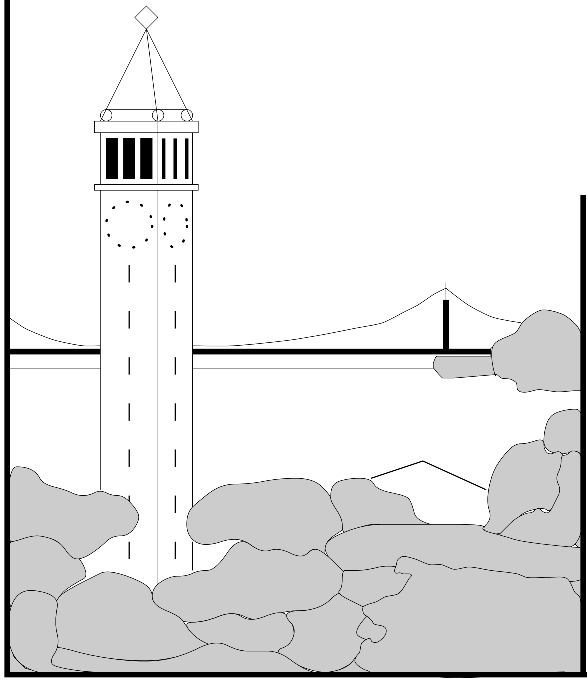
Checking Programmer-Specified Non-Aliasing

Jeffrey S. Foster *Alex Aiken*



Report No. UCB/CSD-01-1160

October 2001

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

Checking Programmer-Specified Non-Aliasing[*]

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

October 3, 2001

Abstract

We study the new ANSI C type qualifier `restrict`, which allows programmers to specify pointers that are not aliased to other pointers. The main contribution of this paper is a formal semantics for `restrict` and a type and effect system for checking that `restrict`-annotated programs are correct with respect to our semantics. We give an efficient inference algorithm for our type system and describe natural extensions of our type system to include subtyping, parametric polymorphism, and `affects` clauses that capture the effects of calling a function. We also discuss ways in which our type system differs from the ANSI C standard.

1 Introduction

Almost all program analyses for languages with pointers must perform *alias analysis*: when a program indirectly loads or stores through a pointer p, the analysis must determine to which location(s) p points.

The research literature abounds with proposed alias analysis techniques, including [LR92, And94, BCCH94, EGH94, WL95, Ste96, Das00], some of which scale to very large programs [FFSA98, RF01, HT01]. Almost all of these techniques are fully automatic. That is, the analyses take a bare program and infer all possible aliasing.

While these techniques show great promise, we believe that alias analysis is too important and too brittle to be left entirely to the compiler (see Section 1.3 for two examples). In this paper, we study `restrict`, a type qualifier defined in the new ANSI C standard (C99) [ANS99]. Intuitively, if a programmer marks a function parameter x in a C99 program with `restrict`, then a compiler may assume—without any checking—that at the beginning of the function, no other paths are aliases of the location $*x$. This idea harks back to the semantics of FORTRAN. The FORTRAN language specification states that if the programmer passes an array to a function multiple times under different names, then that array may not be modified by the function, and it is up to the programmer to enforce this requirement [ANS78, ABM$^+$97]. The qualifier `restrict` tells a C compiler where to use FORTRAN-style semantics.

We believe that `restrict` is useful in any language with updateable references, not just C. In this paper, we study `restrict` as an extension to the λ-calculus with updateable references. Our version of `restrict` stays close to the definition in the C99 standard but differs in some respects, discussed in Section 6. We believe that our study of `restrict` in a concise formal system helps explain the C99 definition and may be a guide to future revisions of the standard.

The main contributions of this paper are:

- We formalize `restrict`, giving it a precise semantics (Section 3).

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

- We give a *type and effect system* [LG88] for checking that a `restrict`-annotated program is correct with respect to our semantics (Sections 2 and 3).
- We present an $O(kn + n\alpha(n))$ algorithm (Section 4) for type inference, where n is the size of the typed program, k is the number of `restrict` annotations in the program, and $\alpha(\cdot)$ is the inverse Ackerman's function.
- We describe three natural extensions to our language: subtyping between function types [TJ95], parametric polymorphism, and an `affects` clause for describing the effects of calling a function (Section 5).

1.1 `restrict`

Let `p` be a pointer declared as

```
int *restrict p;
```

and suppose `p` points to object `X`. Then the C99 standard requires that, within the scope of `p`, all accesses to `X` are through the name `p`.[1]

The qualifier `restrict` is useful because it allows the programmer to tell the compiler that certain pointers are never aliased. The C99 specification gives the following example code illustrating the use of restrict ([ANS99], page 111):

```
void f(int n, int *restrict p, int *restrict q) {
  while (n-- > 0)
    *p++ = *q++;
}
```

Here because the types of `p` and `q` are annotated with `restrict`, the compiler can infer that `*p` is only accessed through the name `p` and `*q` is only accessed through the name `q`, and thus `*p` and `*q` are not aliased.

1.2 Types for `restrict`

We check the correctness of `restrict` annotations using a type and effect system. Pointers in our type system are given types of the form $ref^{\rho}(\tau)$, meaning a pointer to abstract location ρ, where abstract location ρ contains a value of type τ.

Our type system proves judgments of the form

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

where L, an effect, is the set of abstract locations the evaluation of e may read or write. Intuitively, within the scope of a `restrict`-qualified pointer `p`, we cannot dereference any aliases of `p`. We enforce this requirement with constraints of the form $\rho \notin L$, which holds only if ρ is never accessed during evaluation of e.

Our type system admits an efficient inference algorithm. Our type inference system generates constraints of the form $\tau_1 = \tau_2$, $L \subseteq \varepsilon$, and $\rho \notin L$, where ε is a variable ranging over sets of effects. As stated above, these constraints can be solved in $O(kn + n\alpha(n))$ time, where n is the size of the typed program and k is the number of `restrict` annotations in the program.

1.3 Applications

We briefly outline two useful applications of `restrict`.

[1]The ANSI C standard actually states that this property must hold only if `X` is modified within the scope of `p`. We believe this extra condition makes the definition of `restrict` more complicated to little benefit (see Section 6).

Optimization. As mentioned above, `restrict` was added to the C standard to enable compilers to recover non-aliasing information and thus to optimize pointer accesses. For example, given the code for function `f()` above, a compiler can use the assumption that `*p` and `*q` are not aliased to reorder the assignments in the loop across iterations. Without `restrict` annotations, a compiler has no indication of the programmer's intention (that `p` and `q` are not aliased), and the programmer has no way to indicate that a compiler should try to optimize the loop.

In the C standard `restrict` qualifiers are unchecked, and hence there is no guarantee that optimizations that rely on `restrict` are valid. If a program using `restrict` type checks in our system, however, then the `restrict` annotations are guaranteed to be correct (up to the usual C features that defeat the type system, such as casting), and hence optimizations based on `restrict` are safe.

Strong Updates. A key benefit of `restrict` that does not seem to be anticipated in the standard is local recovery of the ability to perform *strong updates* [CWZ90] in a flow-sensitive analysis. Consider the following code skeleton:

```
void foo(int *x) {... ① *x = e; ② ...}
```

Here ① is the program point just before the assignment `*x = e`, and ② is the point just after the assignment. Assume `x` does not appear to the left of ①.

Suppose we perform a standard forward data-flow analysis on this program, and suppose *loc* is our abstraction for the location that `x` points to. Let $[\![\cdot]\!]$ be our interpretation function mapping program values and expressions to abstract values.

There are two ways to handle the assignment to `*x`. If we know that *loc* is a single, unique location, then we can perform a strong update so that after the assignment

$$[\![\, loc \,]\!]_{②} = [\![e]\!]$$

On the other hand, if *loc* may represent more than one memory cell, then after the update *loc* summarizes locations that contain both old and new information. Therefore, we perform a weak update, and after the assignment

$$[\![\, loc \,]\!]_{②} = [\![e]\!] \cup [\![\, loc \,]\!]_{①}$$

which is more conservative.

Suppose, however, that `x` is declared with a `restrict` qualifier:

```
void foo(int *restrict x) {... ① *x = e; ② ...}
```

Then at the beginning of `foo`, the programmer can access *loc* only through the name `x`. In other words, out of all the possible run-time locations that *loc* represents, by adding a `restrict` qualifier the programmer has specified that the body of `foo` accesses only the single location passed as the argument `x`.

Therefore within the body of `foo` we can use a fresh location loc' as our abstraction for the location `x` points to. Then at the assignment `*x = e` we can perform a strong update on loc'. When `foo` returns `*x` may again summarize many locations, so at the exit of `foo` we perform a weak update from loc' to *loc*. Thus `restrict` lets us capture a common mutual exclusion pattern (see Section 5).

2 Language and Type Checking

We present our type system for `restrict` using an extended λ-calculus. For example uses of `restrict`, see Section 5. Section 6 discusses the issues in applying these ideas to C.

We typecheck the following language:

$$\begin{array}{rcll}
e & ::= & v & \\
& \mid & \texttt{ref}\, e & \text{Allocate memory initialized to } e \\
& \mid & \texttt{*}\, e & \text{Dereference pointer } e \\
& \mid & e_1 \,\texttt{:=}\, e_2 & \text{Assign } e_2 \text{ to } e_1 \\
& \mid & e_1\ e_2 & \text{Apply function } e_1 \text{ to argument } e_2 \\
& \mid & \texttt{restrict}\, x \,\texttt{=} e_1 \,\texttt{in}\, e_2 & \\
& & & \text{Restrict } e_1 \text{ to name } x \text{ in } e_2 \\
v & ::= & x & \text{Variable} \\
& \mid & n & \text{Integer} \\
& \mid & \lambda x.e & \text{Function}
\end{array}$$

Our language contains a new scoping construct

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

with the following meaning: x, which must be a pointer, is initialized to e_1 and bound within the body e_2. Within e_2, the only access to the location x points to is through x or values derived from x.

In C terms, the expression $\texttt{restrict}\, x \,\texttt{=} e_1 \,\texttt{in}\, e_2$ is equivalent to

```
{ T *restrict x = e1;
  e2;
}
```

for some type `T`.

To enforce the semantics of `restrict`, our type system needs two non-standard features. We need to explicitly model memory locations, and we need to track which locations evaluation may access. Our type language is given by the following grammar:

$$\begin{array}{rcl}
\tau & ::= & \alpha \mid \mathit{int} \mid \mathit{ref}^{\rho}(\tau) \mid \tau_1 \xrightarrow{L} \tau_2 \\
L & ::= & \emptyset \mid \{\rho\} \mid L_1 \cup L_2 \mid L - \{\rho\}
\end{array}$$

The base types, variables α and an integer type, are standard. Pointers are given types of the form $\mathit{ref}^{\rho}(\tau)$, meaning a pointer to a value of type τ. The ρ is an *abstract location* naming the location pointed to. In alias analysis terms, ρ can be thought of as a label naming a location.

Functions are given types of the form $\tau_1 \xrightarrow{L} \tau_2$, meaning a function with domain τ_1, range τ_2, and *effect* L. Effects are sets of locations. Intuitively, a function of type $\tau_1 \xrightarrow{L} \tau_2$ may access (read or write) locations in L when executed.

Our type system proves judgments of the form

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

meaning that expression e has type τ in type environment Γ, and the evaluation of e may read or write the locations in L. We write $\mathit{locs}(\tau)$ for the set of locations occurring in the type τ, defined as

$$\begin{array}{rcl}
\mathit{locs}(\alpha) & = & \emptyset \\
\mathit{locs}(\mathit{int}) & = & \emptyset \\
\mathit{locs}(\mathit{ref}^{\rho}(\tau)) & = & \{\rho\} \cup \mathit{locs}(\tau) \\
\mathit{locs}(\tau_1 \xrightarrow{L} \tau_2) & = & \mathit{locs}(\tau_1) \cup \mathit{locs}(\tau_2) \cup L
\end{array}$$

The first case, $\mathit{locs}(\alpha) = \emptyset$, is sound for our monomorphic type system because α represents a base type. See Section 5.3 for a discussion of this case for a polymorphic type system.

We define $\mathit{locs}(\Gamma)$ as $\bigcup_{x:\tau \in \Gamma} \mathit{locs}(\tau)$. In proving judgments, our type system uses auxiliary constraints of the form $\rho \notin L$ (notice that syntactically the set $\mathit{locs}(\tau)$ can be considered an effect L), which holds if ρ does not appear in the set L.

