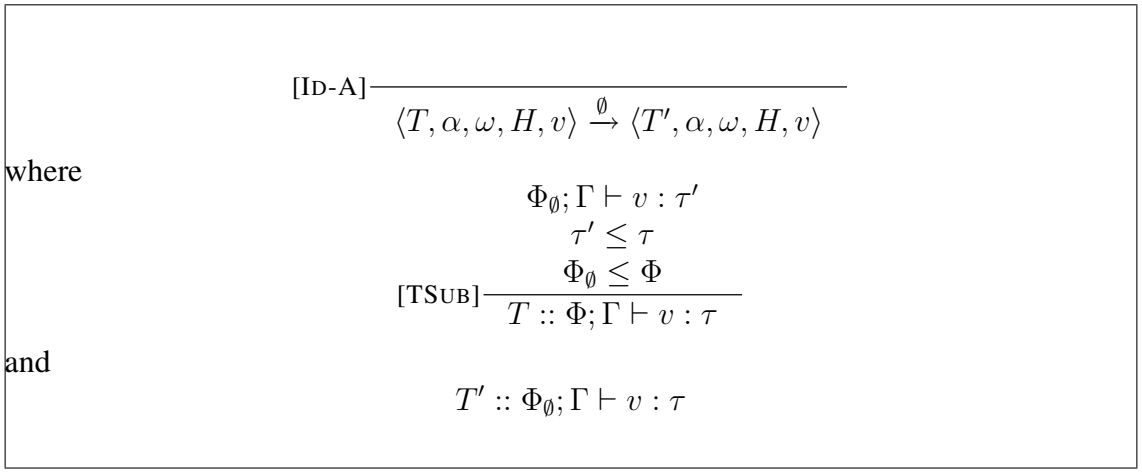


Figure C.1: Typed operational semantics for values

evaluation rules, writing the constraints on typing derivations T that annotate states after the rule as side conditions.

A more interesting case is the rule for typed semantics of the evaluation of function calls [CALL-A], shown in Fig. C.2. As before, we annotate the first and last state of each evaluation (both in the conclusion and the premises of the rule) with a typing. For the conclusion, we require the typing T of the application to be canonical (ending with [TSUB], followed by [TAPP]). We require the typing for the evaluation of e_1 in the first premise to be the same as in the premise of T, this way forcing Φ_1 to be the same between the typing of e_1 in the premise and the typing of e_1 as a subterm of the conclusion. Note that this constraint is essentially the definition of which Φ in the typing of the super-term is the "correct" one to use for a subterm. In other words, this constraint specifies that Φ_1 in the typing T is the effect of the evaluation of e_1. This way, we assign static effects Φ from the static typing of a term to the evaluations of its sub-terms. We can then prove that this assignment is indeed sound, i.e. the Φ_1 we selected for the evaluation of e_1 provides a sound approximation for the actual contextual effect of the evaluation of e_1.

We require the typing T_1' of the result of the first premise to be canonical (ending with [TSUB]) followed by [TLAM] since the result is a lambda-term. The typing T_1' uses environment Γ' which will be a superset of Γ, possibly with extra bindings. We annotate the second premise of the typed evaluation with a typing T_2, constructed by weakening T_1 to use Γ'. The partial function WEAKEN $([]\,,)$ is only applicable when $\Gamma' \supseteq \Gamma$. We cannot directly use T_2 to annotate the initial state of the second premise, because we need to maintain the invariant that the environment in each state types all the locations of the heap at that state. For that reason we constrain T_2' to be a weakened version of T_2, and later prove that Γ' is always a superset of Γ and therefore the weakening is well defined and T_2' is a valid derivation. In any case T_2 and T_2' share the same Φ_2 from the original typing, which we later prove that correctly approximates the contextual effects of the evaluation in the second premise.

As before, we annotate the resulting state of the second premise with a canonical derivation T_2'', which types the resulting value v_2 under environment Γ_2 to produce the same type τ_1 as T_2. As before, we do not need to constrain Γ_2, since we can prove that

$\Gamma_2 \supseteq \Gamma_1$.

Interestingly, the third premise of the [CALL] evaluation does not reduce a term that exists in the super-term, but instead the term that results from substituting x with v in e, where e is the body of the lambda term of the first premise. Therefore, there is no straightforward way to use an existing typing derivation from the sub-derivations of T to annotate the third evaluation. Instead, we construct the correct typing derivation from the premises of T_f (the typing of the lambda term) and T_2'' (the typing of v_2). We define a partial function SUBST $([T], [T'])$ that constructs a typing for $[x \mapsto e]v$ given appropriate typings for e and v, similarly to the weakening function. We later prove that it can be applied and will construct a typing for the term under Φ_f, the effect used to type the body of the function in T_1'. Note that this is one of the few cases where we need the typing that annotates the result of a typed evaluation. Finally, we annotate the result of the third premise of the evaluation with T_v, a typing of v under $\Phi_\emptyset$ and Γ_3, which we will show is a superset of Γ_2, to give the same type τ' as T.

Definition C.5.1 (Consistent type state) *A typed operational semantics state*

$$\langle T, \alpha, \omega, H, e \rangle$$

where

$$T :: \Phi; \Gamma \vdash e : \tau$$

is consistent, written

$$\vdash \langle T, \alpha, \omega, H, e \rangle$$

if

$$\Gamma \vdash H$$

C.6 Soundness proof

Lemma C.6.1 (Environment grows, types do not) *If*

$$\langle T, \alpha, \omega, H, e \rangle \xrightarrow{\varepsilon} \langle T_v, \alpha', \omega', H', v \rangle$$

and

$$T :: \Phi; \Gamma \vdash e : \tau$$

and

$$T_v :: \Phi'; \Gamma' \vdash v : \tau'$$

then

$$\Gamma' \supseteq \Gamma$$

and

$$\tau = \tau'$$

$$\text{[CALL-A]}\frac{\begin{array}{c}\langle T_1, \alpha, \omega, H, e_1\rangle \xrightarrow{\varepsilon_1} \langle T_1', \alpha_1, \omega_1, H_1, \lambda x.e\rangle \\ \langle T_2', \alpha_1, \omega_1, H_1, e_2\rangle \xrightarrow{\varepsilon_2} \langle T_2'', \alpha_2, \omega_2, H_2, v_2\rangle \\ \langle T_3, \alpha_2, \omega_2, H_2, [v_2 \mapsto e]x\rangle \xrightarrow{\varepsilon_3} \langle T_v, \alpha', \omega', H', v\rangle\end{array}}{\langle T, \alpha, \omega, H, e_1\ e_2\rangle \xrightarrow{\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3} \langle T_v, \alpha', \omega', H', v\rangle}$$

where

$$\text{[TAPP]}\ \frac{\begin{array}{c}T_1 :: \Phi_1; \Gamma \vdash e_1 : \tau_1 \to^{\Phi_f} \tau_2 \\ T_2 :: \Phi_2; \Gamma \vdash e_2 : \tau_1 \\ \Phi_1 \rhd \Phi_2 \rhd \Phi_f \hookrightarrow \Phi\end{array}}{\Phi; \Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\text{[TSUB]}\frac{\tau_2 \leq \tau' \qquad \Phi \leq \Phi'}{T :: \Phi'; \Gamma \vdash e_1\ e_2 : \tau'}$$

$$\text{[TLAM]}\ \frac{T_f :: \Phi_f'; \Gamma_1, x : \tau_1' \vdash e : \tau_2'}{\Phi_\emptyset; \Gamma_1 \vdash \lambda x.e : \tau_1' \to^{\Phi_f'} \tau_2'}$$

$$\text{[TSUB]}\frac{\tau_1' \to^{\Phi_f'} \tau_2' \leq \tau_1 \to^{\Phi_f} \tau_2 \qquad \Phi_\emptyset \leq \Phi_\emptyset}{T_1' :: \Phi_\emptyset; \Gamma_1 \vdash \lambda x.e : \tau_1 \to^{\Phi_f} \tau_2}$$

$$T_2' = \textsc{Weaken}\left([T_2], \Gamma_1\right)$$

$$\text{[TSUB]}\frac{\begin{array}{c}T_{v_2} :: \Phi_\emptyset; \Gamma_2 \vdash v_2 : \tau_1'' \\ \tau_1'' \leq \tau_1 \qquad \Phi_\emptyset \leq \Phi_\emptyset\end{array}}{T_2'' :: \Phi_\emptyset; \Gamma_2 \vdash v_2 : \tau_1}$$

$$T_f' = \textsc{Weaken}\left([T_f], (\Gamma_2, x : \tau_1')\right)$$

$$T_3 \text{ is } \textsc{Subst}\left(\left[\frac{\begin{array}{c}T_f' :: \Phi_f'; \Gamma_2, x : \tau_1' \vdash e : \tau_2' \\ \tau_2' \leq \tau_2 \leq \tau' \\ \Phi_f' \leq \Phi_f\end{array}}{\Phi_f; \Gamma_2, x : \tau_1' \vdash e : \tau'}\right], \left[\frac{\begin{array}{c}T_{v_2} :: \Phi_\emptyset; \Gamma_2 \vdash v_2 : \tau_1'' \\ \tau_1'' \leq \tau_1 \leq \tau_1' \\ \Phi_\emptyset \leq \Phi_\emptyset\end{array}}{\Phi_\emptyset; \Gamma_2 \vdash v_2 : \tau_1'}\right]\right)$$

