Existential Label Flow Inference via CFL Reachability

Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks

University of Maryland, College Park
{polyvios,jfoster,mwh}@cs.umd.edu

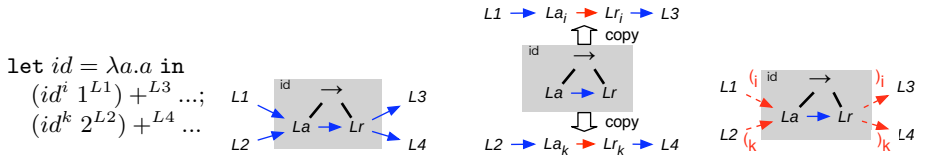
Abstract. In programming languages, existential quantification is useful for describing relationships among members of a structured type. For example, we may have a list in which there *exists* some mutual exclusion lock l in each list element such that l protects the data stored in that element. With 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 paper presents a context-sensitive *label flow analysis* algorithm with support for existential quantification. Label flow analysis is a core part of many static analysis systems. Following Rehof et al, we use context-free language (CFL) reachability 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, in the style of Mossin. We have implemented a variant of our analysis as part of a data race detection tool for C programs.

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. 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 [1] that cannot be precisely analyzed.

One way to solve this problem is to use *existential quantification* [2] 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 [3]; a pointer to data and the lock that must be held when accessing it [4, 5]; or a closure, consisting of a function and a pointer to its environment [6]. The important idea is that such relations are sound even when the identity of individual data structure elements cannot be discerned.

This paper presents a context-sensitive *label flow analysis* algorithm that supports existential quantification. Label flow analysis attempts to answer queries of the form "During program execution, can a value v flow to some expression e?" Answering such queries is at the core of a variety of static analyses, including points-to analysis [7, 8], information flow [9], type qualifier inference [10–12], and race detection [4]. Our goal is to provide a formal foundation for augmenting such analyses with support for existential quantification. The core result of this paper is a provably sound and efficient type inference system for label flow that supports existential quantification. This paper makes the following contributions:



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

Fig. 1. Universal Types Example

- We present COPY, a subtyping-based label-flow system in the style of Mossin [13]. In COPY, context sensitivity for functions corresponds to universal types (parametric polymorphism). Our contribution is to show how to support existential quantification using existential types [2], applying the duality of $\forall$ and $\exists$. We prove that the resulting system is sound. (Sect. 3)
- We present CFL, an alternative to COPY that supports efficient inference. Following Rehof et al [14, 15], determining flow in CFL is reduced to a context-free language (CFL) reachability problem, and the resulting inference system runs in time $O(n^3)$ in the worst case. Our contribution is to show that existentially-quantified flow can also be expressed as a CFL problem, and to prove that CFL is sound by reducing it to COPY. 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. (Sect. 4)
- We briefly discuss how a variation of CFL is used as part of LOCKSMITH, a race detection tool [4] for C programs that correlates memory locations to mutual exclusion locks protecting them. LOCKSMITH uses existential quantification to precisely relate locks and locations that reside within dynamic data structures, thereby eliminating a source of false alarms. (Sect. 2.3)

2 Polymorphism via Context-Free Language Reachability

We begin by introducing type-based label flow analysis, presenting the encoding of context sensitivity as universal types, and sketching our new technique for supporting first-class existential types. We also describe our application of these ideas to LOCKSMITH, a race detection tool for C [4]. Sects. 3 and 4 formally develop the label flow systems introduced here.

2.1 Universal Types and Label Flow

The goal of label flow analysis is to determine which values may flow to which operations. In the program in Fig. 1(a), values 1 and 2 are annotated with *flow labels* $L1$ and $L2$, respectively, and the two $+$ operations are labeled with $L3$ and $L4$. Therefore label flow analysis should show that $L1$ flows to $L3$ and $L2$ flows to $L4$. In this program we annotate calls to id with indices i and k, which we will explain shortly.

To compute the flow of labels, we perform a type- and constraint-based analysis in which base types are annotated with labels. For our example, the function id is given the type $int^{La} \to int^{Lr}$, 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 Fig. 1(b), and flow is determined by graph reachability. Thus the graph accurately shows that $L1$ flows to $L3$ and $L2$ flows to $L4$. However, the graph conflates the two calls to id—its type is monomorphic—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* using Hindley-Milner style universal types. The standard approach [13], shown in Fig. 1(c), is to give id a *polymorphically constrained* universal type $\forall La, Lr[La \leq Lr].int^{La} \to int^{Lr}$, 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.

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 [16–18] 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 [14]. This solution adds call and return edges to the graph and labels them with parentheses indexed by the call site, as shown in Fig. 1(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. Rehof et al [14] have shown that using CFL reachability with matched paths can be reduced to a type system with polymorphically constrained types.