Figure 1 gives the type checking rules for our language. We briefly discuss the rules.

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

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

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

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

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

$$\frac{\Gamma[x \mapsto \tau_1] \vdash e : \tau_2; L}{\Gamma \vdash \lambda x.e : \tau_1 \xrightarrow{L} \tau_2; \emptyset} \text{ (Lam)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \xrightarrow{L} \tau_2; L_1 \qquad \Gamma \vdash e_2 : \tau_1; L_2}{\Gamma \vdash e_1\ e_2 : \tau_2; L_1 \cup L_2 \cup L} \text{ (App)}$$

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

$$\frac{\Gamma \vdash e : \tau; L \qquad \rho_1, \ldots, \rho_n \notin \mathit{locs}(\Gamma, \tau)}{\Gamma \vdash e : \tau; L - \{\rho_1, \ldots, \rho_n\}} \text{ (Down)}$$

Figure 1: Type Checking Rules

- (Var) and (Int) are standard.
- (Ref) constructs a pointer type. Notice that ρ is unconstrained in this rule, because initialization is not an update in our semantics.
- (Deref) deconstructs a pointer type. Since operationally a dereference reads a location, we add ρ, the abstract location pointed to by e, to the effect set.
- (Assign) updates a location. As with (Deref), we add ρ to the effect set, since the assignment updates e_1. Notice we require that the type of e_2 and the type pointed to by e_1 match. Since those types may themselves contain abstract locations ρ, this rule enforces a kind of may-alias analysis in the style of Steensgaard [Ste96].
- (Lam) constructs a function type with effect L, the effect of evaluating the function body e.
- (App) deconstructs a function type. Notice that the effect of $e_1\ e_2$ includes the effect of evaluating e_1, the effect of evaluating e_2, and the effect of calling the function e_1.

The key rule in this system is (Restrict). Recall that the semantics of $\texttt{restrict}\, x \texttt{=} e_1\, \texttt{in}\, e_2$ state that during the evaluation of e_2, the object x points to may only be accessed through x. To enforce this rule with types, (Restrict) binds x to a type with a fresh abstract location ρ'. With this binding we can distinguish accesses through x (or values derived from x), which have an effect on location ρ', from accesses through aliases of x, which have an effect on location ρ. Within the scope of e_2 only accesses through x are allowed. Hence the constraint $\rho \notin L_2$.

The final hypothesis of (Restrict), $\rho' \notin locs(\Gamma, \tau, \tau_2)$, prevents x from escaping the scope of e_2. Consider the following program:

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

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

An alternative formulation is to use a flow-sensitive type system, where in (Restrict) `x` is given the type $ref^{\rho'}(\tau)$ within e_2's scope, and escaping occurrences of `x` are given the type $ref^{\rho}(\tau)$ after the `restrict`. We did not pursue this strategy because it is significantly more complex, and to be useful `restrict` must be easily understood by programmers.

Finally, notice that the conclusion of (Restrict) contains the effect $\{\rho\}$, i.e., restricting a location is itself an effect. This naturally forbids the following program:

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

The last rule, (Down), states that effects on any non-escaping location can be removed from the effect set [GJLS87, LG88, Cal01]. The rule (Down) is the one non-syntactic rule in our system. We can construct a purely syntax-directed version of our system by incorporating down into the type rules.

Lemma 1 *A proof of $\Gamma \vdash e : \tau; L$ can be rewritten to contain at most one occurrence of (Down) in sequence.*

Lemma 2 *A proof of $\Gamma \vdash e : \tau; L$ can be rewritten so that the only uses of (Down) are as the final step in the proof, the hypothesis of (Lam), or the e_2 hypothesis of (Restrict).*

Proof: All rules except (Down), (Lam), and (Restrict) are monotonic in their effects. That is, for each rule except (Down) and (Lam) the set of effects in the conclusion is a superset of all the sets of effects in the hypotheses. Now, for any effect sets L and L' we have

$$(L - \{\rho_1, ..., \rho_n\}) \cup L' = (L \cup L') - \{\rho_{i_1}, ..., \rho_{i_m}\}$$

where

$$\{\rho_{i_1}, ..., \rho_{i_m}\} = \{\rho_1, ..., \rho_n\} \cap \neg L'$$

Thus we can always move a use of (Down) above one of the hypotheses below the conclusion.

In (Restrict), we can move uses of (Down) from above the e_1 hypothesis to below the conclusion. We cannot move arbitrary uses of (Down) from above the e_2 hypothesis because of the constraint $\rho \notin L_2$. □

$$\frac{l \in dom(S)}{S \vdash l \to l; S} \text{ [Var/Loc]}$$

$$\frac{}{S \vdash n \to n; S} \text{ [Int]}$$

$$\frac{S \vdash e \to v; S' \quad l \notin dom(S')}{S \vdash \texttt{ref}\, e \to l; S'[l \mapsto v]} \text{ [Ref]}$$

$$\frac{S \vdash e \to l; S' \quad l \in dom(S')}{S \vdash *e \to S'(l); S'} \text{ [Deref]}$$

$$\frac{\begin{array}{cc} S \vdash e_1 \to l; S' & S' \vdash e_2 \to v; S'' \\ l \in dom(S'') & S''(l) \neq \mathbf{err} \end{array}}{S \vdash e_1 \texttt{ := } e_2 \to v; S''[l \mapsto v]} \text{ [Assign]}$$

$$\frac{}{S \vdash \lambda x.e \to \lambda x.e; S} \text{ [Lam]}$$

$$\frac{\begin{array}{c} S \vdash e_1 \to \lambda x.e; S' \quad S' \vdash e_2 \to v; S'' \\ S'' \vdash e[x \mapsto v] \to v'; S''' \end{array}}{S \vdash e_1\ e_2 \to v'; S'''} \text{ [App]}$$

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

Figure 2: Big-Step Semantics

Based on these two lemmas, we can eliminate (Down) and incorporate it into (Lam) and (Restrict):

$$\frac{\begin{array}{c} \Gamma[x \mapsto \tau_1] \vdash e : \tau_2; L \\ \rho_1, \ldots, \rho_n \notin locs(\Gamma, \tau_1, \tau_2) \\ L' = L - \{\rho_1, \ldots, \rho_n\} \end{array}}{\Gamma \vdash \lambda x.e : \tau_1 \xrightarrow{L'} \tau_2; \emptyset} \text{ (Lam}'\text{)}$$

$$\frac{\begin{array}{c} \Gamma \vdash e_1 : ref^{\rho}(\tau); L_1 \\ \Gamma[x \mapsto ref^{\rho'}(\tau)] \vdash e_2 : \tau_2; L_2 \\ \rho_1, \ldots, \rho_n \notin locs(\Gamma, \tau, \tau_2) \cup \{\rho'\} \\ L_2' = L_2 - \{\rho_1, \ldots, \rho_n\} \\ \rho \notin L_2' \qquad \rho' \notin locs(\Gamma, \tau, \tau_2) \end{array}}{\Gamma \vdash \texttt{restrict}\, x \texttt{=} e_1\, \texttt{in}\, e_2 : \tau_2; L_1 \cup L_2' \cup \{\rho\}} \text{ (Restrict}'\text{)}$$

3 Semantics and Soundness

Figure 2 gives a big-step operational semantics for our language. Judgments are of the form $S \vdash e \to v; S'$, meaning that given store S (a map from locations to values), expression e evaluates to value v and new store S'. Values are locations, integers, and functions. We implicitly assume that if $S \vdash e \to v; S'$ is not provable by the rules then $S \vdash e \to \mathbf{err}; S$. The token **err** is not a value.

Notice that our semantics contains no environment for variables. We use substitution to bind variables to values. Locations are bound in the store to either values or **err**.

We only discuss two interesting features of the rules. First, [Assign] has a surprising case: We need to check whether $S''(l)$ is **err** before updating location l. We do not need to check for this explicitly in [Deref], because the semantics are strict in **err**. However, because [Assign] does not examine the contents of l, we need to add this check.

Second, [Restrict] uses copying to enforce `restrict`'s semantics. To evaluate `restrict` x `=` e_1 `in` e_2, we first evaluate e_1 normally, which must yield a pointer l. Within the body of e_2, the only way to access what l points to should be via the particular value that resulted from evaluating e_1. We enforce this by allocating a fresh location l' initialized with the contents of l, and then binding l to **err** to forbid access through l. Recall that because our semantics is strict in **err**, any program that dereferences l within e_2 will result in **err**.

The soundness of our type system (see below) implies that no program evaluates to **err**, which in turn implies that an implementation can safely optimize `restrict` by eliding the copy of l. Instead, in an implementation `restrict` simply binds x to l.

Notice it is not an error to use the value l, but only to dereference it. When e_2 has been evaluated we re-initialize l to point to the value `x` points to, and then forbid accesses through l'. Forbidding access through l' corresponds to the requirement in the type rule (Restrict) that ρ' not escape. An alternative formulation is to rename occurrences of l' to l after e_2 finishes. We could write this style of rule in a small-step semantics.

3.1 Soundness

We next sketch a proof of soundness. In our proof, we implicitly extend typing judgments to semantic values in the following way. Locations l are represented in the proof as program variables, and thus their types are stored in Γ and they typecheck using (Var). We implicitly treat evaluated and unevaluated integers identically and use (Int) to typecheck both. Functions are represented not as closures but as syntactic functions, as in standard small-step semantics subject-reduction proofs [WF94, EST95]. Thus evaluated functions are typechecked using (Lam).

To show soundness we first show a subject-reduction result. We begin by introducing a notion of compatibility to capture when it is safe to evaluate an expression.

Definition 1 (Compatibility) *We say Γ and L are* compatible *with store S, written $(\Gamma, L) \sim S$, if*

1. *$dom(\Gamma) = dom(S)$ and*
2. *for all $l \in dom(S)$, there exist ρ, τ such that $\Gamma(l) = ref^{\rho}(\tau)$ and*

$$\begin{cases} \Gamma \vdash S(l) : \tau; \emptyset & \text{if } S(l) \neq \textbf{\textit{err}} \\ \rho \notin L & \text{if } S(l) = \textbf{\textit{err}} \end{cases}$$

Intuitively, $(\Gamma, L) \sim S$ if an expression e that typechecks in environment Γ and has effect L can execute safely in store S. Notice that the definition of compatibility requires $dom(\Gamma) = dom(S)$, i.e., that expressions typed in environment Γ may contain locations but not other free variables. This property is maintained during evaluation because in [App] we implement function call with substitution.

As evaluation progresses the proof of subject reduction extends Γ with new locations allocated by `ref` expressions. It is a property of our semantics and type system that these additions to Γ are safe, in the following sense:

Definition 2 (Safe Extension) *We say that (Γ', S') is a* safe extension *of (Γ, S), written $(\Gamma, S) \Rightarrow (\Gamma', S')$, if*

1. *$dom(\Gamma) = dom(S)$ and $dom(\Gamma') = dom(S')$,*
2. *$\Gamma'|_{dom(\Gamma)} = \Gamma$,*
3. *for all $l \in dom(S') - dom(S)$, if $S'(l) = \textbf{\textit{err}}$ and $\Gamma'(l) = ref^{\rho}(\tau)$, then $\rho \notin locs(\Gamma)$, and*

4. *for all $l \in dom(S)$, if $S'(l) = \mathbf{err}$ then $S(l) = \mathbf{err}$.*

Here $\Gamma'|_{dom(\Gamma)}(x)$ is the restriction of Γ' to the domain of Γ. Intuitively, $(\Gamma, S) \Rightarrow (\Gamma', S')$ if the **err**-bound locations in S' are either also **err**-bound in S, or if they are fresh (do not appear in Γ).

With these definitions we can state our subject-reduction theorem. We use r to stand for a semantic reduction result, either a value v or **err**.

Theorem 1 (Subject Reduction) *If $\Gamma \vdash e : \tau; L$ and $S \vdash e \rightarrow r; S'$, where $(\Gamma, L \cup L') \sim S$ for some L', then there exists Γ' such that*

1. *$\Gamma' \vdash r : \tau; \emptyset$ (which implies $r \neq \mathbf{err}$),*

2. *$(\Gamma', L') \sim S'$, and*

3. *$(\Gamma, S) \Rightarrow (\Gamma', S')$*

Proof (Sketch): By induction on the structure of the derivation $S \vdash e \rightarrow r; S'$. The interesting case is $\texttt{restrict}\, x \,\texttt{=} e_1 \,\texttt{in}\, e_2$.