and

$$T_v :: \Phi_\emptyset; \Gamma_3 \vdash v : \tau'$$

Figure C.2: Typed operational semantics for function call

$$\text{[REF-A]}\ \frac{\langle T_1, \alpha, \omega, H, e\rangle \xrightarrow{\varepsilon} \langle T_v, \alpha', \omega', H', v\rangle \quad r \notin \mathit{dom}(H')}{\langle T, \alpha, \omega, H, \mathsf{ref}^L\ e\rangle \xrightarrow{\varepsilon} \langle T_r, \alpha', \omega', (H', r \mapsto v), r_L\rangle}$$

where

$$\text{[TREF]}\ \frac{T_1 :: \Phi; \Gamma \vdash e : \tau}{\Phi; \Gamma \vdash \mathsf{ref}^L\ e : \mathit{ref}^{\{L\}}(\tau)}$$

$$\text{[TSUB]}\ \frac{\mathit{ref}^{\{L\}}(\tau) \leq \mathit{ref}^{\varepsilon}(\tau') \quad \Phi \leq \Phi'}{T :: \Phi'; \Gamma \vdash \mathsf{ref}^L\ e : \mathit{ref}^{\varepsilon}(\tau')}$$

and

$$T_v :: \Phi_\emptyset; \Gamma' \vdash v : \tau$$

and

$$\text{[TSUB]}\ \frac{\Phi_\emptyset; (\Gamma_1, r \mapsto \tau) \vdash r_L : \mathit{ref}^{\{L\}}(\tau) \quad \Phi_\emptyset \leq \Phi_\emptyset \quad \mathit{ref}^{\{L\}}(\tau) \leq \mathit{ref}^{\varepsilon}(\tau')}{T_r :: \Phi_\emptyset; (\Gamma_1, r \mapsto \tau) \vdash r_L : \mathit{ref}^{\{\varepsilon\}}(\tau')}$$

Figure C.3: Typed operational semantics for reference

Proof: Trivial. □

Lemma C.6.2 (Consistent typed states) *If*

$$\langle T, \alpha, \omega, H, e\rangle \xrightarrow{\varepsilon} \langle T_v, \alpha', \omega', H', v\rangle$$

and

$$\vdash \langle T, \alpha, \omega, H, e\rangle$$

then

$$\vdash \langle T_v, \alpha', \omega', H', v\rangle$$

Proof:

case [ID-A] **:**

Given

$$\langle T, \alpha, \omega, H, v\rangle \xrightarrow{\varepsilon} \langle T, \alpha, \omega, H, v\rangle$$

and

$$\vdash \langle T, \alpha, \omega, H, v\rangle$$

then obviously

$$\vdash \langle T, \alpha, \omega, H, v\rangle$$

$$\text{[DEREF-A]}\ \frac{\langle T_1, \alpha, \omega, H, e\rangle \xrightarrow{\varepsilon} \langle T_r, \alpha', \omega' \cup \{L\}, H', r_L\rangle \quad r \in \mathit{dom}(H')}{\langle T, \alpha, \omega, H, !\, e\rangle \xrightarrow{\varepsilon} \langle T_v, \alpha', \omega', H', v\rangle}$$

where

$$\text{[TDEREF]}\ \frac{T_1 :: \Phi_1; \Gamma \vdash e : \mathit{ref}^{\varepsilon}(\tau') \quad \Phi_2^{\varepsilon} = \varepsilon \quad \Phi_1 \rhd \Phi_2 \hookrightarrow \Phi'}{\Phi'; \Gamma \vdash !\, e : \tau'}$$

$$\text{[TSUB]}\ \frac{\tau' \leq \tau \quad \Phi' \leq \Phi}{T :: \Phi; \Gamma \vdash !\, e : \tau}$$

and

$$\text{[TLOC]}\ \frac{\Gamma'(r) = \tau}{\Phi_\emptyset; \Gamma' \vdash r_L : \mathit{ref}^{\{L\}}(\tau)}$$

$$\text{[TSUB]}\ \frac{\Phi_\emptyset \leq \Phi_\emptyset \quad \mathit{ref}^{\{L\}}(\tau) \leq \mathit{ref}^{\varepsilon}(\tau')}{T_r :: \Phi_\emptyset; \Gamma' \vdash r_L : \mathit{ref}^{\varepsilon}(\tau')}$$

and

$$T_v = \Phi_\emptyset; \Gamma' \vdash v : \tau$$

Figure C.4: Typed operational semantics for dereference

$$\text{[IF-T-A]}\frac{\begin{array}{c}\langle T_1, \alpha, \omega, H, e_1\rangle \xrightarrow{\varepsilon_1} \langle T_{v_1}, \alpha_1, \omega_1, H_1, v_1\rangle \\ v_1 = 0 \qquad \langle T_2', \alpha_1, \omega_1, H, e_2\rangle \xrightarrow{\varepsilon_1} \langle T_v, \alpha_2, \omega_2, H_2, v\rangle\end{array}}{\langle T, \alpha, \omega, H, \mathsf{if0}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3\rangle \xrightarrow{\varepsilon_1 \cup \varepsilon_2} \langle T_v, \alpha_2, \omega_2, H_2, v\rangle}$$

where

$$\text{[TIF]}\frac{\begin{array}{cc}T_1 :: \Phi_1; \Gamma \vdash e_1 : int & T_2 :: \Phi_2; \Gamma \vdash e_2 : \tau \\ T_3 :: \Phi_2; \Gamma \vdash e_3 : \tau & \Phi_1 \rhd \Phi_2 \hookrightarrow \Phi\end{array}}{T :: \Phi; \Gamma \vdash \mathsf{if0}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau}$$

$$T_{v_1} :: \Phi_\emptyset; \Gamma_1 \vdash v_1 : int$$

$$T_2' = \textsc{Weaken}\left([T_2], \Gamma_1\right)$$

$$T_v :: \Phi_\emptyset; \Gamma_2 \vdash v : \tau$$

$$\text{[IF-F-A]}\frac{\begin{array}{c}\langle T_1, \alpha, \omega, H, e_1\rangle \xrightarrow{\varepsilon_1} \langle T_{v_1}, \alpha_1, \omega_1, H_1, v_1\rangle \\ v_1 \neq 0 \qquad \langle T_2', \alpha_1, \omega_1, H, e_3\rangle \xrightarrow{\varepsilon_2} \langle T_v, \alpha_2, \omega_2, H_2, v\rangle\end{array}}{\langle T, \alpha, \omega, H, \mathsf{if0}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3\rangle \xrightarrow{\varepsilon_1 \cup \varepsilon_2} \langle T_v, \alpha_2, \omega_2, H_2, v\rangle}$$

where

$$\text{[TIF]}\frac{\begin{array}{cc}T_1 :: \Phi_1; \Gamma \vdash e_1 : int & T_2 :: \Phi_2; \Gamma \vdash e_2 : \tau \\ T_3 :: \Phi_2; \Gamma \vdash e_3 : \tau & \Phi_1 \rhd \Phi_2 \hookrightarrow \Phi\end{array}}{T :: \Phi; \Gamma \vdash \mathsf{if0}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau}$$

