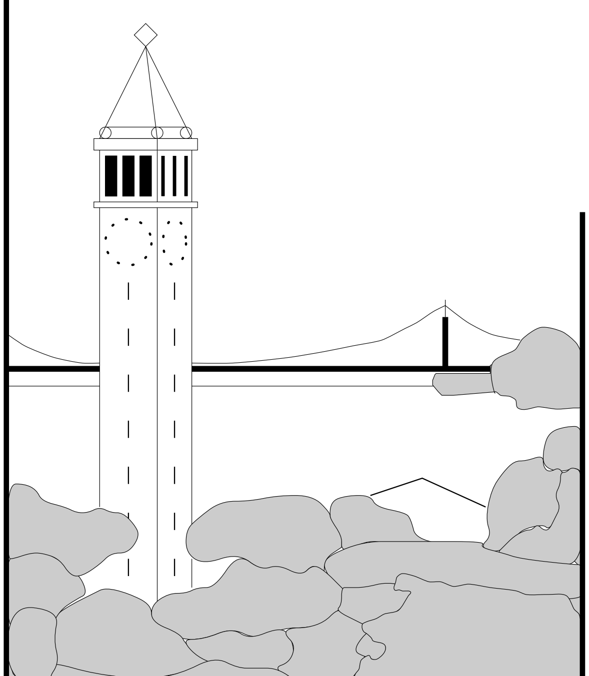
Types for Lexically-Scoped Access Control

Tachio Terauchi *Alex Aiken* *Jeffrey S. Foster*



Report No. UCB/CSD-3-1282

October 2003

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

Types for Lexically-Scoped Access Control *

Tachio Terauchi   Jeffrey S. Foster   Alex Aiken

October 15, 2003

Abstract

We develop a new system for defining and enforcing access control statically. In our system, *key-pairs* guard access to *resources*, and the association between key-pairs and resources can be changed at any program point (i.e., the binding is late). Our static system uses an ordering on lexically scoped abstract names to allow local access control policies to be enforced in other parts of a program. In particular, this means that individual program components can locally refine access control policies and the policies will be respected by the entire program. The result is a system that can enforce, at compile time, a wide variety of useful, fine-grain access control patterns.

1 Introduction

In situations where a program P interacts with one or more untrusted program components U, a well-specified access control policy protects P's resources from unwanted operations performed by U.

Systems such as Java and the Common Language Runtime (CLR) provide a mechanism for defining and enforcing fine-grain access control. In these systems, a programmer defines permissions denoting resources that pieces of code are allowed to access. For example, a Java class that conducts network transactions needs permission to access network sockets. A key part of *stack-based* access control is ensuring that if component C does not have permission to access resource v, then C cannot gain access to v by calling a function that has permission to access v. To enforce stack-based access control policies, the run-time system records the set of permissions $S_1, S_2, \ldots$ of each codes $f_1, f_2, \ldots$ on the execution stack. When a resource v is accessed, the run-time system permits the access only if $v \in \bigcap_i S_i$. That is, v can be used only if all functions on the stack have permission to access v. Stack-based access control systems usually also include a mechanism for gaining privileges, so that only a suffix of the stack is examined for access right.

In addition to a set of pre-defined permissions, Java and the CLR allow programmers to define new permissions. For example, in an airline ticketing program, a flight scheduling component may define new permissions used to protect internal resources such as flight time information from being modified by a ticketing agent component.

This paper presents a new stack-based access control system that is enforced statically, in contrast to the run-time enforcement in Java and the CLR. We begin with an example application of our system, slightly simplified for the purposes of introduction. Before giving the example, we give a very compressed overview of our approach.

In our system, permissions are expressed as *key-pairs* consisting of a *grant* key and a *limit* key. Each key-pair is associated with a set of resources, but the two keys are used for different operations on those resources. We have already mentioned that in stack-based access control each piece of code may limit access to a set of resources; this is the function of limit keys. The power to limit access is not sufficient, however. Something must also grant access to resources, at least initially, or no resources could ever be used; this is the function of grant keys. Splitting the management of access rights for a set of resources into grant and limit keys is fundamental to our design and, we believe, gives our system practical advantages (see Section 2).

*This research was supported in part by Subcontract no. PY-1099 to Stanford, Dept. of the Air Force prime contract no. F33615-00-C-1693.

A second novel feature of our system is *subkeys*. Subkeys are analogous to subclasses in an object-oriented language. If a limit (grant) key k_1 is a subkey of a limit (grant) key k_2 then k_1 guards access to a subset of the resources k_2 protects (see Section 2.5).

Finally, in our approach the programmer writes code to expresses access control policies using the following expressions for manipulating keys and resources:

- `newkey` generates a new pair of keys; `newkey` $< p$ generates new subkeys of the key-pair p. Initially, keys are not associated with any resources.
- `associate` e_1 `with` e_2 associates the key-pair e_1 with the resource e_2.
- `limit` e_1 `in` e_2 limits e_2's access rights to the resources associated with the limit key e_1.
- `grant` e_1 `in` e_2 grants e_2 access rights to the resources associated with the grant key e_1.
- The function `gKey`(p) (resp. `lKey`(p)) returns the grant (resp. limit) key of the key pair p.