By assumption, we know

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

We also have $S \vdash \texttt{restrict}\, x \,\texttt{=} e_1 \,\texttt{in}\, e_2 \rightarrow r; S'$. By inspection of the semantic rules, this reduction must have contained a reduction of e_1:

$$S \vdash e_1 \rightarrow r_{e_1}; S'_{e_1}$$

We apply induction to show that there exists a Γ'_{e_1} satisfying

1. $\Gamma'_{e_1} \vdash r_{e_1} : \mathit{ref}^{\rho}(\tau); \emptyset$

2. $(\Gamma'_{e_1}, L_2 \cup \{\rho\} \cup L') \sim S'_{e_1}$

3. $(\Gamma, S) \Rightarrow (\Gamma'_{e_1}, S'_{e_1})$

After this step, though, we cannot directly apply induction to the evaluation

$$S'_{e_1}[r_{e_1} \mapsto \mathbf{err}, l' \mapsto S'_{e_1}(r_{e_1})] \vdash e_2[x \mapsto l'] \rightarrow r_{e_2}; S'_{e_2}$$

of e_2. The problem is that we would need to show the following compatibility:

$$(\Gamma'_{e_1}, L_2 \cup \{\rho\} \cup L') \sim S'_{e_1}[r_{e_1} \mapsto \mathbf{err}, l' \mapsto S'_{e_1}(r_{e_1})]$$

But of course this compatibility does not hold, because r_{e_1} maps to **err**. We can solve this problem by simply removing ρ from the effect set. But there is a deeper problem: although l' is fresh, ρ' may not be, and thus there may be some location l'' such that $\Gamma'_{e_1}(l'') = \mathit{ref}^{\rho'}(\tau)$ and $S'_{e_1}(l'') = \mathbf{err}$.

To solve this problem, we observe that the name ρ' is arbitrary. We construct a substitution $R = [\rho' \mapsto \rho'']$ for some fresh ρ''. Then from

$$\Gamma[x \mapsto \mathit{ref}^{\rho'}(\tau)] \vdash e_2 : \tau_2; L_2$$

we conclude

$$R(\Gamma[x \mapsto \mathit{ref}^{\rho'}(\tau)]) \vdash e_2 : R(\tau_2); R(L_2)$$

Then using the hypothesis $\rho' \notin \mathit{locs}(\Gamma, \tau, \tau_2)$ in the type rule (Restrict), we derive

$$\Gamma[x \mapsto \mathit{ref}^{\rho''}(\tau)] \vdash e_2 : \tau_2; R(L_2)$$

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

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

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

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

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

$$\frac{\begin{array}{c}\Gamma[x \mapsto \alpha], \varepsilon_{\Gamma[x \mapsto \alpha]} \vdash e : \tau_2; L \qquad \alpha, \varepsilon \text{ fresh} \\ \varepsilon_\Gamma \cup \mathit{locs}(\alpha) \subseteq \varepsilon_{\Gamma[x \mapsto \alpha]} \\ L \cap (\varepsilon_{\Gamma[x \mapsto \alpha]} \cup \mathit{locs}(\tau_2)) \subseteq \varepsilon\end{array}}{\Gamma, \varepsilon_\Gamma \vdash \lambda x.e : \alpha \xrightarrow{\varepsilon} \tau_2; \emptyset} \text{(Lam)}$$

$$\frac{\begin{array}{cc}\Gamma, \varepsilon_\Gamma \vdash e_1 : \tau_1; L_1 & \Gamma, \varepsilon_\Gamma \vdash e_2 : \tau_2; L_2 \\ \tau_1 = \tau_2 \xrightarrow{\varepsilon} \alpha & \alpha, \varepsilon \text{ fresh}\end{array}}{\Gamma, \varepsilon_\Gamma \vdash e_1\ e_2 : \tau_2; L_1 \cup L_2 \cup \varepsilon} \text{(App)}$$

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

Figure 3: Type Inference Rules

Now we can show the following compatibility:

$$(\Gamma'_{e_1}, R(L_2) \cup (L' - \{\rho\})) \sim S'_{e_1}[r_{e_1} \mapsto \mathbf{err}, l' \mapsto S'_{e_1}(r_{e_1})]$$

and thus apply induction to the evaluation of e_2.

We apply the same renaming trick to uses of (Down). Because $\rho_1, \ldots, \rho_n$ do not escape in (Down), their names are arbitrary and we can freely rename them. □

Given the subject-reduction theorem, soundness is easy to show:

Corollary 1 *If $\emptyset \vdash e : \tau; L$, then $\emptyset \vdash e \to r; S'$, where r is not* ***err****.*

Proof: First observe that $(\emptyset, L) \sim \emptyset$. Then in our semantics every term can be reduced to some result r. By the subject-reduction theorem, there is a Γ' such that $\Gamma' \vdash r : \tau; \emptyset$. Thus r is not **err**. □

4 Inference

Figure 3 gives the type inference rules for our system. Our type rules generate three kinds of constraints C, equality constraints between types, inclusion constraints between effects, and disinclusion constraints between locations and effects:

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

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

In addition to the type environment Γ, during type inference we construct effect variables ε_Γ to contain the set of effects occurring in environment Γ. As we extend environment Γ with a new binding $x \mapsto \tau$ in (Lam) and (Restrict), we generate a constraint

$$\varepsilon_\Gamma \cup \mathit{locs}(\tau) \subseteq \varepsilon_{\Gamma[x \mapsto \tau]}$$

In this way we succinctly capture the set $\mathit{locs}(\Gamma)$ without needing an explicit linear pass through Γ. We briefly discuss the rules.

- (Var) and (Int) are identical to the type checking rules except for the addition of ε_Γ to the left of the turnstile.
- (Ref), (Deref), (Assign), and (App) are written with explicit fresh variables and equality constraints between types.
- (Lam) incorporates (Down) according to Lemmas 1 and 2. Notice that we express (Down) as an intersection property. Also notice that function types always have an effect variable ε on the arrow. In this way when we propagate equality constraints on types to equality constraints on effects, we will always yield equalities between effect variables rather than between arbitrary effects.
- (Restrict) also incorporates (Down), written as an intersection property. (Restrict) is the only rule that generates constraints of the form $\rho \notin L$.

Let m be the size of the initial program. Applying the type inference rules in Figure 3 takes $O(m)$ time and generates a system of constraints C of size $O(m)$.

We split the resolution of the side constraints C into two phases. First, we apply unification to solve the type equality constraints $\tau_1 = \tau_2$. Figure 4a gives the type unification algorithm as a series of left-to-right rewrite rules. This step can be completed in $O(m\alpha(m))$ time, where $\alpha(\cdot)$ is the inverse Ackerman's Function.

After this first step, we replace the sets $\mathit{locs}(\tau)$ by the set of locations contained in τ (defined in Section 2). Let n be the size of the program annotated with *standard* types. Notice that substituting for the sets $\mathit{locs}(\tau)$ takes time $O(n)$, and the resulting constraint system is size $O(n)$, because of our restriction that arrows are always annotated with effect variables.

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

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

Lemma 3 *If effect constraint system C has a solution, then C has a least solution.*

Proof: Let Σ be any set of solutions of C. We claim that $\bigcap_{\sigma \in \Sigma} \sigma$ is also a solution of C, where $(\bigcap_{\sigma \in \Sigma} \sigma)(\varepsilon) = \bigcap_{\sigma \in \Sigma} \sigma(\varepsilon)$. Suppose C contains a constraint $\rho \notin L$. Then by assumption for all $\sigma \in \Sigma$ we have $\rho \notin \sigma(L)$. Thus it must be that $\rho \notin \bigcap_{\sigma \in \Sigma} \sigma(L)$. Suppose C contains a constraint $L \subseteq \varepsilon$. Then by assumption for all $\sigma \in \Sigma$ we have $\sigma(L) \subseteq \sigma(\varepsilon)$. But then $(\bigcap_{\sigma \in \Sigma} \sigma(L)) \subseteq (\bigcap_{\sigma \in \Sigma} \sigma(\varepsilon))$.

Then since $(\bigcap_{\sigma \in \Sigma} \sigma) \leq \sigma$ for all $\sigma \in \Sigma$, the set of solutions of C has a least element. $\square$

$$
\begin{array}{rcl}
C \cup \{\alpha = \tau\} & \Rightarrow & C[\alpha \mapsto \tau] \\
C \cup \{\mathit{int} = \mathit{int}\} & \Rightarrow & C \\
C \cup \{\mathit{ref}^{\rho_1}(\tau_1) = \mathit{ref}^{\rho_2}(\tau_2)\} & \Rightarrow & \\
 & & C \cup \{\rho_1 = \rho_2\} \cup \{\tau_1 = \tau_2\} \\
C \cup \{\tau_1 \xrightarrow{\varepsilon} \tau_2 = \tau_1' \xrightarrow{\varepsilon'} \tau_2'\} & \Rightarrow & \\
 & & C \cup \{\tau_1 = \tau_1'\} \cup \{\tau_2 = \tau_2'\} \cup \{\varepsilon = \varepsilon'\} \\
C \cup \{\text{other type eqn}\} & \Rightarrow & \text{unsatisfiable} \\
C \cup \{\rho_1 = \rho_2\} & \Rightarrow & C[\rho_1 \mapsto \rho_2] \\
C \cup \{\varepsilon_1 = \varepsilon_2\} & \Rightarrow & C[\varepsilon_1 \mapsto \varepsilon_2]
\end{array}
$$

(a) Type Unification

$$
\begin{array}{rcl}
C \cup \{\rho \not\in L\} & \Rightarrow & \\
 & & C \cup \{\rho \not\in \varepsilon\} \cup \{L \subseteq \varepsilon\} \quad \varepsilon \text{ fresh} \\
C \cup \{\emptyset \subseteq \varepsilon\} & \Rightarrow & C \\
C \cup \{L_1 \cup L_2 \subseteq \varepsilon\} & \Rightarrow & C \cup \{L_1 \subseteq \varepsilon\} \cup \{L_2 \subseteq \varepsilon\} \\
C \cup \{\emptyset \cap L \subseteq \varepsilon\} & \Rightarrow & C \\
C \cup \{L \cap \emptyset \subseteq \varepsilon\} & \Rightarrow & C \\
C \cup \{(L_1 \cup L_2) \cap L \subseteq \varepsilon\} & \Rightarrow & \\
 & & C \cup \{\varepsilon' \cap L \subseteq \varepsilon\} \cup \{L_1 \cup L_2 \subseteq \varepsilon'\} \quad \varepsilon' \text{ fresh} \\
C \cup \{L \cap (L_1 \cup L_2) \subseteq \varepsilon\} & \Rightarrow & \\
 & & C \cup \{L \cap \varepsilon' \subseteq \varepsilon\} \cup \{L_1 \cup L_2 \subseteq \varepsilon'\} \quad \varepsilon' \text{ fresh}
\end{array}
$$

(b) Constraint Normalization

Figure 4: Constraint Resolution

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

$$
\begin{array}{rcl}
C & ::= & L \subseteq \varepsilon \mid \rho \not\in \varepsilon \\
M & ::= & \{\rho\} \mid \varepsilon \\
L & ::= & M \mid M \cap M
\end{array}
$$

Notice that the rules in Figure 4b preserve least solutions but not arbitrary solutions. Also notice that in Figure 4b we do not consider the case $(L_1 \cap L_2) \cap L \subseteq \varepsilon$ or $L \cap (L_1 \cap L_2) \subseteq \varepsilon$. Inspection of the rules of Figure 3 shows that constraints with nested intersections are never generated. Applying the rules in Figure 4b takes time $O(n)$.

We view the inclusion constraints in a normal form effect constraint system as a directed graph:

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

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

Given a normal form effect constraint system, we test satisfiability by checking, for each constraint $\rho \not\in \varepsilon$, whether $\rho \in S(\varepsilon)$ in the least solution S. Figure 5 shows the modified depth-first search we use to check this

```
Associate a count $C(v)$ with each node $v$ in the graph
Initialize $C(v) = 0$ for all $v$
Let $W = \{\rho\}$, the set of nodes left to visit
While $W$ is not empty
  Remove some node $v$ from $W$
  If $v == \varepsilon$ return **unsatisfiable**
  For each edge $v \to \varepsilon'$
    If $C(\varepsilon') == 0$ then
      $C(\varepsilon') = 1$
      Add $\varepsilon'$ to $W$
  For each edge $v \to \cap$
    If $C(\cap) == 0$ then
      $C(\cap) = 1$
    If $C(\cap) == 1$ then
      $C(\cap) = 2$
      Add $\cap$ to $W$
Return **satisfiable**
```

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

condition. The algorithm in Figure 5 takes time $O(n)$ for each $\rho \notin \varepsilon$ constraint. Given an initial program with k occurrences of `restrict`, the system considered in Figure 5 will have $O(k)$ constraints of the form $\rho \notin \varepsilon$. Hence the total time for this step is $O(kn)$.