$$T_{v_1} :: \Phi_\emptyset; \Gamma_1 \vdash v_1 : int$$

$$T_2' = \textsc{Weaken}\left([T_2], \Gamma_1\right)$$

$$T_v :: \Phi_\emptyset; \Gamma_2 \vdash v : \tau$$

Figure C.5: Typed operational semantics for conditional

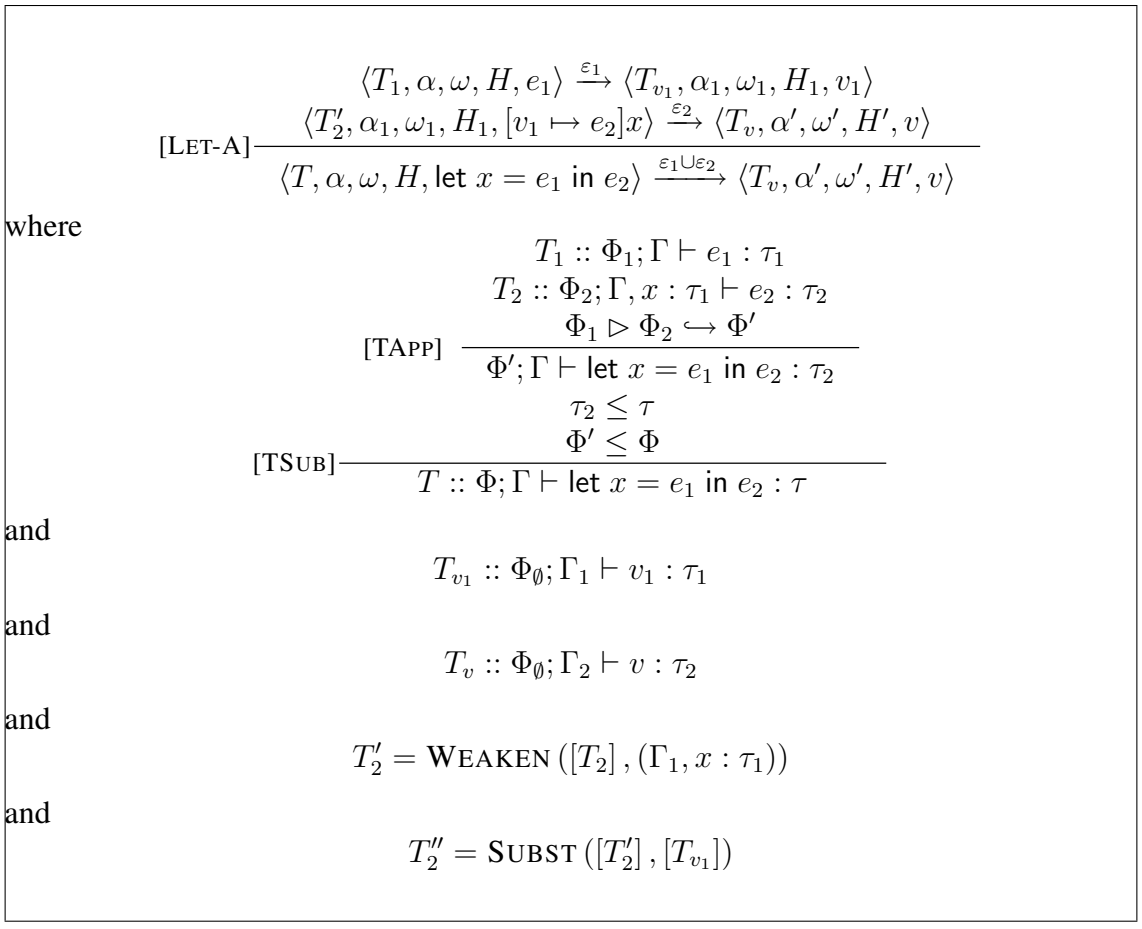


Figure C.6: Typed operational semantics for let

case [CALL-A] **:**

Given

$$[\text{CALL-A}]\frac{\begin{array}{c}\langle T_1, \alpha, \omega, H, e_1\rangle \xrightarrow{\varepsilon_1} \langle T_1', \alpha_1, \omega_1, H_1, \lambda x.e\rangle \\ \langle T_2', \alpha_1, \omega_1, H_1, e_2\rangle \xrightarrow{\varepsilon_2} \langle T_2'', \alpha_2, \omega_2, H_2, v_2\rangle \\ \langle T_3, \alpha_2, \omega_2, H_2, [v_2 \mapsto e]x\rangle \xrightarrow{\varepsilon_3} \langle T_v, \alpha', \omega', H', v\rangle\end{array}}{\langle T, \alpha, \omega, H, e_1\ e_2\rangle \xrightarrow{\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3} \langle T_v, \alpha', \omega', H', v\rangle}$$

where

$$[\text{TSUB}]\frac{[\text{TAPP}]\ \dfrac{\begin{array}{c}T_1 :: \Phi_1; \Gamma \vdash e_1 : \tau_1 \to^{\Phi_f} \tau_2 \\ T_2 :: \Phi_2; \Gamma \vdash e_2 : \tau_1 \\ \Phi_1 \rhd \Phi_2 \rhd \Phi_f \hookrightarrow \Phi\end{array}}{\Phi; \Gamma \vdash e_1\ e_2 : \tau_2} \quad \begin{array}{c}\tau_2 \leq \tau' \\ \Phi \leq \Phi'\end{array}}{T :: \Phi'; \Gamma \vdash e_1\ e_2 : \tau'}$$

and

$$\vdash \langle T, \alpha, \omega, H, e_1\ e_2\rangle$$

we have

$$\Gamma \vdash H$$

From that and T_1 we have

$$\vdash \langle T_1, \alpha, \omega, H, e_1\rangle$$

Therefore, by induction we get

$$\vdash \langle T_1', \alpha_1, \omega_1, H_1, \lambda x.e\rangle$$

where

$$T_1' :: \Phi_\emptyset; \Gamma_1 \vdash \lambda x.e : \tau_1 \to^{\Phi_f} \tau_2$$

which gives

$$\Gamma_1 \vdash H_1$$

and

$$T_f :: \Phi_f'; \Gamma, x : \tau_1' \vdash e : \tau_2'$$

Also, lemma C.6.1 gives $\Gamma_1 \supseteq \Gamma$, therefore

$$T_2' = \textsc{Weaken}\left([T_2], \Gamma_1\right)$$

gives

$$\vdash \langle T_2', \alpha_1, \omega_1, H_1, e_2\rangle$$

By induction we get

$$\vdash \langle T_2'', \alpha_2, \omega_2, H_2, v_2\rangle$$

where

$$T_2'' :: \Phi_\emptyset; \Gamma_2 \vdash v_2 : \tau_1$$

$$\Gamma_2 \vdash H_2$$

and lemma C.6.1 gives $\Gamma_2 \supseteq \Gamma_1$, therefore

$$T'_f = \textsc{Weaken}\left([T_f], \Gamma_2, x : \tau'_1\right)$$

is

$$T'_f :: \Phi'_f; \Gamma_2, x : \tau'_1 \vdash e : \tau'_2$$

From T''_2 and $\tau_1 \leq \tau'_1$ we get

$$T'''_2 :: \Phi_\emptyset; \Gamma_2 \vdash v_2 : \tau'_1$$

Then with

$$T_3 = \textsc{Subst}\left(\left[T'_f\right], [T'''_2]\right)$$

we have

$$\vdash \langle T_3, \alpha_2, \omega_2, H_2, [v_2 \mapsto e]x\rangle$$

and by induction

$$\vdash \langle T_v, \alpha', \omega', H', v\rangle$$

case [REF-A] **:**

Given

$$\textsc{[Ref-A]}\frac{\begin{array}{c}\langle T_1, \alpha, \omega, H, e\rangle \xrightarrow{\varepsilon} \langle T_v, \alpha', \omega', H', v\rangle \\ r \notin \mathit{dom}(H')\end{array}}{\langle T, \alpha, \omega, H, \mathsf{ref}^L\ e\rangle \xrightarrow{\varepsilon} \langle T_r, \alpha', \omega', (H', r \mapsto v), r_L\rangle}$$

and

$$\vdash \langle T, \alpha, \omega, H, \mathsf{ref}^L\ e\rangle$$

Similarly to the above, we get

$$\vdash \langle T_1, \alpha, \omega, H, e\rangle$$

and by induction

$$\vdash \langle T_v, \alpha', \omega', H', v\rangle$$

where $T_v :: \Phi_\emptyset; \Gamma' \vdash v : \tau$. Then for $\Gamma'' = (\Gamma', r \mapsto \tau\ H'' = (H', r \mapsto v)$ we have $\Gamma'' \vdash H''$ Also $\Gamma'' \supseteq \Gamma'$ and from lemma C.6.1 we get $\Gamma' \supseteq \Gamma$, therefore

$$\vdash \langle T_r, \alpha', \omega', H'', r_L\rangle$$

case Other **:**

The remaining cases are similar.

□

Lemma C.6.3 (A typed evaluation derivation exists) *If*

$$T :: \Phi; \Gamma \vdash e : \tau$$

and

$$D :: \langle \alpha, \omega, H, e \rangle \xrightarrow{\varepsilon} \langle \alpha', \omega', H', v \rangle$$

and

$$\vdash \langle \alpha, \omega, H, e \rangle$$

then there exists T_v such that

$$\langle T, \alpha, \omega, H, e \rangle \xrightarrow{\varepsilon} \langle T_v, \alpha', \omega', H', v \rangle$$

Proof:

case [ID] **:**

Given

$$T :: \Phi; \Gamma \vdash v : \tau$$

and

$$D :: \langle \alpha, \omega, H, v \rangle \xrightarrow{\emptyset} \langle \alpha, \omega, H, v \rangle$$