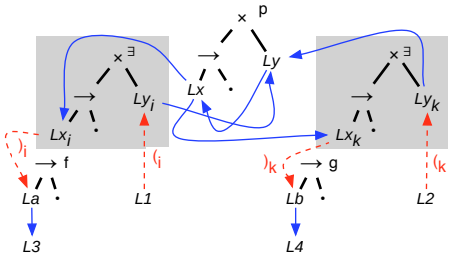
2.2 Existential Types and Label Flow

The goal of this paper is to show how to use existential quantification during static analysis to efficiently model properties of data structures more precisely. Consider the example shown in Fig. 2(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 Sect. 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

```
let f = λa.a +^L3 ··· in
let g = λb.b +^L4 ··· in
let p = if ··· then
  pack^i (f, 1^L1)
else
  pack^k (g, 2^L2) in
unpack (p1, p2) = p in
  p1 @ p2
```

(a) Source program



(b) Flow graph

Fig. 2. Existential Types Example

conservatively conflate the types at the two pack sites, generating spurious constraints $L1 \leq L4$ and $L2 \leq L3$. To solve this problem, Sect. 3 presents COPY, a system that can model p precisely by giving it a polymorphically constrained existential type $\exists Lx, Ly[Ly \leq Lx].(int^{Lx} \rightarrow int) \times int^{Ly}$, 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 $\texttt{pack}^i$, this type is instantiated to yield $L1 \leq La$, and since $La \leq L3$ we have $L1 \leq L3$ transitively. Instantiating at $\texttt{pack}^k$ 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 Fig. 1 because id is polymorphic in the label it is called with—whatever it is called with, it returns. Conversely, in Fig. 2 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.

The key contribution of this paper is to show how to perform inference with existential types efficiently using CFL reachability, as presented in Sect. 4. Fig. 2(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.

2.3 Existential Quantification and Race Detection

Our interest in studying existential label flow arose from the development of LOCKSMITH, a C race detection tool [4]. 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.

```
struct cache_entry { int refs; pthread_mutex_t refs_mutex; ... };

void cache_entry_addref(cache_entry *entry) { ...
  pthread_mutex_lock(&entry->refs_mutex);
  entry->refs++;
  pthread_mutex_unlock(&entry->refs_mutex);
... }
```

Fig. 3. Example code with a per-element lock

Each time a location ρ is accessed with lock ℓ held, LOCKSMITH generates a *correlation constraint* $\rho \rhd \ell$. 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 CFL reachability closure rules to solve for correlations context-sensitively.

During our experiments we found several examples of code similar to Fig. 3, which is taken from the *knot* multithreaded web server [19]. 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 $\exists \ell, \rho[\rho \rhd \ell].\{\texttt{refs} : \mathit{ref}^{\rho}\ \mathit{int}, \texttt{refs\_mutex} : \mathit{lock}\ \ell, \ldots\}$, and thus LOCKSMITH can verify that the lock `refs_mutex` always guards the `refs` field in a given node.

While our prior work sketches the use of existential types, it gives neither type rules nor proofs for them, which are the main contributions of this paper. The remainder of this paper focuses exclusively on existential types for label flow, and we refer the reader to our other paper for details on LOCKSMITH [4].

3 Label Flow with Polymorphically Constrained Types

We begin our formal presentation by studying label flow in the context of a polymorphically-constrained type system COPY, which is essentially Mossin's label flow system [13] extended to include existential types. Note that COPY supports label polymorphism but not polymorphism in the type structure. We use the following source language throughout the paper:

$$\begin{aligned} e ::= {} & n^L \mid x \mid \lambda^L x.e \mid e_1 @^L e_2 \mid \texttt{if0}^L\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid (e_1, e_2)^L \mid e.^L j \\ & \mid\ \texttt{let}\ f = e_1\ \texttt{in}\ e_2 \mid \texttt{fix}\, f.e_1 \mid f^i \mid \texttt{pack}^{L,i}\ e \mid \texttt{unpack}^L\ x = e_1\ \texttt{in}\ e_2 \end{aligned}$$

In this language, constructors and destructors are annotated with constant labels L. The goal of our type system is to determine which constructor labels flow to which

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

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

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

$$\text{[Cond]}\frac{C; \Gamma \vdash_{cp} e_0 : int^l \quad C \vdash l \leq L \quad C; \Gamma \vdash_{cp} e_1 : \tau \quad C; \Gamma \vdash_{cp} e_2 : \tau}{C; \Gamma \vdash_{cp} \texttt{if0}^L\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau} \qquad \text{[Sub]}\frac{C; \Gamma \vdash_{cp} e : \tau_1 \quad C; \emptyset \vdash \tau_1 \leq \tau_2}{C; \Gamma \vdash_{cp} e : \tau_2}$$

Fig. 4. COPY Monomorphic Rules

destructor labels. For example, in the expression $(\lambda^L x.e)@^{L'} e'$, the label L flows to the label L'. Our language 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. Our language also includes binding constructs `let` and `fix`, which introduce universal types. Each use of a universally quantified function f^i is indexed by an *instantiation site* i. Expressions also include existential packages, which are created with $\texttt{pack}^{L,i}$ 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 Sect. 4.