Therefore the total time for this algorithm is $O(m\alpha(m) + kn)$. Given that the typed program is strictly larger than the untyped program, the algorithm runs in time $O(n\alpha(n) + kn)$, where n is the size of the program annotated with standard types.

5 Examples and Extensions

We present some small examples that illustrate interesting features of our type system for `restrict`. We write our examples in C, and unless otherwise specified the behavior of each usage in our type system matches C99.

The obvious application of `restrict` is to control the name through which a location is accessed. For example, consider the following code:

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

Since `p` is annotated with `restrict`, we may dereference `p` but we may not dereference `q`.

As another example, consider the following code, which shows how `restrict`-qualified pointers may be passed from outer to inner scopes:

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

Here within the scope of `r` we can dereference `r` but not `p`, and when `r` goes out of scope we recover the ability to dereference `p`.

As another example, consider

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

Notice that binding p to r actually writes the value of p to memory (r is an l-value). In this case, within the scope of p our type system allows us to store and retrieve the value of p from memory as long as that memory does not escape the scope of p. This particular example is not allowed by the C standard.

The rule (Down) allows us to distinguish restricted locations in different instances of the same function. For example, consider the following code:

```
void foo(void) {
  int a;
  int *restrict p = &a;
  foo();    // valid
  *p = 3;   // valid
}
```

Because a is purely local to the body of foo(), applying (Down) removes the effect on a from the effect set of foo(). Thus foo() does not have any visible effect, and calling foo() within the scope of p is valid.

One of the most interesting properties of restrict is that it lets us recover non-aliasing information from complicated aliasing conditions. For example, consider the following code:

```
{ lock *restrict l = a[i]->lock;
  spin_lock(l);
  ...
  spin_unlock(l);
}
```

In a standard type system all elements of an array have the same type. Thus in this example the pointer l may point to a large set of possible locations. Recall from the introduction that restrict gives us the ability to temporarily recover strong updates. Thus even though l may point to a large number of possible locks, by annotating the type of l with restrict the programmer has guaranteed that they only access one particular lock within the scope of l. Thus using a standard data-flow analysis we can treat the calls to spin_lock and spin_unlock as strong updates and successfully check whether this piece of code adheres to a standard locking protocol (e.g., spin_lock is never called twice in a row on the same lock).

5.1 affects Clauses

Our notation includes effect sets L on function types $\tau_1 \xrightarrow{L} \tau_2$. Clearly if we wish to apply our type system to real source code, we need some notation for annotating function types with these effect sets.

One natural design is to use syntax similar to Java exceptions. We allow functions to be declared with affects clauses of the form

```
void foo(int **x) affects *x, **x;
```

In our type system, x is given a type of the form

$$ref^{\rho}(ref^{\rho'}(int))$$

The affects clause means that foo() should be given the type

$$ref^{\rho}(ref^{\rho'}(int)) \xrightarrow{\{\rho,\rho'\}} \texttt{void}$$

The type checker incorporates this notation by using the declared effect set of foo wherever foo is used. When the typechecker types the body of foo and computes its effect L, it verifies that $L = \{\rho, \rho'\}$, i.e., that the declared type was valid.

5.2 Subtyping

We can extend our type and effect system to admit a form of subtyping among function types [GJLS87, TJ95]. Given our type system, it is natural to add a rule of the form

$$\frac{\tau_1' \leq \tau_1 \qquad \tau_2 \leq \tau_2' \qquad L \subseteq L'}{\tau_1 \xrightarrow{L} \tau_2 \leq \tau_1' \xrightarrow{L'} \tau_2'}$$

and to add subsumption

$$\frac{\Gamma \vdash e : \tau; L \qquad \tau \leq \tau'}{\Gamma \vdash e : \tau'} \;\text{(Sub)}$$

to the typing rules.

Subtyping increases the usefulness of the type system in two ways. First, more programs will typecheck with subtyping. Second, subtyping makes `affects` clauses easier to write. Since the programmer is likely not to annotate functions with exactly the right set of effects, if we incorporate subtyping then instead of requiring that the declared effect L' or a function match exactly the inferred effect L, we can require $L' \supseteq L$. Our subject reduction and soundness theorems from Section 3 both hold in a system with (Sub). Our inference algorithm can also be easily adapted to admit subsumption.

5.3 Parametric Polymorphism

Our type and effect system can also be extended with parametric polymorphism. Function types are extended to *polymorphically constrained types*

$$\tau ::= \cdots \mid \forall \vec{\rho}\vec{\varepsilon}.\tau_1 \xrightarrow{L} \tau_2 \backslash C$$

Here $\vec{\rho}$ are the generalized abstract locations, $\vec{\varepsilon}$ are the generalized effect variables, and C is a collection of constraints of the form $\rho \notin L$, $\rho \notin \mathit{locs}(\Gamma)$, and $\rho \notin \mathit{locs}(\tau)$. The type rules are rewritten to track the constraints generated in each sub-proof, e.g.,

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

We add the standard rules for quantification and instantiation:

$$\frac{\Gamma, C \vdash e : \tau; L \qquad \vec{\rho}, \vec{\varepsilon} \notin \mathit{locs}(\Gamma)}{\Gamma, C \vdash e : \forall \vec{\rho}\vec{\varepsilon}.\tau \backslash C} \;\text{(Poly)}$$

$$\frac{\begin{array}{c}\Gamma, C \vdash x : \forall \vec{\rho}\vec{\varepsilon}.\tau \backslash C' \qquad C \vdash RC' \\ R = [\vec{\rho} \mapsto \vec{\rho'}, \vec{\varepsilon} \mapsto \vec{\varepsilon'}]\end{array}}{\Gamma, C \vdash x : R\tau} \;\text{(Inst)}$$

We restrict generalization to function types, as is standard [Wri95].

If we also allow type polymorphism in addition to location and effect polymorphism, then in (Restrict) we may need to check constraints of the form $\rho \notin \mathit{locs}(\alpha)$ for some type variable α generalized in (Poly). In this case we cannot define $\mathit{locs}(\alpha) = \emptyset$, since α may be instantiated with a *ref* type at some point. Instead, we carry the sets $\mathit{locs}(\alpha)$ syntactically in the polymorphically constrained types and expand them when the type variables α are instantiated.

We believe our soundness proof can be easily adapted to admit parametric polymorphism, as can our inference algorithm if we restrict ourselves to Hindley-Milner style let-polymorphism. We leave polymorphic recursive type inference in the style of [TT94, RF01] as an open problem.

6 ANSI C

In this section we discuss some of the issues in checking `restrict` in C99. `restrict` is useful in any language with updateable references, not just C, and that several of the issues in checking `restrict` in C99 have more to do with particular features of C99 than with programming language fundamentals.

Names. In C, almost all names refer to *l*-values, that is, *ref*-types in our notation. That means, for example, if we declare

```
int *restrict p = ...;
```

then `p` may change during evaluation, which could change what object `p` points to. (Recall that in our λ-calculus notation, the name x in $\texttt{restrict}\, x = e_1 \,\texttt{in}\, e_2$ is an r-value.) The standard is imprecise on this issue, suggesting that while it is invalid to update `p` to point to a different `restrict`ed value, it may be permissible to update `p` to a different but non-`restrict`ed value.

There are several solutions to this problem. The simplest solution is to require that all `restrict`-qualified pointers also be annotated with `const`, so that they cannot change. An alternative is to distinguish between reading and writing an abstract location ρ in our effect sets, and then prevent the user from writing to the location ρ corresponding to the l-value of p with constraints of the form $wr(\rho) \notin L$, where $wr(\rho)$ is the effect of writing to location ρ. A third solution, and the one most likely taken in a C compiler, is to perform a flow-sensitive analysis limited to a single function body to determine whether `p` is updated to point within the same object or whether it is updated with a new pointer (e.g., `p++` versus `p = q`). The former should be allowed, and the latter should be forbidden.

Initialization. Our λ-calculus syntax for restrict forces the user to initialize `restrict`ed pointers as soon as they are declared. C99 has no such requirement. However, we feel that requiring `restrict`ed pointers to be initialized is not much of a burden, because the common case for `restrict` is for function parameters, and function parameters are always initialized.

Over-Estimation of Effects. The C99 standard defines `restrict` in a completely dynamic fashion. The accesses to object `X` within a block `B` are those that occur at run-time when `B` is executed. Since our analysis is static, we may over-estimate the set of locations accessed during evaluation, and hence we may fail to type check a program that executes correctly according to the standard.

Arrays. The C99 standard contains the following example of a valid use of `restrict` ([ANS99], page 111):

```
void f(int n, int *restrict p, int *restrict q)
{
  while (n-- > 0)
    *p++ = *q++;
}
void g(void) {
  extern int d[100];
  f(50, d + 50, d); // ok
}
```

In this example, the user has implicitly split the array `d` into two disjoint smaller arrays, and then called `f` knowing that as `f` traverses the arrays `p` and `q` it will only access the first 50 elements of each. In our type system this program will fail to type check, because this property—accessing only 50 elements of each array—cannot be deduced from the type of `f`. This application of `restrict` is useful in C, and must be allowed. We feel that the best way to handle this situation in a manner consistent with C is to force the programmer to insert a type cast at the call to `f()` to tell the compiler that `d[0..49]` and `d[50..99]` should be treated as distinct objects.

Escaping pointers. The C99 standard explicitly allows certain pointers annotated with `restrict` to escape the scope of their declaration. Specifically, a function whose body declares a `restrict`ed pointer `p` may return the value of `p`. The only example of this in the standard is the definition of a function that returns a pointer to a structure, one of whose fields is annotated with `restrict`. Thus the motivation for allowing escaping `restrict`ed pointers seems to be to handle this case of `restrict` in a structure declaration. We believe (see below) that annotating structure field names with `restrict` is not well-defined in general. Thus we do not support this usage, and our type system forbids `restrict`ed pointers from escaping entirely.

Data Structures. The C99 standard contains an example in which a `struct` (a record type) contains a `restrict` qualifier ([ANS99], page 112):

```
typedef struct {int n; float *restrict v;} vector;
```

This particular use of `restrict` is easy to handle in our system and semantically well-defined, since `vector` is a shorthand for a pair, and thus we can think of all operations on `vector` as syntactic sugar for operations on the individual elements of the pair. Thus we can rewrite

```
vector x = { 3, a };
... x.n ... x.v ...
```

as

```
int x_n = 3;
float *restrict x_v = a;
... x_n ... x_v ...
```

On the other hand, uses of `restrict` in defining recursive data structures are problematic. For example, consider

```
struct list {int x; struct list *restrict next; };
```

What does `restrict` mean here? The problem is that the name `next` refers to a set of (possibly distinct) objects rather than a single object in memory. For example, if we construct a circular list `p`, then `p->next` and `p->next->next` may be the same. Is that forbidden by the `restrict` annotation? It seems not, because both accesses go through the name `next`. Clearly, though, a compiler cannot use the name `next` to infer anything about potential aliasing of list elements.

A compiler needs stronger information than `restrict` to infer non-aliasing of heap objects. One such property is linearity or uniqueness [TWM95, BS96]. A *unique* object has at most one pointer to it at any time. Thus if the `next` field of `struct list` were annotated as being unique, then a compiler could assume (and enforce) that each element of `p` is distinct.

Modified Objects. The definition of `restrict` given in the introduction of this paper is slightly simpler than the standard's definition. Suppose that `p` is declared `int *restrict p` and that `p` points to object `X`. Then the standard states that the `restrict` qualifier is only meaningful if `X` is modified within the scope of `p`. We refer to this as the *mod semantics* of `restrict`.

We consider the mod semantics of `restrict` unnecessarily complicated. The main reason we see to have the mod semantics is that for optimization purposes there is no benefit to `restrict` for locations that are not modified—optimizations must preserve read-write and write-write dependencies, but read-read dependencies can be safely ignored.