From [ID-A] we get

$$\langle T, \alpha, \omega, H, v \rangle \xrightarrow{\emptyset} \langle T, \alpha, \omega, H, v \rangle$$

case [CALL] **:**

The assumptions are:

$$[\text{TSUB}]\dfrac{[\text{TAPP}]\dfrac{\begin{array}{c} T_1 :: \Phi_1; \Gamma \vdash e_1 : \tau_1 \rightarrow^{\Phi_f} \tau_2 \\ T_2 :: \Phi_2; \Gamma \vdash e_2 : \tau_1 \\ \Phi_1 \rhd \Phi_2 \rhd \Phi_f \hookrightarrow \Phi \end{array}}{\Phi; \Gamma \vdash e_1\ e_2 : \tau_2} \quad \begin{array}{c} \tau_2 \leq \tau' \\ \Phi \leq \Phi' \end{array}}{T :: \Phi'; \Gamma \vdash e_1\ e_2 : \tau'}$$

and

$$[\text{CALL}]\dfrac{\begin{array}{c} D_1 :: \langle \alpha, \omega, H, e_1 \rangle \xrightarrow{\varepsilon_1} \langle \alpha_1, \omega_1, H_1, \lambda x.e \rangle \\ D_2 :: \langle \alpha_1, \omega_1, H_1, e_2 \rangle \xrightarrow{\varepsilon_2} \langle \alpha_2, \omega_2, H_2, v_2 \rangle \\ D_3 :: \langle \alpha_2, \omega_2, H_2, [v_2 \mapsto e]x \rangle \xrightarrow{\varepsilon_3} \langle \alpha', \omega', H', v \rangle \end{array}}{D :: \langle \alpha, \omega, H, e_1\ e_2 \rangle \xrightarrow{\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3} \langle \alpha', \omega', H', v \rangle}$$

and $\Gamma \vdash H$. From T_1, D_1 and $\Gamma \vdash H$, by induction we have: $E_1 :: \langle T_1, \alpha, \omega, H, e_1 \rangle \xrightarrow{\varepsilon_1} \langle T_1', \alpha_1, \omega_1, H_1, \lambda x.e \rangle$ where

$$[\text{TSUB}]\dfrac{[\text{TLAM}]\dfrac{T_f :: \Phi_f'; \Gamma_1, x : \tau_1' \vdash e : \tau_2'}{\Phi_\emptyset; \Gamma_1 \vdash \lambda x.e : \tau_1' \rightarrow^{\Phi_f'} \tau_2'} \quad \begin{array}{c} \tau_1' \rightarrow^{\Phi_f'} \tau_2' \leq \tau_1 \rightarrow^{\Phi_f} \tau_2 \\ \Phi_\emptyset \leq \Phi_\emptyset \end{array}}{T_1' :: \Phi_\emptyset; \Gamma_1 \vdash \lambda x.e : \tau_1 \rightarrow^{\Phi_f} \tau_2}$$

is a canonical typing, $\Gamma_1 \supseteq \Gamma$ and $\Gamma_1 \vdash H_1$.

From T_2 and $\Gamma_1 \supseteq \Gamma$ we get $T_2' = \textsc{Weaken}\left([T_2], \Gamma_1\right)$.

From T_2', D_2, and $\Gamma_1 \vdash H_1$, we get by induction: $E_2 :: \langle T_2', \alpha_1, \omega_1, H_1, e_2\rangle \xrightarrow{\varepsilon_2} \langle T_{v_2}, \alpha_2, \omega_2, H_2, v_2\rangle$ where $T_{v_2} :: \Phi_\emptyset; \Gamma_2 \vdash v_2 : \tau_1$, $\Gamma_2 \supseteq \Gamma_1$ and $\Gamma_2 \vdash H_2$.

From $\tau_1' \rightarrow^{\Phi_f'} \tau_2' \leq \tau_1 \rightarrow^{\Phi_f} \tau_2$ we get $\tau_1 \leq \tau_1'$, $\tau_2' \leq \tau_2$ and $\Phi_f' \leq \Phi_f$, therefore

$$[\textsc{TSub}]\frac{\begin{array}{c} T_{v_2} :: \Phi_\emptyset; \Gamma_2 \vdash v_2 : \tau_1 \\ \tau_1 \leq \tau_1' \\ \Phi_\emptyset \leq \Phi_\emptyset \end{array}}{T_2'' :: \Phi_\emptyset; \Gamma_2 \vdash v_2 : \tau_1'}$$

Also, from $\Gamma_2 \supseteq \Gamma_1$ we get a $T_f' = \textsc{Weaken}\left([T_f], \Gamma_2\right)$. Then we construct $T_3' = \textsc{Subst}\left(\left[T_f'\right], [T_2'']\right)$ such that $T_3' :: \Phi_f'; \Gamma_2 \vdash [v_2 \mapsto e]x : \tau_2'$. Finally, from $\tau_2' \leq \tau_2$ we construct T_3:

$$[\textsc{TSub}]\frac{\begin{array}{c} T_3' :: \Phi_f'; \Gamma_2 \vdash [v_2 \mapsto e]x : \tau_2' \\ \tau_2' \leq \tau_2 \\ \Phi_f' \leq \Phi_f \end{array}}{T_3 :: \Phi_f; \Gamma_2 \vdash [v_2 \mapsto e]x : \tau_2}$$

From the last and induction, we get $\langle T_3, \alpha_2, \omega_2, H_2, [v_2 \mapsto e]x\rangle \xrightarrow{\varepsilon_3} \langle T_v, \alpha', \omega', H', v\rangle$ where $T_v :: \Phi_\emptyset; \Gamma_3 \vdash v : \tau_2$, $\Gamma_3 \supseteq \Gamma_2$ and $\Gamma_3 \vdash H'$.

So, we can apply [CALL-A] to get

$$[\textsc{Call-A}]\frac{\begin{array}{c} \langle T_1, \alpha, \omega, H, e_1\rangle \xrightarrow{\varepsilon_1} \langle T_1', \alpha_1, \omega_1, H_1, \lambda x.e\rangle \\ \langle T_2, \alpha_1, \omega_1, H_1, e_2\rangle \xrightarrow{\varepsilon_2} \langle T_{v_2}, \alpha_2, \omega_2, H_2, v_2\rangle \\ \langle T_3, \alpha_2, \omega_2, H_2, [v_2 \mapsto e]x\rangle \xrightarrow{\varepsilon_3} \langle T_v, \alpha', \omega', H', v\rangle \end{array}}{\langle T, \alpha, \omega, H, e_1\ e_2\rangle \xrightarrow{\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3} \langle T_v, \alpha', \omega', H', v\rangle}$$

case [REF] **:**

Given

$$[\textsc{TSub}]\frac{[\textsc{TRef}]\ \dfrac{T_1 :: \Phi'; \Gamma \vdash e : \tau'}{\Phi; \Gamma \vdash \mathsf{ref}^L\ e : \mathit{ref}^{\{L\}}(\tau')} \quad \begin{array}{c} \Phi' \leq \Phi \\ \mathit{ref}^{\{L\}}(\tau') \leq \mathit{ref}^{\varepsilon'}(\tau) \end{array}}{T :: \Phi; \Gamma \vdash \mathsf{ref}^L\ e : \mathit{ref}^{\varepsilon'}(\tau)}$$

and

$$[\textsc{Ref}]\frac{\begin{array}{c} D_1 :: \langle \alpha, \omega, H, e\rangle \xrightarrow{\varepsilon} \langle \alpha', \omega', H', v\rangle \\ r \notin \mathit{dom}(H') \end{array}}{D :: \langle \alpha, \omega, H, \mathsf{ref}^L\ e\rangle \xrightarrow{\varepsilon} \langle \alpha', \omega', (H', r \mapsto v), r_L\rangle}$$

and $\Gamma \vdash H$

By induction, T_1 and D_1 we have

$$\langle T_1, \alpha, \omega, H, e\rangle \xrightarrow{\varepsilon} \langle T_v, \alpha', \omega', H', v\rangle$$

where $T_v :: \Phi_\emptyset; \Gamma' \vdash e : \tau'$ is a canonical typing, $\Gamma' \supseteq \Gamma$ and $\Gamma' \vdash H'$

We extend Γ' and H' to define $\Gamma'' = (\Gamma', r \mapsto \tau')$ and $H'' = (H', r \mapsto v)$ respectively. Then clearly $\Gamma'' \supseteq \Gamma'$ and $\Gamma'' \vdash H''$.

Therefore we can apply [REF-A], $\langle T, \alpha, \omega, H, e\rangle \xrightarrow{\varepsilon} \langle T_r, \alpha', \omega', (H', r \mapsto v), r_L\rangle$ where

$$\text{[TSUB]}\frac{\text{[TLOC]}\ \dfrac{\Gamma''(r) = \tau'}{\Phi_\emptyset; \Gamma'' \vdash r_L : \mathit{ref}^{\{L\}}(\tau')} \quad \mathit{ref}^{\{L\}}(\tau') \leq \mathit{ref}^{\varepsilon'}(\tau) \quad \Phi_\emptyset \leq \Phi_\emptyset}{T_r :: \Phi_\emptyset; \Gamma'' \vdash r_L : \mathit{ref}^{\{\varepsilon\}}(\tau)}$$

$\Gamma' \supseteq \Gamma$ and $\Gamma'' \vdash H''$

case [DEREF] **:**

Given

$$\text{[TSUB]}\frac{\text{[TDEREF]}\ \dfrac{\Phi_1; \Gamma \vdash e : \mathit{ref}^{\varepsilon_1}(\tau') \quad \Phi_2^\varepsilon = \varepsilon_1 \quad \Phi_1 \rhd \Phi_2 \hookrightarrow \Phi'}{\Phi'; \Gamma \vdash !\, e : \tau'} \quad \tau' \leq \tau \quad \Phi' \leq \Phi}{\Phi; \Gamma \vdash !\, e : \tau}$$

and

$$\text{[DEREF]}\frac{D_1 :: \langle \alpha, \omega, H, e\rangle \xrightarrow{\varepsilon} \langle \alpha', \omega' \cup \{L\}, H', r_L\rangle \qquad r \in \mathit{dom}(H')}{D :: \langle \alpha, \omega, H, !\, e\rangle \xrightarrow{\varepsilon \cup \{L\}} \langle \alpha' \cup \{L\}, \omega', H', H'(r)\rangle}$$

and $\Gamma \vdash H$.