The types and environments used by COPY are given by the following grammar:

$$\begin{array}{lll} \text{types} & \tau ::= int^l \mid \tau \to^l \tau \mid \tau \times^l \tau \mid \exists^l \vec{\alpha}[C].\tau & \text{schemes} \quad \sigma ::= \forall \vec{\alpha}[C].\tau \mid \tau \\ \text{labels} & l ::= L \mid \alpha & \text{constraints} \ C ::= \emptyset \mid \{l \leq l\} \mid C \cup C \\ \text{env.} & \Gamma ::= \cdot \mid \Gamma, x : \sigma & \end{array}$$

Types include integers, functions, pairs, and existential types. All types are annotated with flow labels l, which may be either constant labels L from the program text or label variables α. Type schemes include normal types and polymorphically-constrained universal types of the form $\forall \vec{\alpha}[C].\tau$. Here C is a set of *flow constraints* each of the form $l \leq l'$. In our type rules, *substitutions* ϕ map label variables to labels. The universal type $\forall \vec{\alpha}[C].\tau$ stands for any type $\phi(\tau)$ where $\phi(C)$ is satisfied, for any substitution ϕ. When $l \leq l'$, we say that label l flows to label l'. The type $\exists^l \vec{\alpha}[C].\tau$ stands for the type $\phi(\tau)$ where constraints $\phi(C)$ are satisfied for *some* substitution ϕ. Universal types may only appear in type environments while existential types may appear arbitrarily. The free labels of types ($\mathit{fl}(\tau)$) and environments ($\mathit{fl}(\Gamma)$) are defined as usual.

The expression typing rules are presented in Figs. 4 and 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. In these type rules $C \vdash l \leq l'$ means that the constraint $l \leq l'$ is in the

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

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

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

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

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

Fig. 5. COPY Polymorphic Rules

transitive closure of the constraints in C, and $C \vdash C'$ means that all constraints in C' are in the transitive closure of C.

Fig. 4 contains the monomorphic typing rules, which are as in the standard λ calculus except for the addition of labels and subtyping. The constructor rules ([Int], [Lam] and [Pair]) require $C \vdash L \leq l$, i.e., the constructor label L must flow to the corresponding label of the constructed type. The destructor rules ([Cond], [App] and [Proj]) require the converse. The subtyping rule [Sub] is discussed below.

Fig. 5 contains the polymorphic typing rules. Universal types are introduced by [Let] and [Fix]. As is standard, we allow generalization only of label variables that are not free in the type environment Γ. 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 $C \vdash \phi(C')$ 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 $\phi(\tau)$ is abstracted to a type $\exists^l \vec{\alpha}[C'].\tau$. Notice that the substitution maps abstract τ and C' to concrete $\phi(\tau)$ and $\phi(C')$—creating an existential corresponds to passing an argument to "the rest of the program," as if that were universally quantified in $\vec{\alpha}$, and the constraints C' are determined by how the existential package is used after it is unpacked. Similarly to [Inst], the [Pack] premise $C \vdash \phi(C')$ inlines the abstract constraints $\phi(C')$ into the current constraints.

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

$$\text{[Sub-Pair]}\ \frac{\begin{array}{c} C; D \vdash l \leq l' \\ C; D \vdash \tau_1 \leq \tau_1' \\ C; D \vdash \tau_2 \leq \tau_2' \end{array}}{C; D \vdash \tau_1 \times^l \tau_2 \leq \tau_1' \times^{l'} \tau_2'} \qquad \text{[Sub-Fun]}\ \frac{\begin{array}{c} C; D \vdash l \leq l' \\ C; D \vdash \tau_1' \leq \tau_1 \\ C; D \vdash \tau_2 \leq \tau_2' \end{array}}{C; D \vdash \tau_1 \to^l \tau_2 \leq \tau_1' \to^{l'} \tau_2'}$$

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

Fig. 6. COPY Subtyping

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 $\vec{\alpha}$ needed to check e_2 must be in C'. Second, the labels $\vec{\alpha}$ must not escape the scope of the `unpack` (as is standard [2]), which is ensured by the subset constraint.

The [Sub] rule in Fig. 4 uses the subtyping relation shown in Fig. 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 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 $C \vdash \exists\alpha[\emptyset].int^\alpha \leq \exists\alpha[\emptyset].int^\beta$. 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 $C \vdash (\exists\alpha, \beta[\alpha \leq \beta].int^\alpha \to int^\beta) \leq (\exists\alpha, \beta[\emptyset].int^\alpha \to int^\alpha)$, which is permissible because β is a bound variable with no other lower bounds except α, hence it can be set to α 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 CFL reachability-based inference.

We prove soundness for COPY using subject reduction. Using a standard small-step operational semantics $e \longrightarrow e'$, we define a *flow-preserving evaluation step* as one

[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 CFL to COPY discussed in Sect. 4.

[2] Our technical report [20] uses an equivalent version of D that makes the reduction proof easier.

whose flow is allowed by some constraint set C. Then we prove that if a program is well-typed according to C then it always preserves flow.