However, from a language design point of view, it is undesirable to have a construct that is sometimes ignored. This is especially a problem if we think of `restrict` as an annotation to aid the programmer. For example, suppose we have the declaration `f(int *restrict a, int *restrict b)`. Is it safe to call `f(x,x)`? Under the mod semantics we cannot tell without either an `affects` clause (see Section 5.1) or examining the source code for `f`. We believe that rather than use the mod semantics, we should use our semantics and simply remove `restrict` qualifiers when they are unnecessary.

However, it is relatively easy to incorporate the mod semantics of `restrict` into our system. Rather than having effects of the form ρ, we split effects into three kinds: $rd(\rho)$, generated in (Deref), $wr(\rho)$, generated in (Assign), and $rstr(\rho)$, generated in (Restrict). Then we split the type rule (Restrict) into two cases using conditional constraints:

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

Here $\rho \notin L$ is shorthand for $rd(\rho) \notin L$, $wr(\rho) \notin L$, and $rstr(\rho) \notin L$, and similarly for $\rho \notin locs(\tau)$ and $\rho \notin locs(\Gamma)$.

In (Restrict$''$) we use a conditional constraint to enforce the mod semantics. The constraint $wr(\rho') \in L_2 \Rightarrow \rho \notin L_2$ is satisfiable if either $wr(\rho') \notin L_2$ or $\rho \notin L_2$. In other words, we only require that ρ is not accessed in e_2 if ρ' is modified.

Notice that in (Restrict$''$) we still require that ρ' not escape e_2. As before this forbids having different, non-restricted abstract locations corresponding to the same concrete location. Finally, notice that it is an error if ρ is modified in e_2 (formally $wr(\rho) \in L_2$). This is because if ρ is modified in e_2, then in the mod semantics we also require $\rho \notin L_2$, which is an immediate contradiction.

For type inference we proceed as before. We first infer ρ annotations. Next we normalize the constraints, with the addition of replacing the constraints $wr(\rho') \in L \Rightarrow \rho \notin L$ with $wr(\rho') \in \varepsilon \Rightarrow \rho \notin \varepsilon$ and $L \subseteq \varepsilon$.

Finally, after checking satisfiability of the $\rho \notin \varepsilon$ constraints, we check the conditional constraints. For each constraint $wr(\rho') \in \varepsilon \Rightarrow \rho \notin \varepsilon$, we use the modified depth-first search procedure of Figure 5 to check whether $wr(\rho')$ reaches ε. If $wr(\rho')$ does not reach ε, then the constraint is satisfiable. If $wr(\rho')$ does reach ε, then the constraint is satisfiable if and only if $\rho \notin \varepsilon$, which we check with another call to the depth-first search procedure. Since there are $O(k)$ conditional constraints, where k is the number of `restrict` annotations in the program, this step takes time $O(kn)$, where n is the size of the typed program. Thus the whole inference algorithm still takes time $O(kn + n\alpha(n))$.

7 Related Work

Effect systems were first described by Gifford and Lucassen for FX-87 [GJLS87, LG88]. FX-87 includes subtyping, polymorphism, and notation for declaring the effects of expressions [GJLS87]. One of the best-known type and effect systems is the region type system proposed by Tofte and Talpin [TT94]. Our type system bears some interesting similarities to region inference. The ρ annotations can be thought of as regions, and we can apply the rule (Down) whenever we discover that a location is purely local to a lexical scope of the computation [Cal01].

As mentioned in Section 5.3, we leave as an open problem inference for our type system with polymorphic recursion. Given the similarities between our type system and Tofte and Talpin's region type system, we believe that Tofte and Birkedal's region inference algorithm [TB98] can be adapted to our type system.

Automatic alias analysis has been heavily studied in recent years [And94, BCCH94, CRL99, Das00, DMW98, Deu94, DRS98, EGH94, FRD00, HT01, HP98, LR92, SRW99, SH97, Ste96, WL95, YHR99, ZRL96]. Our type system incorporates may-alias analysis to check the correctness of `restrict` annotations. The may-alias analysis we use is very conservative, and in the future we plan to extend our type system to use more precise may-alias analysis.

The key benefit of our type system compared with a fully-automatic alias analysis is that the programmer has control over the alias analysis and, by extension, any subsequent analyses (e.g., optimizations) based on aliasing information. If the type system rejects a `restrict` annotation as being inconsistent, the programmer knows that either their annotation is incorrect or the underlying may-alias analysis is overly conservative, and they can take the appropriate steps to correct the problem. Once a program with `restrict` annotations passes the type checker, the programmer knows at least some of the places where the compiler assumes non-aliasing. We feel that this exposure of aliasing information is helpful to a programmers.

As described in the introduction, one of the most interesting properties of `restrict` is that it lets us locally recover the ability to perform strong updates [CWZ90]. We believe that `restrict` can be added to extend the flow-sensitive type systems of Walker, Smith, and Morrisett [SWM00, WM00] and Vault [FD01]. In these type systems, locations are either linear, which we will indicate by 1, or non-linear, which we will indicate by ω [TWM95, BS96]. In these type systems, updates to linear locations, which represent a single object, are strong updates, and updates to non-linear locations are weak updates.

In this kind of type system there is a natural subtyping relation $1 \leq \omega$, i.e., it is safe to treat linear locations as non-linear. We can relate `restrict` to this subtyping. The Calculus of Capabilities includes a mechanism to temporarily promote linear locations to non-linear locations and then recover linearity in a continuation [CWM99]. By using `restrict` we can capture a new and different paradigm: We can locally downcast non-linear locations to linear locations. The downcast from ω to 1 is safe because of the semantics of `restrict`. By adding a `restrict` qualifier to the type of a pointer `p` the programmer agrees that they will only access `*p` through the name `p`. As long as that agreement is valid—i.e., within the scope of `restrict`—an analysis can treat `p` as linear, and perform strong updates.

8 Conclusion

We have presented a formal semantics for `restrict` and a type and effect system for checking the correctness of `restrict` annotations. While our type system does not address all the issues in checking `restrict` in ANSI C programs, we believe that it captures the key properties that a `restrict` annotation should have. We have shown that our proposed type and effect system is sound with respect to our semantics for `restrict`. We have also presented a type inference algorithm for `restrict` and described natural extensions of our system to include `affects` clauses, subtyping, and parametric polymorphism.

Acknowledgments

Thanks to David Gay, Ben Liblit, Tachio Terauchi, and David Walker for their comments.

References

[ABM$^+$97] Jeanne C. Adams, Walter S. Brainerd, Jeanne T. Martin, Brian T. Smith, and Jerrold L. Wagener. *FORTRAN 95 Handbook: Complete ISO/ANSI Reference*. MIT Press, Cambridge, 1997.

[And94] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, May 1994.

[ANS78] ANSI. *Fortran 77 Standard*, 1978. ANSI X3.9-1978.

[ANS99] ANSI. *Programming languages – C*, 1999. ISO/IEC 9899:1999.

[BCCH94] Michael Burke, Paul Carini, Jong-Deok Choi, and Michael Hind. Flow-Insensitive Interprocedural Alias Analysis in the Presence of Pointers. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, volume 892 of *Lecture Notes in Computer Science*, pages 234–250. Springer-Verlag, 1994.

[BS96] Erik Barendsen and Sjaak Smetsers. Uniqueness Typing for Functional Languages with Graph Rewriting Semantics. *Mathematical Structures in Computer Science*, 6(6):579–612, 1996.

[Cal01] Cristiano Calcagno. Stratified Operational Semantics for Safety and Correctness of The Region Calculus. In *Proceedings of the 28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 155–165, London, United Kingdom, January 2001.

[CRL99] Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. Relevant Context Inference. In *Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 133–146, San Antonio, Texas, January 1999.

[CWM99] Karl Crary, David Walker, and Greg Morrisett. Typed Memory Management in a Calculus of Capabilities. In *Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 262–275, San Antonio, Texas, January 1999.

[CWZ90] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of Pointers and Structures. In *Proceedings of the 1990 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 296–310, White Plains, New York, June 1990.

[Das00] Manuvir Das. Unification-based Pointer Analysis with Directional Assignments. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46, Vancouver B.C., Canada, June 2000.

[Deu94] Alain Deutsch. Interprocedural May-Alias Analysis for Pointers: Beyond k-limiting. In *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 230–241, Orlando, Florida, June 1994.

[DMW98] Saumya Debray, Robert Muth, and Matthew Weippert. Alias Analysis of Executable Code. In *Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 12–24, San Diego, California, January 1998.

[DRS98] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Detecting Memory Errors via Static Pointer Analysis. In *Proceedings of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 27–34, Montreal, Canada, June 1998.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–256, Orlando, Florida, June 1994.

[EST95] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type Inference for Recursively Constrained Types and its Application to OOP. In *Mathematical Foundations of Programming Semantics, Eleventh Annual Conference*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995.

[FD01] Manuel Fähndrich and Robert DeLine. Enforcing High-Level Protocols in Low-Level Software. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Snowbird, Utah, June 2001. To appear.

[FFSA98] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial Online Cycle Elimination in Inclusion Constraint Graphs. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, Montreal, Canada, June 1998.

[FRD00] Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable Context-Sensitive Flow Analysis using Instantiation Constraints. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 253–263, Vancouver B.C., Canada, June 2000.

[GJLS87] 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, September 1987.

[HP98] Michael Hind and Anthony Pioli. Assessing the Effects of Flow-Sensitivity on Pointer Alias Analyses. In Giorgio Levi, editor, *Static Analysis, Fifth International Symposium*, volume 1503 of *Lecture Notes in Computer Science*, pages 57–81, Pisa, Italy, September 1998. Springer-Verlag.

[HT01] Nevin Heintze and Olivier Tardieu. Ultra-fast Aliasing Analysis using CLA: A Million Lines of C Code in a Second. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Snowbird, Utah, June 2001. To appear.

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

[LR92] William Landi and Barbara G. Ryder. A Safe Approximate Algorithm for Interprocedural Pointer Aliasing. In *Proceedings of the 1992 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 235–248, San Francisco, California, June 1992.

[RF01] Jakob Rehof and Manuel Fähndrich. Type-Based Flow Analysis: From Polymorphic Subtyping to CFL-Reachability. In *Proceedings of the 28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 54–66, London, United Kingdom, January 2001.

[SH97] Marc Shapiro and Susan Horwitz. Fast and Accurate Flow-Insensitive Points-To Analysis. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Paris, France, January 1997.

[SRW99] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric Shape Analysis via 3-Valued Logic. In *Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 105–118, San Antonio, Texas, January 1999.

[Ste96] Bjarne Steensgaard. Points-to Analysis in Almost Linear Time. In *Proceedings of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, St. Petersburg Beach, Florida, January 1996.

[SWM00] Frederick Smith, David Walker, and Greg Morrisett. Alias Types. In Gert Smolka, editor, *9th European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–381, Berlin, Germany, 2000. Springer-Verlag.

[TB98] Mads Tofte and Lars Birkedal. A Region Inference Algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):724–767, July 1998.

[TJ95] Yan Mei Tang and Pierre Jouvelot. Effect Systems with Subtyping. pages 45–53, La Jolla, California, USA, June 1995.

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

[TWM95] David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *FPCA '95 Conference on Functional Programming Languages and Computer Architecture*, pages 1–11, La Jolla, California, June 1995.

[WF94] Andrew K. Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115(1):38–94, 1994.

[WL95] Robert P. Wilson and Monica S. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. In *Proceedings of the 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, La Jolla, California, June 1995.

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

[Wri95] Andrew K. Wright. Simple Imperative Polymorphism. In *Lisp and Symbolic Computation 8*, volume 4, pages 343–356, 1995.

[YHR99] Suan Hsi Yong, Susan Horwitz, and Thomas Reps. Pointer Analysis for Programs with Structures and Casting. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 91–103, Atlanta, Georgia, May 1999.

[ZRL96] Sean Zhang, Barbara G. Ryder, and William A. Landi. Program Decomposition for Pointer Aliasing: A Step toward Practical Analyses. In *Fourth Symposium on the Foundations of Software Engineering*, Toulouse, France, October 1996.

A Proof of Soundness

In our soundness proof, we will implicitly extend typing judgments to values $\Gamma \vdash v : \tau; L$ in the following way: Locations l will be represented in the proof as variables, and thus their types will be stored in Γ and they will typecheck using the rule (Var). We will implicitly treat semantic integers as syntactic integers, and use (Int) to typecheck them. Functions will be represented not as closures but as syntactic functions, as in standard small-step semantics subject-reduction proofs [WF94]. Thus semantic functions can be typechecked using (Lam). Notice that in the judgment $\Gamma \vdash v : \tau; L$, the set L must always be $\emptyset$, by simple inspection of the type rules. We will use this property implicitly in our proof.