By induction, T_1 and D_1 we have

$$\langle T_1, \alpha, \omega, H, e\rangle \xrightarrow{\varepsilon} \langle T_r, \alpha', \omega' \cup \{L\}, H', r_L\rangle$$

where

$$\text{[TSUB]}\frac{\text{[TLOC]}\ \dfrac{\Gamma'(r) = \tau'}{\Phi_\emptyset; \Gamma' \vdash r_L : \mathit{ref}^{\varepsilon'}(\tau')} \quad \Phi_\emptyset \leq \Phi_\emptyset \quad \mathit{ref}^{\varepsilon'}(\tau') \leq \mathit{ref}^{\varepsilon}(\tau)}{T_r :: \Phi_\emptyset; \Gamma' \vdash r_L : \mathit{ref}^{\varepsilon}(\tau)}$$

and $\Gamma' \vdash H'$.

From $\Gamma' \vdash H'$ we have $r \in \mathit{dom}(H') = \mathit{dom}(\Gamma')$ and

$$T_v :: \Phi_\emptyset; \Gamma' \vdash H'(r) : \Gamma'(r)$$

Now we can apply [DEREF-A]:

$$\text{[DEREF-A]}\frac{\begin{array}{c}\langle T_1, \alpha, \omega, H, e\rangle \xrightarrow{\varepsilon} \langle T_r, \alpha', \omega' \cup \{L\}, H', r_L\rangle \\ r \in \mathit{dom}(H')\end{array}}{\langle T, \alpha, \omega, H, !\, e\rangle \xrightarrow{\varepsilon} \langle T_v, \alpha', \omega', H', v\rangle}$$

case [ASSIGN] **:**

Similar to [DEREF].

case [IF-T] **:**

Given

$$\text{[IF-T]}\frac{\begin{array}{c}D_1 :: \langle \alpha, \omega, H, e_1\rangle \xrightarrow{\varepsilon_1} \langle \alpha_1, \omega_1, H_1, v_1\rangle \qquad v_1 = 0 \\ D_2 :: \langle \alpha_1, \omega_1, H_1, e_2\rangle \xrightarrow{\varepsilon_2} \langle \alpha_2, \omega_2, H_2, v\rangle\end{array}}{D :: \langle \alpha, \omega, H, \mathsf{if0}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3\rangle \xrightarrow{\varepsilon_1 \cup \varepsilon_2} \langle \alpha_2, \omega_2, H_2, v\rangle}$$

and

$$\text{[TIF]}\frac{\begin{array}{c}T_1 :: \Phi_1; \Gamma \vdash e_1 : \mathit{int} \\ T_2 :: \Phi_2; \Gamma \vdash e_2 : \tau \\ T_3 :: \Phi_2; \Gamma \vdash e_3 : \tau \\ \Phi_1 \rhd \Phi_2 \hookrightarrow \Phi\end{array}}{T :: \Phi; \Gamma \vdash \mathsf{if0}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau}$$

and $\Gamma \vdash H$.

By induction, D_1 and T_1 we have

$$\langle T_1, \alpha, \omega, H, e_1\rangle \xrightarrow{\varepsilon_1} \langle T_{v_1}, \alpha_1, \omega_1, H_1, v_1\rangle$$

where

$$\text{[TSUB]}\frac{\begin{array}{c}\text{[TINT]}\ \dfrac{}{\Phi_\emptyset; \Gamma_1 \vdash v_1 : \mathit{int}} \\ \Phi_\emptyset \leq \Phi_\emptyset \\ \mathit{int} \leq \mathit{int}\end{array}}{T_{v_1} :: \Phi_\emptyset; \Gamma_1 \vdash v_1 : \mathit{int}}$$

and $\Gamma_1 \supseteq \Gamma$ and $\Gamma_1 \vdash H_1$.

We have $\Gamma_1 \supseteq \Gamma$, so we get $T_2' = \textsc{Weaken}\left([T_2], \Gamma_1\right) :: \Phi_2; \Gamma_1 \vdash e_2 : \tau$.

By induction, T_2' and D_2 we get

$$\langle T_2', \alpha_1, \omega_1, H, e_1\rangle \xrightarrow{\varepsilon_1} \langle T_v, \alpha_2, \omega_2, H_2, v\rangle$$

where

$$T_v :: \Phi_\emptyset; \Gamma_2 \vdash v : \tau$$

and $\Gamma_2 \supseteq \Gamma_1$ and $\Gamma_2 \vdash H_2$.

Then we can apply [IF-T-A] to get

$$\text{[IF-T-A]}\frac{\begin{array}{c}\langle T_1, \alpha, \omega, H, e_1\rangle \xrightarrow{\varepsilon_1} \langle T_{v_1}, \alpha_1, \omega_1, H_1, v_1\rangle \\ v_1 = 0 \\ \langle T_2', \alpha_1, \omega_1, H, e_1\rangle \xrightarrow{\varepsilon_1} \langle T_v, \alpha_2, \omega_2, H_2, v\rangle\end{array}}{\langle T, \alpha, \omega, H, \mathsf{if0}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3\rangle \xrightarrow{\varepsilon_1 \cup \varepsilon_2} \langle T_v, \alpha_2, \omega_2, H_2, v\rangle}$$

case [IF-F] **:**

Similar to [IF-T].

case [LET] **:**

Similar to [CALL].

□

Lemma C.6.4 (The typed evaluation derivation is complete w.r.t. the evaluation) *If*

$$E :: \langle T, \alpha, \omega, H, e\rangle \xrightarrow{\varepsilon} \langle T_v, \alpha', \omega', H', v\rangle$$

Then

$$\langle \alpha, \omega, H, e\rangle \xrightarrow{\varepsilon} \langle \alpha', \omega', H', v\rangle$$

Proof: Trivial. Sketch: all typed evaluation rules have a corresponding untyped evaluation rule, we can convert each typed evaluation rule to its corresponding untyped by just removing the annotation T. □

Theorem C.6.5 (Prior and Future Effect Soundness) *If*

$$E :: \langle T, \alpha, \omega, H, e\rangle \xrightarrow{\varepsilon} \langle T_v, \alpha', \omega', H', v\rangle$$

where

$$T :: \Phi; \Gamma \vdash e : \tau$$

and

$$\alpha \subseteq \Phi^{\alpha}$$

and

$$\omega' \subseteq \Phi^{\omega}$$

then for all sub-derivations E_i of E,

$$E_i :: \langle T_i, \alpha_i, \omega_i, H_i, e_i\rangle \xrightarrow{\varepsilon} \langle T_{v_i}, \alpha_i', \omega_i', H_i', v_i\rangle$$

where

$$T_i :: \Phi_i; \Gamma_i \vdash e_i : \tau_i$$

it will hold that

$$\alpha_i \subseteq \Phi_i^{\alpha}$$

and

$$\omega_i' \subseteq \Phi_i^{\omega}$$

Proof:

case [ID-A] **:**

Given

$$\text{[ID-A]}\frac{}{E :: \langle T, \alpha, \omega, H, v\rangle \xrightarrow{\emptyset} \langle T, \alpha, \omega, H, v\rangle}$$
$$T :: \Phi; \Gamma \vdash e : \tau$$
$$\alpha \subseteq \Phi^{\alpha}$$
$$\omega' \subseteq \Phi^{\omega}$$

There are no sub-derivations, therefore the lemma holds vacuously.

case [CALL-A] **:**

Given

$$\text{[CALL-A]}\frac{\begin{array}{c} E_1 :: \langle T_1, \alpha, \omega, H, e_1\rangle \xrightarrow{\varepsilon_1} \langle T_1', \alpha_1, \omega_1, H_1, \lambda x.e\rangle \\ E_2 :: \langle T_2', \alpha_1, \omega_1, H_1, e_2\rangle \xrightarrow{\varepsilon_2} \langle T_2'', \alpha_2, \omega_2, H_2, v_2\rangle \\ E_3 :: \langle T_3, \alpha_2, \omega_2, H_2, [v_2 \mapsto e]x\rangle \xrightarrow{\varepsilon_3} \langle T_v, \alpha', \omega', H', v\rangle \end{array}}{E :: \langle T, \alpha, \omega, H, e_1\ e_2\rangle \xrightarrow{\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3} \langle T_v, \alpha', \omega', H', v\rangle}$$

and

$$T :: \Phi; \Gamma \vdash e_1\ e_2 : \tau'$$
$$\alpha \subseteq \Phi^{\alpha}$$
$$\omega' \subseteq \Phi^{\omega}$$

From [CALL-A] we have:

$$T_1 :: \Phi_1; \Gamma \vdash e_1 : \tau_1 \rightarrow^{\Phi_f} \tau_2$$
$$T_2 :: \Phi_2; \Gamma \vdash e_2 : \tau_1$$
$$T_2' = \textsc{Weaken}\left([T_2], \Gamma_1\right)$$
$$T_3 :: \Phi_f; \Gamma_2 \vdash [v_2 \mapsto e]x : \tau'$$

Using Destruction of the typed evaluation (Lemma C.6.4), we get the corresponding untyped evaluation to E. We can then use Weakening of evaluations (Lemma C.3.1) to relax α and ω to 1, we can apply Standard Effect soundness (Theorem C.2.1) to get

$$\varepsilon_1 \subseteq \Phi_1^{\varepsilon}$$
$$\varepsilon_2 \subseteq \Phi_2^{\varepsilon}$$
$$\varepsilon_3 \subseteq \Phi_f^{\varepsilon}$$