Definition 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 1 (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 a companion technical report [20].

4 CFL-Based Label Flow Inference

The COPY 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 CFL based on CFL reachability, in the style of Rehof et al [14, 15]. This system uses a single, global set of constraints, which correspond to flow graphs like those shown in Figs. 1(d) and 2. Given a flow graph, we can answer queries "Does any value labeled l_1 flow to a destructor labeled l_2?", written $l_1 \rightsquigarrow l_2$, by using CFL reachability. We first present type checking rules for CFL and then explain how they are used to interpret the flow graph in Fig. 2. Then we explain how the rules can be interpreted to yield an efficient inference algorithm. Finally, we prove that CFL reduces to COPY and thus is sound.

Types in CFL are as follows:

$$\text{types } \tau ::= int^l \mid \tau \rightarrow^l \tau \mid \tau \times^l \tau \mid \exists^l \vec{\alpha}.\tau \qquad \text{schemes } \sigma ::= (\forall \vec{\alpha}.\tau, \vec{l}) \mid \tau$$

In contrast to COPY, universal types $(\forall \vec{\alpha}.\tau, \vec{l})$ and existential types $\exists^l \vec{\alpha}.\tau$ 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 [14, 21]. For clarity universal types also include $\vec{\alpha}$, the set of labels that are quantified, but it is always the case that $\vec{\alpha} = \mathit{fl}(\tau) \setminus \vec{l}$. 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 CFL have the form $I; C; \Gamma \vdash e : \tau$, where I and C describe the edges in the flow graph. C has the same form as in COPY, consisting of subtyping constraints $l \leq l'$ (shown as unlabeled directed edges in Figs. 1 and 2). I contains *instantiation constraints* [14] of the form $l \preceq^i_p l'$. 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.) 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 $l \longrightarrow^{)i} l'$. When p is $-$ the reverse holds, and so we draw the constraint $l \preceq^i_- l'$ as an edge $l' \longrightarrow^{(i} l$. Instantiation constraints correspond to substitutions in COPY, and they enable context-sensitivity without the

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

$$\text{[Lam]}\frac{I; C; \Gamma, x : \tau \vdash_{cfl} e : \tau' \quad C \vdash L \leq l}{I; C; \Gamma \vdash_{cfl} \lambda^L x.e : \tau \to^l \tau'} \qquad \text{[App]}\frac{I; C; \Gamma \vdash_{cfl} e_1 : \tau \to^l \tau' \quad I; C; \Gamma \vdash_{cfl} e_2 : \tau \quad C \vdash l \leq L}{I; C; \Gamma \vdash_{cfl} e_1 @^L e_2 : \tau'}$$

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

$$\text{[Cond]}\frac{I; C; \Gamma \vdash_{cfl} e_0 : int^l \quad C \vdash l \leq L \quad I; C; \Gamma \vdash_{cfl} e_1 : \tau \quad I; C; \Gamma \vdash_{cfl} e_2 : \tau}{I; C; \Gamma \vdash_{cfl} \texttt{if0}^L\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau} \qquad \text{[Sub]}\frac{I; C; \Gamma \vdash_{cfl} e : \tau_1 \quad C; \emptyset; \emptyset \vdash \tau_1 \leq \tau_2}{I; C; \Gamma \vdash_{cfl} e : \tau_2}$$

Fig. 7. CFL Monomorphic Rules

need to copy constraint sets. A full discussion of instantiation constraints is beyond the scope of this paper; see Rehof et al [14] for a thorough description.

The monomorphic rules for CFL are presented in Fig. 7. With the exception of [Sub] and the presence of I, these are identical to the rules in Fig. 4. Fig. 8 presents the polymorphic CFL rules. In these type rules $I \vdash l \preceq^i_p l'$ means that the instantiation constraint $l \preceq^i_p l'$ is in I. We define $fl(\tau)$ to be the free labels of a type as usual, except $fl(\forall\vec{\alpha}.\tau, \vec{l}) = (fl(\tau) \setminus \vec{\alpha}) \cup \vec{l}$. Rules [Let] and [Fix] bind f to a universal type. As is standard we cannot quantify label variables that are free in the environment Γ, which we represent by setting $\vec{l} = fl(\Gamma)$ in type $(\forall\vec{\alpha}.\tau_1, \vec{l})$. The [Inst] rule instantiates the type τ of f to τ' using an instantiation constraint $I; \emptyset \vdash \tau \preceq^i_+ \tau' : \phi$. This constraint represents a renaming ϕ, analogous to that in COPY's [Inst] rule, such that $\phi(\tau) = \tau'$. 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 COPY's [Pack], there is a substitution such that $\tau' = \phi(\tau)$, and thus 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 $\vec{\alpha}$ is not specified. As in other systems for inferring first-class existential and universal types [22–25], 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].