Definition 3 (Compatibility) *We say Γ and L are* compatible *with store S, written $(\Gamma, L) \sim S$, if*

1. *$dom(\Gamma) = dom(S)$ and*
2. *for all $l \in dom(S)$, there exist ρ, τ such that $\Gamma(l) = ref^{\rho}(\tau)$ and*

$$\begin{cases} \Gamma \vdash S(l) : \tau; \emptyset & \text{if } S(l) \neq \textbf{\textit{err}} \\ \rho \notin L & \text{if } S(l) = \textbf{\textit{err}} \end{cases}$$

Intuitively, $(\Gamma, L) \sim S$ if an expression e that typechecks in environment Γ and has effects L can be run safely in store S.

Lemma 4 *If $(\Gamma, L \cup L') \sim S$ then $(\Gamma, L') \sim S$.*

Definition 4 (Extension) *We say that (Γ', S') is an* extension *of (Γ, S) if*

1. *$dom(\Gamma) = dom(S)$ and $dom(\Gamma') = dom(S')$ and*
2. *$\Gamma'|_{dom(\Gamma)} = \Gamma$*

Definition 5 (Safe Extension) *We say that (Γ', S') is a* safe extension *of (Γ, S), written $(\Gamma, S) \Rightarrow (\Gamma', S')$, if*

1. (Γ', S') is an extension of (Γ, S),

2. for all $l \in dom(S') - dom(S)$, if $S'(l) = \textbf{err}$ and $\Gamma'(l) = ref^{\rho}(\tau)$, then $\rho \notin \Gamma$, and

3. for all $l \in dom(S)$, if $S'(l) = \textbf{err}$ then $S(l) = \textbf{err}$.

*Intuitively, $(\Gamma, S) \Rightarrow (\Gamma', S')$ if the abstract locations corresponding to new **err** locations in S' don't appear in Γ.*

Lemma 5 *If $dom(\Gamma) = dom(S)$, then $(\Gamma, S) \Rightarrow (\Gamma, S)$.*

Lemma 6 *If $(\Gamma, S) \Rightarrow (\Gamma', S')$ and $(\Gamma', S') \Rightarrow (\Gamma'', S'')$, then $(\Gamma, S) \Rightarrow (\Gamma'', S'')$.*

Lemma 7 (ρ-Renaming) *If $\Gamma \vdash e : \tau; L$ and $\rho' \notin \Gamma, \tau, L$, then $R(\Gamma) \vdash e : R(\tau); R(L)$ for any substitution $R = [\rho \mapsto \rho']$.*

Proof: The assumption $\rho' \notin \Gamma, \tau, L$ means that ρ' is completely fresh: it is does not appear anywhere in the proof of $\Gamma \vdash e : \tau; L$. □

Lemma 8 (Substitution) *If $\Gamma \vdash v : \tau; \emptyset$ and $\Gamma[x \mapsto \tau] \vdash e : \tau'; L$, then $\Gamma \vdash e[x \mapsto v] : \tau'; L$.*

Theorem 2 (Subject Reduction) *If $\Gamma \vdash e : \tau; L$ and $S \vdash e \to r; S'$, where $(\Gamma, L \cup L') \sim S$, then there exists Γ' such that*

1. $\Gamma' \vdash r : \tau; \emptyset$ (which implies $r \neq \textbf{err}$),

2. $(\Gamma', L') \sim S'$, and

3. $(\Gamma, S) \Rightarrow (\Gamma', S')$

Proof: By induction on the structure of the proof $S \vdash e \to r; S'$. Recall that for each rule of Figure 2, there are corresponding reductions to **err** for cases for invalid programs. Thus for each case below, we will first reason based on the shape of e to decide which possible rule we used, and then show that r is not **err**.

$\boxed{\text{(Sub)}}$

Assume without loss of generality that all uses of (Sub) are after uses of (Down). Observe that as the last step of the proof $\Gamma \vdash e : \tau; L$ we may have used the rule (Sub):

$$\frac{\Gamma \vdash e : \tau'; L \qquad \tau' \leq \tau}{\Gamma \vdash e : \tau} \tag{1}$$

By assumption we also know

$$S \vdash e \to r; S' \tag{2}$$

$$(\Gamma, L \cup L') \sim S \tag{3}$$

Then by (1), (2), (3), and the case analysis below (in which we assume without loss of generality that the last rule applied was not (Sub)) there exists a Γ' such that

$$\Gamma' \vdash r : \tau'; \emptyset \tag{4}$$

$$(\Gamma', L') \sim S$$

$$(\Gamma, S) \Rightarrow (\Gamma', S')$$

Then combining (4) and (1) we have

$$\Gamma' \vdash r : \tau; \emptyset$$

and thus our conclusion holds.

Assume without loss of generality that the last rule applied in the proof was not (Sub).

$\boxed{\text{(Down)}}$

Observe that as the last step of the proof $\Gamma \vdash e : \tau; L - \{\rho_1, \ldots, \rho_n\}$ we may have used the rule (Down):

$$\frac{\Gamma \vdash e : \tau; L \qquad \rho_1, \ldots, \rho_n \notin \Gamma, \tau}{\Gamma \vdash e : \tau; L - \{\rho_1, \ldots, \rho_n\}} \tag{5}$$

By Lemma 1 we can assume that the proof of $\Gamma \vdash e : \tau; L$ does not use (Down) as the last step. We want to apply the case analysis below to show our conclusion, but first we need to do some more work, because we need to know that it's safe to evaluate an expression with effect L.

By assumption we also know

$$S \vdash e \rightarrow r; S' \tag{6}$$

$$(\Gamma, (L - \{\rho_1, \ldots, \rho_n\}) \cup L') \sim S \tag{7}$$

Pick fresh $\rho'_1, \ldots, \rho'_n$, that is, for all i

$$\begin{array}{c} \rho'_i \notin \Gamma, \tau, L, L' \\ \rho'_i \neq \rho'_j \quad \text{if } i \neq j \end{array} \tag{8}$$

Let

$$R = [\rho_1 \mapsto \rho'_1, \ldots, \rho_n \mapsto \rho'_n]$$

Then from (5) and Lemma 7 we have

$$R(\Gamma) \vdash e : R(\tau); R(L)$$

Then by (8) we derive

$$\Gamma \vdash e : \tau; R(L) \tag{9}$$

In other words, because $\rho_1, \ldots, \rho_n$ did not escape the scope of e, their names are arbitrary, and we can repeat the typing proof of e with any choice of names.

Now we want to show

$$(\Gamma, R(L) \cup L') \sim S \tag{10}$$

Clearly from (7) we have $dom(\Gamma) = dom(S)$, and for all $m \in dom(S)$ there are ρ_m, τ_m such that $\Gamma(m) = ref^{\rho_m}(\tau_m)$. If $S(m) \neq \mathbf{err}$ then we are done, since also from (7) we have $\Gamma \vdash S(m) : \tau_m; \emptyset$. So suppose that $S(m) = \mathbf{err}$. Then from (7) we know $\rho_m \notin (L - \{\rho_1, \ldots, \rho_n\}) \cup L'$. But since $\rho_m \in \Gamma$ and $\rho'_i \notin \Gamma$ from (8) we know that $\rho'_i \neq \rho_m$. Thus

$$\rho_m \notin (L - \{\rho_1, \ldots, \rho_n\}) \cup \{\rho'_1, \ldots, \rho'_n\} \cup L'$$

and therefore $\rho_m \notin R(L) \cup L'$. Thus (10) holds.

Then by (9), (6), (10), and the case analysis below, there exists a Γ' such that

$$\Gamma' \vdash r : \tau; \emptyset$$

$$(\Gamma', L') \sim S'$$

$$(\Gamma, S) \Rightarrow (\Gamma', S')$$

Thus our conclusion holds.

Assume without loss of generality that the last rule applied in the proof was neither (Sub) nor (Down).

$\boxed{x}$

By assumption we have

$$\frac{}{\Gamma \vdash x : \Gamma(x); \emptyset} \tag{11}$$

$$S \vdash x \rightarrow r; S' \tag{12}$$

$$(\Gamma, \emptyset \cup L') \sim S \tag{13}$$

By (13), $\mathit{dom}(\Gamma) = \mathit{dom}(S)$, so by (11), $x \in \mathit{dom}(S)$. Therefore we must have used the reduction

$$\frac{x \in \mathit{dom}(S)}{S \vdash x \to x; S} \tag{14}$$

So $r = x$ and $S' = S$. Let $\Gamma' = \Gamma$. Then our conclusion trivially holds:

$$\Gamma' \vdash x : \Gamma(x); \emptyset$$

$$(\Gamma', L') \sim S'$$

$$(\Gamma, S) \Rightarrow (\Gamma', S')$$

where the last conclusion holds by Lemma 5.

$\boxed{n}$
Trivial.

$\boxed{\texttt{ref}\, e}$
By assumption we have

$$\frac{\Gamma \vdash e : \tau; L}{\Gamma \vdash \texttt{ref}\, e : \mathit{ref}^{\rho}(\tau); L} \tag{15}$$

$$S \vdash \texttt{ref}\, e \to r; S' \tag{16}$$

$$(\Gamma, L \cup L') \sim S \tag{17}$$

By (16) and inspection of the semantic rules, we must have applied a reduction for e:

$$S \vdash e \to r_e; S'_e \tag{18}$$

By (15), (18), (17), and induction, there exists a Γ'_e satisfying

$$\Gamma'_e \vdash r_e : \tau; \emptyset \tag{19}$$

$$(\Gamma'_e, L') \sim S'_e \tag{20}$$

$$(\Gamma, S) \Rightarrow (\Gamma'_e, S'_e) \tag{21}$$

By (19), r_e is not **err**. Thus the reduction (16) must in fact be

$$\frac{S \vdash e \to r_e; S'_e \quad l \notin \mathit{dom}(S'_e)}{S \vdash \texttt{ref}\, e \to l; S'_e[l \mapsto r_e]} \tag{22}$$

with $S' = S'_e[l \mapsto r_e]$ and $r = l$ (see (16)). Let

$$\Gamma' = \Gamma'_e[l \mapsto \mathit{ref}^{\rho}(\tau)]$$

Clearly

$$\Gamma' \vdash l : \mathit{ref}^{\rho}(\tau); \emptyset$$

Further, since by (22) we have $l \notin \mathit{dom}(S'_e)$, we also have $l \notin \mathit{dom}(\Gamma'_e)$ by (20), and therefore $l \notin \mathit{dom}(\Gamma)$ by (21). Thus (Γ', S') is an extension of (Γ, S). Further, since $S'(l) = r_e \neq \mathbf{err}$ we have

$$(\Gamma'_e, S'_e) \Rightarrow (\Gamma', S') \tag{23}$$

Combining (23) and (21) by lemma 6, we have

$$(\Gamma, S) \Rightarrow (\Gamma', S')$$

Finally, by (20) we also have

$$(\Gamma', L') \sim S'$$

since $l \notin \mathit{dom}(S'_e)$ and $r_e \neq \mathbf{err}$.

Thus our conclusion holds.

$\boxed{*\,e}$

By assumption we have

$$\frac{\Gamma \vdash e : \mathit{ref}^{\rho}(\tau); L}{\Gamma \vdash *\,e : \tau; L \cup \{\rho\}} \tag{24}$$

$$S \vdash *\,e \rightarrow r; S' \tag{25}$$

$$(\Gamma, L \cup \{\rho\} \cup L') \sim S \tag{26}$$

By (25) and inspection of the semantic rules, we must have a reduction for e:

$$S \vdash e \rightarrow r_e; S'_e \tag{27}$$

By (24), (27), (26), and induction, there exists a Γ'_e satisfying

$$\Gamma'_e \vdash r_e : \mathit{ref}^{\rho}(\tau); \emptyset \tag{28}$$

$$(\Gamma'_e, \{\rho\} \cup L') \sim S'_e \tag{29}$$

$$(\Gamma, S) \Rightarrow (\Gamma'_e, S'_e) \tag{30}$$

By (28) we know r_e is a value and is not **err**. By inspection of the type rules we can see that the only type rule that can assign a value to the type $\mathit{ref}^{\rho}(\tau)$ is (Var). Notice that any uses of (Sub) in the proof (28) must be trivial, since there is no subtyping under a *ref* type constructor. Therefore we see that $r_e \in \mathit{dom}(\Gamma'_e)$, hence r_e is in fact a location and $r_e \in \mathit{dom}(S'_e)$ by (29). Therefore the reduction (25) must in fact be

$$\frac{S \vdash e \rightarrow r_e; S'_e \qquad r_e \in \mathit{dom}(S'_e)}{S \vdash *\,e \rightarrow S'_e(l); S'_e} \tag{31}$$

with $S' = S'_e$ and $r = S'_e(r_e)$ (see (25)). Let $\Gamma' = \Gamma'_e$. Then clearly

$$(\Gamma', L') \sim S'$$

by (29), and

$$(\Gamma, S) \Rightarrow (\Gamma', S')$$

by (29) and (30). Further, by (28) we know $\Gamma'(r_e) = \mathit{ref}^{\rho}(\tau)$. Then since $\rho \in \{\rho\} \cup L'$, by (29) we know $S'(r_e) \neq \mathbf{err}$ and

$$\Gamma' \vdash S'(r_e) : \tau; \emptyset$$

Thus our conclusion holds.