From

$$\Phi_1 \rhd \Phi_2 \rhd \Phi_f \hookrightarrow \Phi$$

we get

$$\Phi_1^\alpha = \Phi^\alpha$$
$$\Phi_2^\alpha = \Phi^\alpha \cup \Phi_1^\varepsilon$$
$$\Phi_f^\alpha = \Phi^\alpha \cup \Phi_1^\varepsilon \cup \Phi_2^\varepsilon$$

and

$$\Phi_1^\omega = \Phi^\omega \cup \Phi_2^\varepsilon \cup \Phi_f^\varepsilon$$
$$\Phi_2^\omega = \Phi^\omega \cup \Phi_f^\varepsilon$$
$$\Phi_f^\omega = \Phi^\omega$$

From Traces and Promises (Lemma C.3.3) we get

$$\alpha_1 = \alpha \cup \varepsilon_1$$
$$\alpha_2 = \alpha \cup \varepsilon_1 \cup \varepsilon_2$$

and

$$\omega_1 = \omega \cup \varepsilon_2 \cup \varepsilon_3$$
$$\omega_2 = \omega \cup \varepsilon_3$$

Therefore, for E_1

$$\alpha \subseteq \Phi^\alpha = \Phi_1^\alpha$$
$$\omega_1 = \omega \cup \varepsilon_2 \cup \varepsilon_3 \subseteq \Phi^\omega \cup \Phi_2^\varepsilon \cup \Phi_f^\varepsilon = \Phi_1^\omega$$

We can now apply the lemma inductively on E_1 to get that then for all sub-derivations E_i of E_1,

$$E_i :: \langle T_i, \alpha_i, \omega_i, H_i, e_i \rangle \xrightarrow{\varepsilon} \langle T_{v_i}, \alpha_i', \omega_i', H_i', v_i \rangle$$

where

$$T_i :: \Phi_i; \Gamma_i \vdash e_i : \tau_i$$

it will hold that

$$\alpha_i \subseteq \Phi_i^\alpha$$

and

$$\omega_i' \subseteq \Phi_i^\omega$$

For E_2

$$\alpha_1 = \alpha \cup \varepsilon_1 \subseteq \Phi^\alpha \cup \Phi_1^\varepsilon = \Phi_2^\alpha$$
$$\omega_2 = \omega \cup \varepsilon_3 \subseteq \Phi^\omega \cup \Phi_f^\varepsilon = \Phi_2^\omega$$

Similarly to E_1, we can now apply induction to get the wanted property for all sub-derivations of E_2.

For E_3

$$\alpha_2 = \alpha \cup \varepsilon_1 \cup \varepsilon_2 \subseteq \Phi^\alpha \cup \Phi_1^\varepsilon \cup \Phi_2^\varepsilon = \Phi_f^\alpha$$
$$\omega' = \omega \subseteq \Phi^\omega = \Phi_f^\omega$$

As before, we can now apply induction to get the wanted property for all sub-derivations of E_3.

case [REF-A] **:**

From [REF-A] we have

$$\text{[REF-A]}\frac{\begin{array}{c} E_1 :: \langle T_1, \alpha, \omega, H, e\rangle \xrightarrow{\varepsilon} \langle T_v, \alpha', \omega', H', v\rangle \\ r \notin \mathit{dom}(H') \end{array}}{E :: \langle T, \alpha, \omega, H, \mathsf{ref}^L\ e\rangle \xrightarrow{\varepsilon} \langle T_r, \alpha', \omega', (H', r \mapsto v), r_L\rangle}$$

$$T :: \Phi; \Gamma \vdash \mathsf{ref}^L\ e : \mathit{ref}^{\varepsilon}(\tau)$$

$$\alpha \subseteq \Phi^{\alpha}$$

$$\omega' \subseteq \Phi^{\omega}$$

From the premises of [REF-A]

$$T_1 :: \Phi'; \Gamma \vdash e : \tau'$$

$$\Phi' \leq \Phi$$

Clearly, for E_1 we have from the last

$$\alpha \subseteq \Phi^{\alpha} \subseteq \Phi'^{\alpha}$$

$$\omega' \subseteq \Phi^{\omega} \subseteq \Phi'^{\omega}$$

Similarly to the previous case, we get the wanted property for all sub-derivations by induction.

case [DEREF-A] **:**

$$\text{[DEREF-A]}\frac{\begin{array}{c} E_1 :: \langle T_1, \alpha, \omega, H, e\rangle \xrightarrow{\varepsilon} \langle T_r, \alpha', \omega' \cup \{L\}, H', r_L\rangle \\ r \in \mathit{dom}(H') \end{array}}{E :: \langle T, \alpha, \omega, H, !\, e\rangle \xrightarrow{\varepsilon} \langle T_v, \alpha', \omega', H', v\rangle}$$

$$T :: \Phi; \Gamma \vdash !\, e : \tau$$

$$\alpha \subseteq \Phi^{\alpha}$$

$$\omega' \subseteq \Phi^{\omega}$$

From the premises

$$\Phi_2^{\varepsilon} = \varepsilon$$

$$\Phi_1 \rhd \Phi_2 \hookrightarrow \Phi'$$

$$\Phi' \leq \Phi$$

$$\mathit{ref}^{\{L\}}(\tau) \leq \mathit{ref}^{\varepsilon}(\tau')$$

Therefore

$$\Phi_1^{\alpha} = \Phi'^{\alpha}$$

$$\Phi_1^\omega = \Phi'^\omega \cup \Phi_2^\varepsilon$$

For E_1 we have

$$\alpha \subseteq \Phi^\alpha \subseteq \Phi'^\alpha = \Phi_1^\alpha$$
$$\omega' \cup \{L\} \subseteq \Phi^\omega \cup \varepsilon \subseteq \Phi'^\omega \cup \Phi_2^\varepsilon = \Phi_1^\omega$$

As before, we get the wanted property for all sub-derivations by induction.

case Other **:**

The other cases are similar.

□

Theorem C.6.6 (Contextual Effect Soundness) *Given a program e_p with no free variables, its typing $\mathcal{T}$ and its canonical evaluation $\mathcal{D}$, we can construct a typed evaluation $\mathcal{E}$ such that for every sub-derivation*

$$E :: \langle T, \alpha, \omega, H, e\rangle \xrightarrow{\varepsilon} \langle T_v, \alpha', \omega', H', v\rangle$$

in $\mathcal{E}$, where $T :: \Phi; \Gamma \vdash e : \tau$, it is always the case that $\alpha \subseteq \Phi^\alpha$ $\varepsilon \subseteq \Phi^\varepsilon$ and $\omega \subseteq \Phi^{\omega'}$.

Proof: Sketch: Follows as a corollary from Lemmas C.6.5 and C.6.3, with initial $\Gamma_p = \emptyset$ and $H_p = \emptyset$. Since $\mathcal{D}$ is canonical, it is $\alpha_p = \omega'_p = \emptyset$ for the whole program (base case) and by induction we get the theorem for all sub-derivations. □

Bibliography

[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 '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 233–242, New York, NY, USA, 2005. ACM Press.

[2] Rahul Agarwal and Scott D. Stoller. Type inference for parameterized race-free java. In *Proceedings of the Fifth International Conference on Verification, Model Checking and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 149–160. Springer-Verlag, January 2004.

[3] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design (Addison-Wesley series in computer science and information processing)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1977.

[4] Alex Aiken, Jeffrey S. Foster, John Kodumal, and Tachio Terauchi. Checking and inferring local non-aliasing. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 129–140, New York, NY, USA, 2003. ACM Press.

[5] Alexander Aiken. Introduction to set constraint-based program analysis. *Sci. Comput. Program.*, 35(2-3):79–111, 1999.

[6] Andrei Alexandrescu, Hans Boehm, Kevlin Henney, Ben Hutchings, Doug Lea, and Bill Pugh. Memory model for multithreaded c++: Issues, March 2005.

[7] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[8] Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 3–15, New York, NY, USA, 2008. ACM.

[9] Utpal K. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[10] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for c#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, 2004.

[11] Gregory Biegel, Vinny Cahill, and Mads Haahr. A dynamic proxy based architecture to support distributed java objects in a mobile environment. In *On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002*, pages 809–826, London, UK, 2002. Springer-Verlag.

[12] Gavin Bierman, Matthew Parkinson, and Andrew Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report 563, Cambridge University Computer Laboratory, April 2003.

[13] Didier Le Botlan and Didier Rémy. MI^{f}: raising ml to the power of system f. In Runciman and Shivers [141], pages 27–38.

[14] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 211–230, New York, NY, USA, 2002. ACM Press.

[15] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free java programs. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 56–69, New York, NY, USA, 2001. ACM Press.

[16] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: adding genericity to the java programming language. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 183–200, New York, NY, USA, 1998. ACM Press.

[17] Brendan Burns, Kevin Grimaldi, Alexander Kostadinov, Emery D. Berger, and Mark D. Corner. Flux: A language for programming high-performance servers. In *USENIX Annual Technical Conference, General Track* [158], pages 129–142.