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 (see below) between any labels in $\vec{\alpha}$ and any labels in $\vec{l}$, which is

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

$$\text{[Fix]}\ \frac{I;C;\Gamma, f : (\forall \vec{\alpha}.\tau, \vec{l}) \vdash_{cfl} e : \tau \quad \vec{\alpha} = \mathit{fl}(\tau) \setminus \mathit{fl}(\Gamma) \quad \vec{l} = \mathit{fl}(\Gamma) \quad I;\emptyset \vdash \tau \preceq^i_+ \tau' : \phi \quad I \vdash \vec{l} \preceq^i_+ \vec{l} \quad I \vdash \vec{l} \preceq^i_- \vec{l}}{I;C;\Gamma \vdash_{cfl} \texttt{fix}\, f.e : \tau'}$$

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

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

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

Fig. 8. CFL Polymorphic Rules

the set of labels that could escape. If this condition is violated, then the existentially quantified labels $\vec{\alpha}$ chosen by the programmer are invalid and the program is rejected. The [Unpack] rule in COPY does not forbid interaction between free and bound labels, and therefore CFL is strictly weaker than COPY. However, without this restriction we can produce cases where mixing existentials and universals produces flow paths that should be valid but have mismatched parentheses. Sect. 4.3 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 CFL reachability-based inference.

Fig. 9 defines the subtyping relation used in [Sub]. The only interesting difference with COPY arises because of alpha-conversion. In COPY alpha-conversion is implicit, and only trivial constraints are allowed between bound labels (by [Sub-Label-2] of Fig. 6). We cannot use implicit alpha-conversions in CFL, however, because we are producing a single, global set of constraints. Thus instead of the single D used in COPY's[Sub] rule, CFL 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 Fig. 9, which corresponds to [Sub-Label-2] in Fig. 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 COPY system, but it adds no expressive power and complicates proving soundness.

Fig. 10 defines instantiation constraints on types in terms of instantiation constraints on labels. Judgments have the form $I; D \vdash \tau \preceq^i_p \tau' : \phi$, where ϕ is the renaming defined by the instantiation and D is the same as in Fig. 6—we do not need to allow alpha-

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

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

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

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

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

$$\text{[Sub-}\exists\text{]}\frac{\begin{array}{c}\Delta'_1 = \Delta_1 \oplus \vec{\alpha_1} \quad \Delta'_2 = \Delta_2 \oplus \vec{\alpha_2} \quad \phi(\vec{\alpha_2}) = \vec{\alpha_1} \\ C; \Delta'_1; \Delta'_2 \vdash \tau_1 \leq \tau_2 \qquad C; \Delta_1; \Delta_2 \vdash l_1 \leq l_2\end{array}}{C; \Delta_1; \Delta_2 \vdash \exists^{l_1} \vec{\alpha_1}.\tau_1 \leq \exists^{l_2} \vec{\alpha_2}.\tau_2}$$

Fig. 9. CFL Subtyping

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 Fig. 10 are standard, and details can be found in Rehof et al [14].

Given a flow graph described by constraints I and C, Fig. 11 gives inference rules to compute the relation $l_1 \rightsquigarrow l_2$, which means label l_1 flows to label l_2. Rule [Level] states that constraints in C correspond to flow (represented as unlabeled edges in the flow graph). Rule [Trans] adds transitive closure. Rule [Match] allows flow on a matched path $l_0 \longrightarrow^{(i} l_1 \rightsquigarrow l_2 \longrightarrow^{)i} l_3$. This rule corresponds to "copying" the constraint $l_1 \rightsquigarrow l_2$ to a constraint $l_0 \rightsquigarrow l_3$ at instantiation site i. Rule [Constant] adds a "self-loop" that permits matching flows to or from any constant label. We generate these edges because constants are global names and thus are context-insensitive.

Note that our relation $\rightsquigarrow$ corresponds to the $\rightsquigarrow_m$ relation from Rehof et al [14], 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 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.

$$\text{[Inst-Ind-1]}\ \frac{l, l' \notin D \qquad I \vdash l \preceq_p^i l'}{I; D \vdash l \preceq_p^i l' : \emptyset} \qquad \text{[Inst-Ind-2]}\ \frac{l \in D}{I; D \vdash l \preceq_p^i l : \phi}$$

$$\text{[Inst-Pair]}\ \frac{\begin{array}{c} I; D \vdash l \preceq_p^i l' : \phi \\ I; D \vdash \tau_1 \preceq_p^i \tau_1' : \phi \\ I; D \vdash \tau_2 \preceq_p^i \tau_2' : \phi \end{array}}{I; D \vdash \tau_1 \times^l \tau_2 \preceq_p^i \tau_1' \times^{l'} \tau_2' : \phi} \qquad \text{[Inst-Fun]}\ \frac{\begin{array}{c} I; D \vdash l \preceq_p^i l' : \phi \\ I; D \vdash \tau_1 \preceq_{\bar{p}}^i \tau_1' : \phi \\ I; D \vdash \tau_2 \preceq_p^i \tau_2' : \phi \end{array}}{I; D \vdash \tau_1 \to^l \tau_2 \preceq_p^i \tau_1' \to^{l'} \tau_2' : \phi}$$

$$\text{[Inst-Int]}\ \frac{I; D \vdash l \preceq_p^i l' : \phi}{I; D \vdash int^l \preceq_p^i int^{l'} : \phi} \qquad \text{[Inst-}\exists\text{]}\ \frac{\begin{array}{c} D' = D \cup \vec{\alpha} \qquad I; D' \vdash \tau_1 \preceq_p^i \tau_2 : \phi \\ I; D \vdash l_1 \preceq_p^i l_2 : \phi \end{array}}{I; D \vdash \exists^{l_1} \vec{\alpha}.\tau_1 \preceq_p^i \exists^{l_2} \vec{\alpha}.\tau_2 : \phi}$$