$\boxed{e_1 := e_2}$

By assumption we have

$$\frac{\Gamma \vdash e_1 : \mathit{ref}^{\rho}(\tau); L_1 \qquad \Gamma \vdash e_2 : \tau; L_2}{\Gamma \vdash e_1 := e_2 : \tau; L_1 \cup L_2 \cup \{\rho\}} \tag{32}$$

$$S \vdash e_1 := e_2 \rightarrow r; S' \tag{33}$$

$$(\Gamma, L_1 \cup L_2 \cup \{\rho\} \cup L') \sim S \tag{34}$$

By (33) and inspection of the semantic rules, we must have applied a reduction for e_1:

$$S \vdash e_1 \rightarrow r_{e_1}; S'_{e_1} \tag{35}$$

By (32), (35), (34), and induction, there exists a Γ'_{e_1} satisfying

$$\Gamma'_{e_1} \vdash r_{e_1} : \mathit{ref}^{\rho}(\tau); \emptyset \tag{36}$$

$$(\Gamma'_{e_1}, L_2 \cup \{\rho\} \cup L') \sim S'_{e_1} \tag{37}$$

$$(\Gamma, S) \Rightarrow (\Gamma'_{e_1}, S'_{e_1}) \tag{38}$$

By (36) we see that r_{e_1} is not **err**. Thus by inspection of the semantic rules, in (33) we must also have applied a reduction for e_2:

$$S'_{e_1} \vdash e_2 \rightarrow r_{e_2}; S'_{e_2} \tag{39}$$

By (32) and (38), we have

$$\Gamma'_{e_1} \vdash e_2 : \tau; L_2 \tag{40}$$

Then by (40), (39), (37), and induction, there exists a Γ'_{e_2} satisfying

$$\Gamma'_{e_2} \vdash r_{e_2} : \tau; \emptyset \tag{41}$$

$$(\Gamma'_{e_2}, \{\rho\} \cup L') \sim S'_{e_2} \tag{42}$$

$$(\Gamma'_{e_1}, S'_{e_1}) \Rightarrow (\Gamma'_{e_2}, S'_{e_2}) \tag{43}$$

By (36) we know that r_{e_1} is a value and is not **err**. By inspection of the type rules we see that the only type rule that can assign a value to the type $\mathit{ref}^{\rho}(\tau)$ is (Var). Notice that any uses of (Sub) in the proof (36) must be trivial, since there is no subtyping under a *ref* type constructor. Therefore we see that $r_{e_1} \in \mathit{dom}(\Gamma'_{e_1})$, hence r_{e_1} is in fact a location and $r_{e_1} \in \mathit{dom}(S'_{e_1})$ by (37).

Then by (43) we know that $r_{e_1} \in \mathit{dom}(S'_{e_2})$ and $\Gamma'_{e_2} \vdash r_{e_1} : \mathit{ref}^{\rho}(\tau)$. Then since $\rho \in \{\rho\} \cup L'$, by (42) it must be that $S'_{e_2}(r_{e_1})$ is not **err**. Therefore the reduction (33) must in fact be

$$\frac{\begin{array}{cc} S \vdash e_1 \rightarrow r_{e_1}; S'_{e_1} & S'_{e_1} \vdash e_2 \rightarrow r_{e_2}; S'_{e_2} \\ r_{e_1} \in \mathit{dom}(S'_{e_2}) & S'_{e_2}(r_{e_1}) \neq \mathbf{err} \end{array}}{S \vdash e_1 := e_2 \rightarrow r_{e_2}; S'_{e_2}[r_{e_1} \mapsto r_{e_2}]} \tag{44}$$

where $S' = S'_{e_2}[r_{e_1} \mapsto r_{e_2}]$ and $r = r_{e_2}$ (see (33)). Let $\Gamma' = \Gamma'_{e_2}$ Clearly we have

$$\Gamma' \vdash r_{e_2} : \tau; \emptyset$$

by (41). Combining (43) and (38) we have

$$(\Gamma, S) \Rightarrow (\Gamma'_{e_2}, S'_{e_2}) \tag{45}$$

Clearly, then, (Γ', S') is an extension of (Γ, S). And since $S'(r_{e_1}) = r_{e_2} \neq \mathbf{err}$ by (41), we have

$$(\Gamma, S) \Rightarrow (\Gamma', S')$$

Finally, also since $r_{e_2} \neq \mathbf{err}$ and since $r_{e_1} \in \mathit{dom}(S'_{e_2})$ and $\Gamma' \vdash r_{e_1} : \mathit{ref}^{\rho}(\tau)$, from (42) and (41) we can conclude

$$(\Gamma', L') \sim S'$$

Thus our conclusion holds.

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

By assumption, we know

$$\frac{\begin{array}{cc} \Gamma \vdash e_1 : \mathit{ref}^{\rho}(\tau); L_1 & \Gamma[x \mapsto \mathit{ref}^{\rho'}(\tau)] \vdash e_2 : \tau_2; L_2 \\ \rho' \notin \Gamma, \tau, \tau_2 & \rho \notin L_2 \end{array}}{\texttt{restrict}\, x = e_1\, \texttt{in}\, e_2 : \tau_2; L_1 \cup L_2 \cup \{\rho\}} \tag{46}$$

$$S \vdash \texttt{restrict}\, x = e_1\, \texttt{in}\, e_2 \rightarrow r; S' \tag{47}$$

$$(\Gamma, L_1 \cup L_2 \cup \{\rho\} \cup L') \sim S \tag{48}$$

By (47) and inspection of the semantic rules, we must have applied a reduction for e_1:

$$S \vdash e_1 \rightarrow r_{e_1}; S'_{e_1} \tag{49}$$

By (46), (49), (48), and induction, there exists a Γ'_{e_1} such that

$$\Gamma'_{e_1} \vdash r_{e_1} : \mathit{ref}^{\rho}(\tau); \emptyset \tag{50}$$

$$(\Gamma'_{e_1}, L_2 \cup \{\rho\} \cup L') \sim S'_{e_1} \tag{51}$$

$$(\Gamma, S) \Rightarrow (\Gamma'_{e_1}, S'_{e_1}) \tag{52}$$

By (50) we see that r_{e_1} is a value and is not **err**. By inspection of the type rules we see that the only type rule that can assign a value to the type $\mathit{ref}^{\rho}(\tau)$ is (Var). Notice that any uses of (Sub) in the proof (50) must be trivial, since there is no subtyping under a *ref* type constructor. Therefore we see that $r_{e_1} \in \mathit{dom}(\Gamma'_{e_1})$, hence r_{e_1} is in fact a location and $r_{e_1} \in \mathit{dom}(S'_{e_1})$ by (51). Thus by inspection of the semantic rules, in (47) we must also have applied

$$S'_{e_1}[r_{e_1} \mapsto \mathbf{err}, l' \mapsto S'_{e_1}(r_{e_1})] \vdash e_2[x \mapsto l'] \to r_{e_2}; S'_{e_2} \tag{53}$$

with

$$l' \notin \mathit{dom}(S'_{e_1}) \tag{54}$$

Before we can apply induction to e_2, we need to do a little more work. Pick a fresh ρ''. That is, pick a ρ'' such that

$$\rho'' \notin \{\rho\}, \Gamma, \Gamma'_{e_1}, \tau, \tau_2, L_2, L' \tag{55}$$

Let

$$R = [\rho' \mapsto \rho'']$$

Then by Lemma 7 and (46) we have

$$R(\Gamma[x \mapsto \mathit{ref}^{\rho'}(\tau)]) \vdash e_2 : R(\tau_2); R(L_2)$$

which by (46) is equivalent to

$$\Gamma[x \mapsto \mathit{ref}^{\rho''}(\tau)] \vdash e_2 : \tau_2; R(L_2)$$

Combining this with (52), we derive

$$\Gamma'_{e_1}[x \mapsto \mathit{ref}^{\rho''}(\tau)] \vdash e_2 : \tau_2; R(L_2)$$

Then by α-conversion, since $l' \notin \mathit{dom}(S'_{e_1})$ implies $l' \notin \Gamma'_{e_1}$ by (51), we can rename x to l' and derive

$$\Gamma'_{e_1}[l' \mapsto \mathit{ref}^{\rho''}(\tau)] \vdash e_2[x \mapsto l'] : \tau_2; R(L_2) \tag{56}$$

Finally, before we can apply induction, we need to show compatibility between the type environment in (56) and the store in (53). But which effect set should we use for compatibility? Clearly the set we choose cannot contain ρ, because the store in (53) contains a location corresponding to ρ that maps to **err**. Thus we use the following set L_{e_2}:

$$L_{e_2} = (L' - \{\rho\}) \cup R(L_2)$$

Also let

$$\Gamma_{e_2} = \Gamma'_{e_1}[l' \mapsto \mathit{ref}^{\rho''}(\tau)]$$
$$S_{e_2} = S'_{e_1}[r_{e_1} \mapsto \mathbf{err}, l' \mapsto S'(r_{e_1})]$$

where Γ_{e_2} is from (56) and S_{e_2} is from (53). Notice that Γ_{e_2} is an extension of Γ'_{e_1}, since by (54) and (51) $l' \notin \Gamma'_{e_1}$. We want to show

$$(\Gamma_{e_2}, L_{e_2}) \sim S_{e_2} \tag{57}$$

To see (57), first observe that $r_{e_1} \in \mathit{dom}(S'_{e_1})$ by (50) and (51), and observe that $\mathit{dom}(\Gamma'_{e_1}) = \mathit{dom}(S'_{e_1})$ by (51), and therefore $\mathit{dom}(\Gamma_{e_2}) = \mathit{dom}(S_{e_2})$.

For the second component of compatibility, pick any $m \in \mathit{dom}(S_{e_2})$, and suppose $\Gamma_{e_2}(m) = \mathit{ref}^{\rho_m}(\tau_m)$, which holds trivially by (51) and the construction of Γ_{e_2}. There are three cases:

1. Suppose $m = r_{e_1}$. Then $\rho_m = \rho$, and by construction of S_{e_2} we have $S_{e_2}(m) = \mathbf{err}$. But $\rho \notin L_2$ by (46), and since $\rho'' \neq \rho$ by (55) therefore $\rho \notin L_{e_2}$.

2. Suppose $m = l'$. Then $\rho_m = \rho''$. By construction of S_{e_2} we have $S_{e_2}(m) = S'_{e_1}(r_{e_1})$. But by (50) and (51) we have $S'_{e_1}(r_{e_1}) \neq \mathbf{err}$ and $\Gamma'_{e_1} \vdash S'_{e_1}(r_{e_1}) : \tau; \emptyset$. But then since Γ_{e_2} is an extension of Γ'_{e_1}, we also have $\Gamma_{e_2} \vdash S'_{e_1}(r_{e_1}) : \tau; \emptyset$.

3. Suppose $m \neq r_{e_1}, m \neq l'$. Then $S_{e_2}(m) = S'_{e_1}(m)$ and $\Gamma_{e_2}(m) = \Gamma'_{e_1}(m)$. By (55), it must be that $\rho_m \neq \rho''$. If $S'_{e_1}(m) \neq \mathbf{err}$, then by (51) we have $\Gamma'_{e_1} \vdash S'_{e_1}(m) : \tau_m; \emptyset$, and since Γ_{e_2} is an extension of Γ'_{e_1} we also have $\Gamma_{e_2} \vdash S'_{e_1}(m) : \tau_m; \emptyset$. Otherwise, suppose $S'_{e_1}(m) = \mathbf{err}$. Then by (51) we have $\rho_m \notin L_2 \cup \{\rho\} \cup L'$. Thus clearly $\rho_m \notin L' - \{\rho\}$. Since $\rho_m \notin L_2$ and $\rho'' \neq \rho_m$, we have $\rho_m \notin R(L_2)$. Therefore $\rho_m \notin L_{e_2}$.