[18] Cristiano Calcagno. Stratified operational semantics for safety and correctness of the region calculus. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 155–165, New York, NY, USA, 2001. ACM.

[19] Satish Chandra and Thomas Reps. Physical type checking for c. In *PASTE '99: Proceedings of the 1999 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 66–75, New York, NY, USA, 1999. ACM Press.

[20] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented program-*

ming, systems, languages, and applications, pages 519–538, New York, NY, USA, 2005. ACM Press.

[21] 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 '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 258–269, New York, NY, USA, 2002. ACM Press.

[22] Ravi Chugh, Jan W. Voung, Ranjit Jhala, and Sorin Lerner. Dataflow analysis for concurrent programs using datarace detection. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 316–326, New York, NY, USA, 2008. ACM.

[23] James Compton. Scoop: An investigation of concurrency in eiffel. Master's thesis, Department of Computer Science, The Australian National University, 2000.

[24] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, , and George Necula. Dependent types for low-level programming. In *Proceedings of the 16th European Symposium on Programming*, March 2007.

[25] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. Iterative data-flow analysis, revisited. Technical Report TR04-100, Department of Computer Science, Rice University, 2004.

[26] The Coq proof assistant. `http://coq.inria.fr`.

[27] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 262–275, New York, NY, USA, 1999. ACM Press.

[28] Manuvir Das. Unification-based pointer analysis with directional assignments. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 35–46, New York, NY, USA, 2000. ACM Press.

[29] Manuvir Das, Ben Liblit, Manuel Fähndrich, and Jakob Rehof. Estimating the impact of scalable pointer analysis on optimization. In *SAS '01: Proceedings of the 8th International Symposium on Static Analysis*, pages 260–278, London, UK, 2001. Springer-Verlag.

[30] Xianghua Deng, Matthew B. Dwyer, John Hatcliff, and Masaaki Mizuno. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 442–452, New York, NY, USA, 2002. ACM Press.

[31] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 242–256, New York, NY, USA, 1994. ACM Press.

[32] Michael Emmi, Jeffrey S. Fischer, Ranjit Jhala, and Rupak Majumdar. Lock allocation. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 291–296, New York, NY, USA, 2007. ACM Press.

[33] Dawson Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 237–252, New York, NY, USA, 2003. ACM Press.

[34] Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: spatial and temporal selection of reification. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 27–46, New York, NY, USA, 2003. ACM Press.

[35] Manuel Fähndrich and Alex Aiken. Making set-constraint program analyses scale. Technical report, University of California at Berkeley, Berkeley, CA, USA, 1996.

[36] Manuel Fähndrich and Robert DeLine. Adoption and focus: practical linear types for imperative programming. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 13–24, New York, NY, USA, 2002. ACM Press.

[37] Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 302–312, New York, NY, USA, 2003. ACM Press.

[38] 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.

[39] Manuel Fähndrich, Jakob Rehof, and Manuvir Das. From Polymorphic Subtyping to CFL Reachability: Context-Sensitive Flow Analysis Using Instantiation Constraints. Technical Report MS-TR-99-84, Microsoft Research, March 2000.

[40] Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable context-sensitive flow analysis using instantiation constraints. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 253–263, New York, NY, USA, 2000. ACM Press.

[41] Manuel Alfred Fähndrich. *Bane: a library for scalable constraint-based program analysis*. PhD thesis, University of California, Berkeley, 1999. Chair-Alexander Aiken.

[42] Cormac Flanagan and Martín Abadi. Types for safe locking. In *ESOP '99: Proceedings of the 8th European Symposium on Programming Languages and Systems*, pages 91–108, London, UK, 1999. Springer-Verlag.

[43] Cormac Flanagan and Matthias Felleisen. The semantics of future and its use in program optimization. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 209–220, New York, NY, USA, 1995. ACM Press.

[44] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 235–248, New York, NY, USA, 1997. ACM.

[45] Cormac Flanagan and Stephen N. Freund. Type-based race detection for java. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 219–232, New York, NY, USA, 2000. ACM Press.

[46] Cormac Flanagan and Stephen N. Freund. Detecting race conditions in large programs. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 90–96, New York, NY, USA, 2001. ACM Press.

[47] Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 256–267, New York, NY, USA, 2004. ACM Press.

[48] Cormac Flanagan and Stephen N. Freund. Type inference against races. In Giacobazzi [62], pages 116–132.

[49] Cormac Flanagan and Stephen N. Freund. Automatic synchronization correction. In *Synchronization and Concurrency in Object- Oriented Languages (SCOOL)*, October 2005.

[50] Cormac Flanagan and Stephen N. Freund. Type inference against races. *Sci. Comput. Program.*, 64(1):140–165, 2007.

[51] Cormac Flanagan, Stephen N. Freund, and Marina Lifshin. Type inference for atomicity. In *TLDI '05: Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 47–58, New York, NY, USA, 2005. ACM Press.

[52] Cormac Flanagan, Stephen N. Freund, and Shaz Qadeer. Exploiting purity for atomicity. *SIGSOFT Softw. Eng. Notes*, 29(4):221–231, 2004.

[53] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM Press.

[54] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 338–349, New York, NY, USA, 2003. ACM Press.

[55] International Technology Roadmap for Semiconductors. 2006 update: Overview and working group summaries. `http://www.itrs.net/Links/2006Update/2006UpdateFinal.htm`, 2006.

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

[57] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 192–203, New York, NY, USA, 1999. ACM Press.

[58] Jeffrey S. Foster, Robert Johnson, John Kodumal, and Alex Aiken. Flow-insensitive type qualifiers. *ACM Trans. Program. Lang. Syst.*, 28(6):1035–1087, 2006.

[59] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 2002. ACM Press.

[60] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, New York, NY, USA, 1998. ACM Press.

[61] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[62] Roberto Giacobazzi, editor. *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings*, volume 3148 of *Lecture Notes in Computer Science*. Springer, 2004.

[63] 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.

[64] Dan Grossman. Type-safe multithreading in cyclone. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 13–25, New York, NY, USA, 2003. ACM Press.

[65] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 282–293, New York, NY, USA, 2002. ACM Press.

[66] Chris Hankin and Igor Siveroni, editors. *Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings*, volume 3672 of *Lecture Notes in Computer Science*. Springer, 2005.

[67] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402, New York, NY, USA, 2003. ACM Press.

[68] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM Press.

[69] Görel Hedin, editor. *Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, volume 2622 of *Lecture Notes in Computer Science*. Springer, 2003.

[70] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using cla: a million lines of c code in a second. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 254–263, New York, NY, USA, 2001. ACM.

[71] Fritz Henglein. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):253–289, 1993.

[72] Fritz Henglein, Henning Makholm, and Henning Niss. A direct approach to control-flow sensitive region-based memory management. In *PPDP '01: Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 175–186, New York, NY, USA, 2001. ACM Press.

[73] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. *SIGPLAN Not.*, 39(6):1–13, 2004.

[74] Maurice Herlihy, Victor Luchangco, Mark Moir, and III William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, 2003. ACM Press.

[75] Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Lock inference for atomic sections. In *Proceedings of the First ACM SIGPLAN Workshop on Languages Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, June 2006.

[76] Chris Hote. Run-time error detection through semantic analysis, 2004.

[77] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[78] Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. *ACM Trans. Program. Lang. Syst.*, 27(2):264–313, 2005.

[79] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[80] intel.com. Teraflops research chip, 2007.

[81] Io: a small programming language. `http://www.iolanguage.com`.

[82] Rob Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 9–9, Berkeley, CA, USA, 2004. USENIX Association.

[83] Dynamic proxy classes. `http://java.sun.com/j2se/1.5.0/docs/guide/reflection/proxy.html`.

[84] Executor examples. `http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/etc/notes/tim-executor-examples.html?rev=1.5`.

[85] JSR 166: Concurrency utilities. `http://www.jcp.org/en/jsr/detail?id=166`.

[86] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[87] John Kodumal and Alex Aiken. The set constraint/cfl reachability connection in practice. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 207–218, New York, NY, USA, 2004. ACM Press.

[88] John Kodumal and Alexander Aiken. Banshee: A scalable constraint-based analysis toolkit. In Hankin and Siveroni [66], pages 218–234.

[89] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[90] Konstantin Läufer and Martin Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16(5):1411–1430, September 1994.

[91] R. Greg Lavender and Douglas C. Schmidt. Active object: an object behavioral pattern for concurrent programming. *Pattern languages of program design 2*, pages 483–499, 1996.

[92] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[93] Nancy G. Leveson and Clark S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[94] Ondrej Lhoták and Laurie J. Hendren. Scaling java points-to analysis using spark. In Hedin [69], pages 153–169.

[95] B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 260–267, New York, NY, USA, 1988. ACM Press.

[96] Kamal Lodaya and Meena Mahajan, editors. *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science, 24th International Conference, Chennai, India, December 16-18, 2004, Proceedings*, volume 3328 of *Lecture Notes in Computer Science*. Springer, 2004.

[97] Cristina Videira Lopes and Karl J. Lieberherr. Abstracting process-to-function relations in concurrency object-oriented applications. In *ECOOP '94: Proceedings of the 8th European Conference on Object-Oriented Programming*, pages 81–99, London, UK, 1994. Springer-Verlag.