Fig. 10. CFL Instantiation

$$\text{[Level]}\ \frac{C \vdash l_1 \leq l_2}{I; C \vdash l_1 \rightsquigarrow l_2} \qquad \text{[Trans]}\ \frac{I; C \vdash l_0 \rightsquigarrow l_1 \qquad I; C \vdash l_1 \rightsquigarrow l_2}{I; C \vdash l_0 \rightsquigarrow l_2}$$

$$\text{[Constant]}\ \frac{}{I; C \vdash L \preceq_p^i L} \qquad \text{[Match]}\ \frac{I \vdash l_1 \preceq_-^i l_0 \qquad I; C \vdash l_1 \rightsquigarrow l_2 \qquad I \vdash l_2 \preceq_+^i l_3}{I; C \vdash l_0 \rightsquigarrow l_3}$$

Fig. 11. Flow

4.1 Example

Consider again the example in Fig. 2. The expression $\texttt{pack}^i(f, 1^{L1})$ 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

$$I; C \vdash (int^{Lx_i} \to int) \times int^{Ly_i} \preceq_-^i (int^{La} \to int) \times int^{L1}$$

Proving this judgment requires appealing in several places to [Inst-Ind-1], whose premise $I \vdash l \preceq_p^i l'$ requires that I 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).

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 $I; C \vdash L1 \rightsquigarrow L3$ and $I; C \vdash L2 \rightsquigarrow L4$ (but $I; C \not\vdash L1 \rightsquigarrow L4$). 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.

4.2 An Inference Algorithm

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

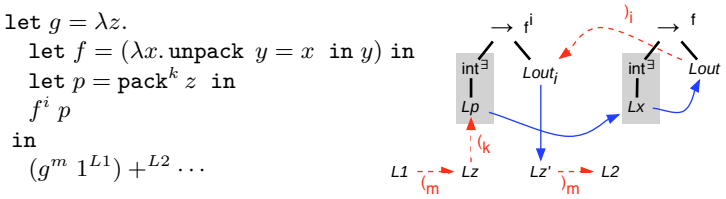
First, we type the program according to the rules in Figs. 7-10. As usual the non-syntactic rule [Sub] can be incorporated into the remaining rules to produce a syntax-directed system [26]. During typing, we interpret a premise $C \vdash l \leq l'$ or $I \vdash \vec{l} \preceq_p^i \vec{l}$ as *generating* a constraint; i.e., we add $l \leq l'$ (or $\vec{l} \preceq_p^i \vec{l}$) to the set of global constraints C (or I). Free occurrences of l in the rules are interpreted as fresh label variables. For example, in [Int] we interpret l as a fresh variable α 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 C and I.

Next, we compute all flows according to the rules in Fig. 11. Excluding the final premise of [Unpack] and the D's in [Sub] and [Inst], performing typing and computing all flows takes time $O(n^3)$ [14]. 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 $\vec{\alpha}$ as we traverse an existential type. We can assume without loss of generality that $|\vec{\alpha}| \leq |\mathit{fl}(\tau)|$ in an existential type, so traversing $\vec{\alpha}$ 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 $\vec{\alpha}$ 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 $O(n^3) + O(n^3) = O(n^3)$.

4.3 Differences between COPY and CFL

As mentioned in Sect. 4, if we weaken CFL's [Unpack] rule to permit existentially bound labels to interact with free labels, then we can construct examples with mismatched flow. Fig. 12(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 data flow. 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 $\texttt{pack}^k$, the programmer wishes to quantify the type of the packed integer, and then compare COPY and CFL as applied to the program.



(a) Source program (b) Flow graph

Fig. 12. Example with Mismatched Flow

The COPY types rules assign f the type scheme

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

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 COPY (technically, we need an application of [Sub] to achieve this), but not allowed in CFL. 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 CFL to the same program. Fig. 12(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 CFL'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 $\texttt{pack}^k$, 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 CFL, we prevent instantiations from renaming labels by adding "self-loops," as in [Inst] in Fig. 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 Fig. 8, we require existentially-quantified labels to not have any flow with escaping labels to forbid this example.