Thus (57) holds.

Then by (56), (53), (57), and induction, there exists a Γ'_{e_2} such that

$$\Gamma'_{e_2} \vdash r_{e_2} : \tau_2; \emptyset \tag{58}$$

$$(\Gamma'_{e_2}, L' - \{\rho\}) \sim S'_{e_2} \tag{59}$$

$$(\Gamma_{e_2}, S_{e_2}) \Rightarrow (\Gamma'_{e_2}, S'_{e_2}) \tag{60}$$

Now we're almost done. By (58) we see that r_{e_2} is not **err**. Combining this with the fact that r_{e_1} is not **err** (from (50)) and with (49), (53), and (54), we see that the reduction (47) must have been

$$\frac{\begin{array}{c} S \vdash e_1 \rightarrow r_{e_1}; S'_{e_1} \\ r_{e_1} \in \mathit{dom}(S'_{e_1}) \qquad\qquad l' \notin \mathit{dom}(S'_{e_1}) \\ S'_{e_1}[r_{e_1} \mapsto \mathbf{err}, l' \mapsto S'_{e_1}(r_{e_1})] \vdash e_2[x \mapsto l'] \rightarrow r_{e_2}; S'_{e_2} \end{array}}{\begin{array}{c} S \vdash \texttt{restrict}\, x = e_1 \,\texttt{in}\, e_2 \rightarrow r_{e_2}; \\ \qquad S'_{e_2}[r_{e_1} \mapsto S'_{e_2}(l'), l' \mapsto \mathbf{err}] \end{array}} \tag{61}$$

with

$$S' = S'_{e_2}[r_{e_1} \mapsto S'_{e_2}(l'), l' \mapsto \mathbf{err}]$$

and

$$r = r_{e_2}$$

(see (47)). Let $\Gamma' = \Gamma'_{e_2}$. We show the conclusions of the inductive hypothesis one by one.

First, by (58) we have

$$\Gamma' \vdash r_{e_2} : \tau_2; \emptyset \tag{62}$$

Next we need to show

$$(\Gamma', L') \sim S' \tag{63}$$

We proceed as in the proof of (57). Clearly $\mathit{dom}(\Gamma') = \mathit{dom}(S'_{e_2})$ by (59). And by construction of S_{e_2} we have $r_{e_1}, l' \in \mathit{dom}(S_{e_2})$. Then by by (60) we see $r_{e_1}, l' \in \mathit{dom}(S'_{e_2})$. Thus $\mathit{dom}(S') = \mathit{dom}(S'_{e_2}) = \mathit{dom}(\Gamma'_{e_2}) = \mathit{dom}(\Gamma')$.

For the second component of compatibility, pick any $m \in \mathit{dom}(S')$, and suppose $\Gamma'(m) = \mathit{ref}^{\rho_m}(\tau_m)$, which holds trivially by (59). There are three cases:

1. Suppose $m = r_{e_1}$. Then $S'(m) = S'_{e_2}(l')$. Since $S_{e_2}(l') \neq \mathbf{err}$, by (60) we see that $S'_{e_2}(l') \neq \mathbf{err}$. By the construction of Γ_{e_2} and (60) we see that $\Gamma'_{e_2}(l') = \mathit{ref}^{\rho''}(\tau)$. But then by (59) we have $\Gamma'_{e_2} \vdash S'_{e_2}(l') : \tau; \emptyset$, and hence $\Gamma' \vdash S'(r_{e_1}) : \tau; \emptyset$.

2. Suppose $m = l'$. Then $\rho_m = \rho''$ and $S'(l') = \mathbf{err}$. But then by (55) we know $\rho'' \notin L'$.

3. Suppose $m \neq r_{e_1}$, $m \neq l'$. Then $S'(m) = S'_{e_2}(m)$. Suppose $S'_{e_2}(m) \neq \mathbf{err}$. Then from (59) we know $\Gamma' \vdash S'(m) : \tau_m; \emptyset$. Otherwise, suppose $S'(m) = S'_{e_2}(m) = \mathbf{err}$. There are two cases. If $m \in \mathit{dom}(S_{e_2})$, then $m \in \mathit{dom}(S')$ by (52). Then by (51) we know $\rho_m \notin L'$. Otherwise, suppose $m \notin \mathit{dom}(S_{e_2})$. Then by (60) $\rho_m \notin \Gamma_{e_2}$. But since $\rho \in \Gamma_{e_2}$ (which we can conclude from (50) and the construction of Γ_{e_2}), we know that $\rho_m \neq \rho$. By (59) we have $\rho_m \notin L' - \{\rho\}$, and since $\rho_m \neq \rho$ we see that $\rho_m \notin L'$.

Thus (63) holds. Finally, we need to show

$$(\Gamma, S) \Rightarrow (\Gamma', S') \tag{64}$$

Clearly by (63) we have $dom(\Gamma') = dom(S')$, and by assumption (48) we have $dom(\Gamma) = dom(S)$. Also by (52), construction of Γ_{e_2}, and (60) we see that (Γ', S') is an extension of (Γ, S). So we just need to show that it's a safe extension.

Pick any $m \in dom(S')$. If $S'(m) \neq \mathbf{err}$ then we're done. Otherwise suppose $S'(m) = \mathbf{err}$ and $\Gamma'(m) = ref^{\rho_m}(\tau_m)$. There are three cases:

1. Suppose $m = r_{e_1}$. This is impossible, since $S'(r_{e_1}) = S'_{e_2}(l') \neq \mathbf{err}$.
2. Suppose $m = l'$. Then $l' \notin dom(S)$ by (54) and (52). So we need to show $\rho_m \notin \Gamma$. But $\rho_m = \rho''$ by construction of Γ_{e_2} and (60). And $\rho'' \notin \Gamma$ by (55).
3. Suppose $m \neq r_{e_1}, m \neq l'$. Then $S'(m) = S'_{e_2}(m)$. If $m \in dom(S'_{e_2}) - dom(S_{e_2})$ then by (60) we see that $\rho_m \notin \Gamma_{e_2}$. But then by construction of Γ_{e_2} and (52) we see $\rho_m \notin \Gamma$.

   Otherwise if $m \in dom(S_{e_2})$ then by (60) we see that $S_{e_2}(m) = \mathbf{err}$. But $S_{e_2}(m) = S'_{e_1}(m)$. Then there are again two cases. If $m \in dom(S'_{e_1}) - dom(S)$, then by (52) we see that $\rho_m \notin \Gamma$. Otherwise if $m \in dom(S)$ then by (52) we see that $S(m) = \mathbf{err}$.

Thus (64) holds. Combining (62), (63), and (64) we see that our conclusion holds.

$\boxed{\lambda x.e}$
Trivial.

$\boxed{e_1\ e_2}$
By assumption we have

$$\frac{\Gamma \vdash e_1 : \tau_1 \xrightarrow{L} \tau_2; L_1 \qquad \Gamma \vdash e_2 : \tau_1; L_2}{\Gamma \vdash e_1\ e_2 : \tau_2; L_1 \cup L_2 \cup L} \tag{65}$$

$$S \vdash e_1\ e_2 \to r; S' \tag{66}$$

$$(\Gamma, L_1 \cup L_2 \cup L \cup L') \sim S \tag{67}$$

By (66) and inspection of the semantic rules, we must have applied a reduction for e_1:

$$S \vdash e_1 \to r_{e_1}; S'_{e_1} \tag{68}$$

By (65), (68), (67), and induction, there exists a Γ'_{e_1} satisfying

$$\Gamma'_{e_1} \vdash r_{e_1} : \tau_1 \xrightarrow{L} \tau_2; \emptyset \tag{69}$$

$$(\Gamma'_{e_1}, L_2 \cup L \cup L') \sim S'_{e_1} \tag{70}$$

$$(\Gamma, S) \Rightarrow (\Gamma'_{e_1}, S'_{e_1}) \tag{71}$$

By (69) we know that r_{e_1} is a value and is not $\mathbf{err}$. By inspection of the type rules we see that the only type rules that can assign a value to the type $\tau_1 \xrightarrow{L} \tau_2$ are (Var) and (Lam), possibly followed by an application of (Sub). But by (70) we know that Γ'_{e_1} assigns only reference types. Thus the proof (69) must in fact be

$$\frac{\dfrac{\Gamma[x \mapsto \tau'_1] \vdash e : \tau'_2; L_f}{\Gamma \vdash \lambda x.e : \tau'_1 \xrightarrow{L_f} \tau'_2; \emptyset} \quad \tau_1 \leq \tau'_1 \qquad \tau'_2 \leq \tau_2 \qquad L_f \subseteq L}{\Gamma \vdash \lambda x.e : \tau_1 \xrightarrow{L} \tau_2; \emptyset} \tag{72}$$

where $r_{e_1} = \lambda x.e$.

Further, by inspection of the semantic rules, in (66) we must also have applied a reduction for e_2:

$$S'_{e_1} \vdash e_2 \rightarrow r_{e_2}; S'_{e_2} \tag{73}$$

By (65) and (71), we have

$$\Gamma'_{e_1} \vdash e_2 : \tau_1; L_2 \tag{74}$$

Then by (74), (73), (70), and induction, there exists a Γ'_{e_2} satisfying

$$\Gamma'_{e_2} \vdash r_{e_2} : \tau_1; \emptyset \tag{75}$$

$$(\Gamma'_{e_2}, L \cup L') \sim S'_{e_2} \tag{76}$$

$$(\Gamma'_{e_1}, S'_{e_1}) \Rightarrow (\Gamma'_{e_2}, S'_{e_2}) \tag{77}$$

From (75) we know that r_{e_2} is not **err**. Thus by inspection of the semantic rules, in (66) we must also have applied a reduction for $e[x \mapsto r_{e_2}]$:

$$S'_{e_2} \vdash e[x \mapsto r_{e_2}] \rightarrow r_e; S'_e \tag{78}$$

Combining (71) and (77) we see

$$(\Gamma, S) \Rightarrow (\Gamma'_{e_2}, S'_{e_2}) \tag{79}$$

Now by (72) and (79) we see that

$$\Gamma'_{e_2}[x \mapsto \tau'_1] \vdash e : \tau'_2; L_f \tag{80}$$

where $L_f \subseteq L$. By (75) and $\tau_1 \leq \tau'_1$ from (72) we see that

$$\Gamma'_{e_2} \vdash r_{e_2} : \tau'_1; \emptyset \tag{81}$$

Then by (80), (81), and Lemma 8 we have

$$\Gamma'_{e_2} \vdash e[x \mapsto r_{e_2}] : \tau'_2; L_f \tag{82}$$

From (76) we have

$$(\Gamma'_{e_2}, L_f \cup (L - L_f) \cup L') \sim S'_{e_2} \tag{83}$$

Now by (82), (78), (83), and induction, there exists a Γ'_e satisfying

$$\Gamma'_e \vdash r_e : \tau'_2; \emptyset \tag{84}$$

$$(\Gamma'_e, (L - L_f) \cup L') \sim S'_e \tag{85}$$

$$(\Gamma'_{e_2}, S'_{e_2}) \Rightarrow (\Gamma'_e, S'_e) \tag{86}$$

where $r = r_e$ and $S = S'_e$ (see (66)). Let $\Gamma' = \Gamma'_e$. Then clearly since $\tau'_2 \leq \tau_2$ by (72) we have

$$\Gamma' \vdash r : \tau_2; \emptyset$$

from (84). Then we also have

$$(\Gamma', L') \sim S'$$

from (85). Finally, we get

$$(\Gamma, S) \Rightarrow (\Gamma', S')$$

combining (79) and (86). Thus our conclusion holds.

□

Given the subject-reduction theorem, soundness is easy to show:

Theorem 3 *If $\emptyset \vdash e : \tau; L$, then $\emptyset \vdash e \rightarrow r; S'$, where r is not **err***

Proof: First observe that $(\emptyset, L) \sim \emptyset$. Then in our semantics every term can be reduced to some result r. By the subject-reduction theorem, there is a Γ' such that $\Gamma' \vdash r : \tau; \emptyset$. Thus r is not **err**. □