[98] David B. Loveman. High performance fortran. *IEEE Parallel Distrib. Technol.*, 1(1):25–42, 1993.

[99] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–57, New York, NY, USA, 1988. ACM Press.

[100] John M. Lucassen. *Types and Effects: Towards the Integration of Functional and Imperative Programming*. PhD thesis, MIT Laboratory for Computer Science, August 1987. MIT/LCS/TR-408.

[101] Kin-Keung Ma and Jeffrey S. Foster. Inferring aliasing and encapsulation properties for java. In *OOPSLA '07: Proceedings of the 22nd ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, October 2007. To appear.

[102] Dragos A. Manolescu. Workflow enactment with continuation and future objects. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 40–51, New York, NY, USA, 2002. ACM Press.

[103] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 378–391, New York, NY, USA, 2005. ACM Press.

[104] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: synchronization inference for atomic sections. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 346–358, New York, NY, USA, 2006. ACM Press.

[105] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 271–283, New York, NY, USA, 1996. ACM Press.

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

[107] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Trans. Program. Lang. Syst.*, 10(3):470–502, 1988.

[108] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 66–77, New York, NY, USA, 1995. ACM Press.

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

[110] Andrew C. Myers. Jflow: practical mostly-static information flow control. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, New York, NY, USA, 1999. ACM Press.

[111] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 308–319, New York, NY, USA, 2006. ACM Press.

[112] Iulian Neamtiu, Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Contextual Effects for Version-Consistent Dynamic Software Updating and Safe Concurrent Programming. Technical Report CS-TR-4875, Dept. of Computer Science, University of Maryland, July 2007.

[113] Iulian Neamtiu, Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, pages 37–50, New York, NY, USA, January 2008. ACM.

[114] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, London, UK, 2002. Springer-Verlag.

[115] news.com. Designer puts 96 cores on single chip, 2007.

[116] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[117] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 167–178, New York, NY, USA, 2003. ACM Press.

[118] Martin Odersky, editor. *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings*, volume 3086 of *Lecture Notes in Computer Science*. Springer, 2004.

[119] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theor. Pract. Object Syst.*, 5(1):35–55, 1999.

[120] Jens Palsberg. Type-based analysis and applications. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 20–27, New York, NY, USA, 2001. ACM Press.

[121] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[122] Kevin Poulsen. Tracking the blackout bug, 2004.

[123] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Context-sensitive correlation analysis for detecting races (extended version). Technical Report CS-TR-4789, Department of Computer Science, University of Maryland, June 2006. Extends PLDI 2006 paper with full formal development.

[124] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Existential label flow inference via CFL reachability. In *Proceedings of the Static Analysis Symposium (SAS)*, Seoul, Korea, August 2006.

[125] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Locksmith: context-sensitive correlation analysis for race detection. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 320–331, New York, NY, USA, 2006. ACM Press.

[126] Polyvios Pratikakis, Jeffrey S. Foster, Michael Hicks, and Iulian Neamtiu. Formalizing soundness of contextual effects. In *Proceedings of the International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, August 2008. To appear.

[127] Polyvios Pratikakis, Michael Hicks, and Jeffrey S. Foster. Existential label flow inference via CFL reachability (extended version). Technical Report CS-TR-4700, Department of Computer Science, University of Maryland, July 2005.

[128] Polyvios Pratikakis, Jaime Spacco, and Michael Hicks. Transparent proxies for java futures. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 206–223, New York, NY, USA, 2004. ACM Press.

[129] Polyvios Pratikakis, Jaime Spacco, and Michael Hicks. Transparent proxies for Java futures (extended version). Technical Report CS-TR-4574, Department of Computer Science, University of Maryland, October 2004.

[130] Shaz Qadeer and Dinghao Wu. Kiss: keep it simple and sequential. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 14–24, New York, NY, USA, 2004. ACM Press.

[131] R. Raje, H. William, and M. Boyles. An asynchronous remote method invocation (armi) mechanism for java, 1997.

[132] Jakob Rehof and Manuel Fähndrich. Type-base flow analysis: from polymorphic subtyping to cfl-reachability. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 54–66, New York, NY, USA, 2001. ACM Press.

[133] Jakob Rehof and Torben Ægidius Mogensen. Tractable constraints in finite semilattices. *Sci. Comput. Program.*, 35(2-3):191–221, 1999.

[134] Didier Rémy. Programming objects with MLART: An extension to ML with abstract and record types. In *Proceedings of the International Symposium on Theoretical Aspects of Computer Science*, pages 321–346, Sendai, Japan, April 1994.

[135] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95: Proceedings of the 22nd ACM*

SIGPLAN-SIGACT symposium on Principles of programming languages, pages 49–61, New York, NY, USA, 1995. ACM Press.

[136] John C. Reynolds. Toward a grainless semantics for shared-variable concurrency. In Lodaya and Mahajan [96], pages 35–48.

[137] John C. Reynolds. Towards a Grainless Semantics for Shared Variable Concurrency. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA, 2004. ACM Press.

[138] Michael F. Ringenburg and Dan Grossman. Atomcaml: first-class atomicity via rollback. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 92–104, New York, NY, USA, 2005. ACM Press.

[139] Jr. Robert H. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

[140] James Rose, Nikhil Swamy, and Michael Hicks. Dynamic inference of polymorphic lock types. *Science of Computer Programming (SCP)*, 58(3), December 2005. Special Issue on Concurrency and Synchronization in Java programs. Supercedes 2004 CSJP paper of the same name.

[141] Colin Runciman and Olin Shivers, editors. *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*. ACM, 2003.

[142] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[143] Helmut Seidl, Varmo Vene, and Markus Müller-Olm. Global invariants for analyzing multi-threaded applications, 2003.

[144] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qaualifiers. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 16–16, Berkeley, CA, USA, 2001. USENIX Association.

[145] Michael Siff, Satish Chandra, Thomas Ball, Krishna Kunchithapadam, and Thomas Reps. Coping with type casts in c. In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 180–198, London, UK, 1999. Springer-Verlag.

[146] Vincent Simonet. An extension of hm(x) with bounded existential and universal data-types. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 39–50, New York, NY, USA, 2003. ACM Press.

[147] Christian Skalka, Scott Smith, and David Van horn. Types and trace effects of higher order programs. *J. Funct. Program.*, 18(2):179–249, 2008.

[148] A. Slominski, M. Govindaraju, D. Gannon, and R. Bramley. Design of an XML based Interoperable RMI System : SoapRMI C++/Java 1.1. In *Proceedings of PDPTA*, pages 1661–1667, June 25-28, 2001.

[149] Frederick Smith, David Walker, and J. Gregory Morrisett. Alias types. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 366–381, London, UK, 2000. Springer-Verlag.

[150] Frederick Smith, David Walker, and J. Gregory Morrisett. Alias types. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 366–381, London, UK, 2000. Springer-Verlag.

[151] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[152] Tim Sweeney. The next mainstream programming language: a game developer's perspective. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 269–269, New York, NY, USA, 2006. ACM Press.

[153] Tomás Sysala and Jan Janecek. Optimizing remote method invocation in java. In *DEXA '02: Proceedings of the 13th International Workshop on Database and Expert Systems Applications*, pages 29–36, Washington, DC, USA, 2002. IEEE Computer Society.

[154] Jean-Pierre Talpin and P‘ierre Jouvelot. The type and effect discipline. *Inf. Comput.*, 111(2):245–296, 1994.

[155] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Inf. Comput.*, 111(2):245–296, 1994.

[156] Tachio Terauchi. Checking race freedom via linear programming. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 1–10, New York, NY, USA, 2008. ACM.

[157] Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Trans. Program. Lang. Syst.*, 20(4):724–767, 1998.

[158] USENIX. *Proceedings of the 2006 USENIX Annual Technical Conference, May 30 - June 3, 2006, Boston, MA, USA*. USENIX, 2006.

[159] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, pages 125–135. IBM Press, 1999.

[160] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: scalable threads for internet services. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 268–281, New York, NY, USA, 2003. ACM Press.

[161] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. Relay: static race detection on millions of lines of code. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 205–214, New York, NY, USA, 2007. ACM.

[162] David Walker, Karl Crary, and Greg Morrisett. Typed memory management via static capabilities. *ACM Trans. Program. Lang. Syst.*, 22(4):701–771, 2000.

[163] David Walker and J. Gregory Morrisett. Alias types for recursive data structures. In *TIC '00: Selected papers from the Third International Workshop on Types in Compilation*, pages 177–206, London, UK, 2001. Springer-Verlag.

[164] Adam Welc, Suresh Jagannathan, and Antony L. Hosking. Transactional monitors for concurrent objects. In Odersky [118], pages 519–542.

[165] J. B. Wells. Typability and type checking in System F are equivalent and undecidable. *Ann. Pure Appl. Logic*, 98(1–3):111–156, 1999.

[166] Michael Edward Wolf. *Improving locality and parallelism in nested loops*. PhD thesis, Stanford University, Stanford, CA, USA, 1992.

[167] Andrew K. Wright. Simple imperative polymorphism. *Lisp Symb. Comput.*, 8(4):343–355, 1995.

[168] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 214–227, New York, NY, USA, 1999. ACM Press.