4.4 Soundness

We have proven that programs that check under CFL are reducible to COPY. The first step is to define a translation function $\Psi_{C,I}$ that takes CFL types and transforms them to COPY types. For monomorphic types $\Psi_{C,I}$ is simply the identity. To translate a polymorphic CFL type $(\forall\vec{\alpha}.\tau, \vec{l})$ or $\exists^l\vec{\alpha}.\tau$ into a COPY type $\forall\vec{\alpha}[C'].\tau$ or $\exists^l\vec{\alpha}[C'].\tau$, respectively, $\Psi_{C,I}$ needs to produce a bound constraint set C'. Rehof et al [14, 15] were able to choose $C' = C^I = \{l_1 \leq l_2 \mid I; C \vdash l_1 \rightsquigarrow l_2\}$, i.e., the closure of C and I. However, the addition of first class existentials causes this approach to fail, because, for example, instantiating a $\forall$ type containing a type $\exists^l\vec{\alpha}[C^I].\tau$ could rename some variables in C^I (since C^I contains all variables used in the program) and thereby violate the inductive hypothesis. Thus we introduce a projection function ψ_S, where we define

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

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

Theorem 2 (Reduction from CFL to COPY). *Let $\mathcal{D}$ be a normal* CFL *derivation of $I; C; \Gamma \vdash_{cfl} e : \tau$. Then $C^I_{fl(\Gamma)\cup fl(\tau)}; \Psi_{C,I}(\Gamma) \vdash_{cp} e : \Psi_{C,I}(\tau)$.*

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 CFL to COPY we can prove the hypothesis $C \vdash \phi(C')$. The other key part occurs in translating an occurrence of [Unpack] from CFL to COPY. In this case, by induction on the typing derivation for e_2 we have $C^I_{fl(\Gamma)\cup fl(\tau)\cup fl(\tau')}; \Psi_{C,I}(\Gamma), x : \Psi_{C,I}(\tau) \vdash_{cp} e_2 : \Psi_{C,I}(\tau')$. By the last hypothesis of [Unpack] in CFL, we know that there are no constraints between the quantified labels $\vec{\alpha}$ and any other labels. Thus we can partition the constraints on the left-hand side of the above typing judgment into two disjoint sets: $C^I_{(fl(\Gamma)\cup fl(\tau)\cup fl(\tau'))\setminus\vec{\alpha}}$ and $C^I_{\vec{\alpha}}$. The former are the constraints needed to type check e_1 in COPY, 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 COPY. A full, detailed proof can be found in our companion technical report [20].

By combining Theorems 1 and 2, we then have soundness for the flow relation $\rightsquigarrow$ computed by CFL. Notice that we have shown reduction but not equivalence. Rehof et al [14, 15] 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 CFL. We leave it as an open question whether this condition can be relaxed to yield provably equivalent systems.

5 Related Work

Our work builds directly on the CFL reachability-based label flow system of Rehof et al [14]. Their cubic-time algorithm for polymorphic recursive label flow inference improves on the previously best-known $O(n^8)$ algorithm [13]. The idea of using CFL reachability in static analysis is due to Reps et al [27], who applied it to first-order data flow 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 [28] (p. 377), in which polymorphism is first class and type inference is undecidable [29]. There have been several proposals to support first-class polymorphic type inference using type annotations to avoid the undecidability problem. In ML^{F} [22], programmers annotate function arguments that have universal types. Laufer and Odersky [23] propose an extension to ML with first-class existential types, and Remy [24] 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 [25] extends HM(X) [30], 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.

6 Conclusion

Existential quantification can be used to precisely characterize relationships within elements of a dynamic data structure, even when the precise identity of those elements is unknown. This paper aims to set a firm theoretical foundation on which to build efficient program analyses that benefit from existential quantification. Our main contribution is a context-sensitive inference algorithm for label flow analysis that supports existential quantification. Programmers specify where existentials are introduced and eliminated, and our inference algorithm automatically infers the bounds on their flow. Our algorithm is efficient, employing context free language (CFL) reachability in the style of Rehof et al [14], and we prove it sound by reducing it to a system based on polymorphically-constrained types in the style of Mossin [13]. We have adapted our algorithm to improve the precision of LOCKSMITH, a tool that aims to prove the absence of race conditions in C programs [4] by correlating locks with the locations they protect. We plan to explore other applications of existential label flow in future work.

Acknowledgments

We would like to thank Manuel Fähndrich, Mike Furr, Ben Liblit, Nik Swamy, and the anonymous referees for their helpful comments. This research was supported in part by NSF grants CCF-0346982, CCF-0346989, CCF-0430118, and CCF-0524036.

References

1. Das, M.: Unification-based Pointer Analysis with Directional Assignments. In: The 2000 Conference on Programming Language Design and Implementation, Vancouver B.C., Canada (2000) 35–46
2. Mitchell, J.C., Plotkin, G.D.: Abstract types have existential type. ACM Transactions on Programming Languages and Systems **10** (1988) 470–502
3. Xi, H., Pfenning, F.: Dependent Types in Practical Programming. In: The 26th Annual Symposium on Principles of Programming Languages, San Antonio, Texas (1999) 214–227
4. Pratikakis, P., Foster, J.S., Hicks, M.: LOCKSMITH: Context-Sensitive Correlation Analysis for Race Detection. In: The 2006 Conference on Programming Language Design and Implementation, Ottawa, Canada (2006) To appear.
5. Flanagan, C., Abadi, M.: Types for Safe Locking. In Swierstra, D., ed.: 8th European Symposium on Programming. Volume 1576 of Lecture Notes in Computer Science., Amsterdam, The Netherlands, Springer-Verlag (1999) 91–108
6. Minamide, Y., Morrisett, G., Harper, R.: Typed closure conversion. In: The 23rd Annual Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida (1996) 271–283
7. Fähndrich, M., Rehof, J., Das, M.: Scalable Context-Sensitive Flow Analysis using Instantiation Constraints. In: The 2000 Conference on Programming Language Design and Implementation, Vancouver B.C., Canada (2000) 253–263
8. Das, M., Liblit, B., Fähndrich, M., Rehof, J.: Estimating the Impact of Scalable Pointer Analysis on Optimization. In Cousot, P., ed.: Static Analysis, Eighth International Symposium, Paris, France (2001) 260–278

9. Myers, A.C.: Practical Mostly-Static Information Flow Control. In: The 26th Annual Symposium on Principles of Programming Languages, San Antonio, Texas (1999) 228–241
10. Foster, J.S., Johnson, R., Kodumal, J., Aiken, A.: Flow-insensitive type qualifiers. (ACM Transactions on Programming Languages and Systems) To appear.
11. Kodumal, J., Aiken, A.: The Set Constraint/CFL Reachability Connection in Practice. In: The 2004 Conference on Programming Language Design and Implementation, Washington, DC (2004) 207–218
12. Johnson, R., Wagner, D.: Finding User/Kernel Bugs With Type Inference. In: The 13th Usenix Security Symposium, San Diego, CA (2004)
13. Mossin, C.: Flow Analysis of Typed Higher-Order Programs. PhD thesis, DIKU, Department of Computer Science, University of Copenhagen (1996)
14. Rehof, J., Fähndrich, M.: Type-Based Flow Analysis: From Polymorphic Subtyping to CFL-Reachability. In: The 28th Annual Symposium on Principles of Programming Languages, London, United Kingdom (2001) 54–66
15. Fähndrich, M., Rehof, J., Das, M.: From Polymorphic Subtyping to CFL Reachability: Context-Sensitive Flow Analysis Using Instantiation Constraints. Technical Report MS-TR-99-84, Microsoft Research (2000)
16. Flanagan, C., Felleisen, M.: Componential Set-Based Analysis. In: The 1997 Conference on Programming Language Design and Implementation, Las Vegas, Nevada (1997) 235–248
17. Fähndrich, M., Aiken, A.: Making Set-Constraint Based Program Analyses Scale. In: First Workshop on Set Constraints at CP'96. (1996) Available as CSD-TR-96-917, University of California at Berkeley.
18. Fähndrich, M.: BANE: A Library for Scalable Constraint-Based Program Analysis. PhD thesis, University of California, Berkeley (1999)
19. von Behren, R., Condit, J., Zhou, F., Necula, G.C., Brewer, E.: Capriccio: Scalable threads for internet services. In: ACM Symposium on Operating Systems Principles. (2003)
20. Pratikakis, P., Hicks, M., Foster, J.S.: Existential Label Flow Inference via CFL Reachability. Technical Report CS-TR-4700, University of Maryland, Computer Science Department (2005)
21. Henglein, F.: Type Inference with Polymorphic Recursion. ACM Transactions on Programming Languages and Systems **15** (1993) 253–289
22. Botlan, D.L., Rémy, D.: ML^F—Raising ML to the Power of System F. In: The Eighth International Conference on Functional Programming, Uppsala, Sweden (2003) 27–38
23. Läufer, K., Odersky, M.: Polymorphic type inference and abstract data types. ACM Transactions on Programming Languages and Systems **16** (1994) 1411–1430
24. Rémy, D.: Programming objects with MLART: An extension to ML with abstract and record types. In: The International Symposium on Theoretical Aspects of Computer Science, Sendai, Japan (1994) 321–346
25. Simonet, V.: An Extension of HM(X) with Bounded Existential and Universal Data Types. In: The Eighth International Conference on Functional Programming, Uppsala, Sweden (2003) 39–50
26. Mitchell, J.C.: Type inference with simple subtypes. Journal of Functional Programming **1** (1991) 245–285
27. Reps, T., Horwitz, S., Sagiv, M.: Precise Interprocedural Dataflow Analysis via Graph Reachability. In: The 22nd Annual Symposium on Principles of Programming Languages, San Francisco, California (1995) 49–61
28. Pierce, B.C.: Types and Programming Languages. The MIT Press (2002)
29. Wells, J.B.: Typability and type checking in System F are equivalent and undecidable. Ann. Pure Appl. Logic **98** (1999) 111–156
30. Odersky, M., Sulzmann, M., Wehr, M.: Type inference with constrained types. Theory and Practice of Object Systems **5** (1999) 35–55