Consider two components `C` and `D` that need to share access to a number of files. This is expressed in our system by creating a key-pair, associating it with the files, and then granting access to those files to both `C` and `D`:

```
shared_files = newkey;
...associate file_a with shared_files;
...associate file_b with shared_files;
[Component C]:
  ...
  grant gKey(shared_files) in
    .../* accesses the shared files */
    ...
[Component D]:
  ...
  grant gKey(shared_files) in
    .../* accesses the shared files */
```

Now assume that we wish to add to `C` a function `f` that is visible to other, less trusted, components, which may or may not have access to `shared_files`. We want only certain files in `shared_files` to be accessible through `f`. Our system is designed to support such refinements of access control policies with only local modifications to component `C`; in particular, neither component `D` nor any other component needs to be modified to implement these changes. We create a subkey-pair `some_files` of `shared_files` and associate it with a subset of files, say just `file_a` and `file_b`. Using `limit`, we ensure that callers of the function `f` can access at most the files associated with `some_files`; which files they can actually access depends on their access rights at the point where they call `f`.

```
[Component C]:
  grant gKey(shared_files) in
    ...
    some_files = newkey<shared_files;
    ...associate file_a with some_files;
    ...associate file_b with some_files;
    ...
    f = λx.limit lKey(some_files) in
           .../* accesses the selected files */
```

Component `D` can call `f` because it has access to `shared_files`. But components without access to the selected files cannot.

In our view, our system provides two key features: late binding of keys and resources (i.e., a resource can be associated with a key after both the key and the resource exist) and the ability to create subkeys at any point. As a result, these features allow late, local refinement of access control policies, as well as non-local enforcement of policies. In the example above, we have added a refinement of the original policy to function `f` in component `C`. This policy is local to `C`, meaning that the entire refinement is done in the component `C`, but it is enforced non-locally; i.e., a call to `f` in `D` respects the refined access control policy.

Our aim is to *statically* check such fine-grain access control policies. There are two benefits to static access control. One benefit is that it can find access violations early at compile time. The other benefit is efficiency; the run-time system need not to check for access violations.

Static checking requires our system to statically track three things: (a) the association between resources and keys; (b) the subkey hierarchy; and (c) which resources are used by expressions. For (a), we use abstract names, which are lexically scoped, to identify keys and to qualify resource types with their associated keys. It is important to note that only the static names of keys are lexically scoped; keys themselves are not lexically scoped and in fact can be passed as arguments, stored in data structures, etc. For (b), the static system tracks the *subnaming* hierarchy, which is the static analog of the subkey hierarchy. Finally, for (c) we develop a type and effect system that, for each expression e, assigns an effect set describing the set of resources accessed in the evaluation of e.

1.1 Contributions and Overview

This paper makes several contributions:

- Our access control system is static, i.e., the system checks for access violations at compile time. This is similar to some other recent research (e.g., [12]) but in stark contrast to common implementations of Java and CLR where violations are checked at run-time.
- Despite being static, the system is able to define and enforce many non-trivial access control patterns through its ability to reason about locally refined access control policies.
- The heart of our formal system is a novel, type-based must alias analysis, which may be of independent interest.
- We have a proof of soundness for the core subset of the system. Our system is sound in the presence of updatable references, higher-order functions, and concurrency.

Our system is type-based, relying only on standard techniques. One reason for choosing a type-based approach is compositional verification. Type systems explicitly express assumptions about the environment of each program fragment. Therefore program components may be checked separately under compatible environments, and then composed to form a well-typed program.

The rest of the paper proceeds as follows. Section 2 introduces our system informally. We introduce the important features of the system step-by-step, making improvements as we proceed. Materials up to and including Section 2.2 are sufficient to perform late, local refinement of access control policies. A small example is given in Section 2.4. To enforce local access policies non-locally, we introduce subkeys and subnaming in Section 2.5. Section 3 presents the static system formally and sketches a proof of soundness (the complete proof appears in Appendix A). Section 4 shows a few extensions to the system. Section 5 discusses our system's relation to must alias analysis. Section 6 discusses related work. Section 7 concludes.

2 Informal Presentation

This section informally develops the key ideas in our access control system. In the interests of clarity, we present some of the material introduced in Section 1 again, but in a more leisurely fashion and with more examples. The formal details of our system are presented in Section 3.

In a typical implementation of a dynamic stack-based access control system (such as Java's), permissions are objects containing strings, and the run-time system checks permissions by inspecting the strings before each access of associated resources. For example, permission to access flight information on flight 356 may be represented as a string `"Flight Info :  356"`. Instead, we design access control into the static semantics of the language so that the type system can precisely track access control policies.

We introduce *key-pairs*, each consisting of a *limit key* and a *grant key*. Each key-pair is used to group resources. Limit keys are used to limit access to resources and grant keys are used to grant access to resources.

We consider every program value (e.g. a function, a pointer, etc) to be a resource possibly associated with some key-pair. Key-pairs are named in the static system using an infinite set `Names`. That is, each $\rho \in \texttt{Names}$ identifies a key-pair. While technically ρ is a static name for a run-time key-pair, for brevity we say "the key-pair ρ" instead of "the key-pair identified by ρ" when there can be no confusion.

Each type τ in our system consists of a name and a *raw type*. A resource of the raw type σ associated with the key-pair ρ is given the type $\rho\ \sigma$; we say that ρ *qualifies* σ in this type. For example, each pointer type is of the form $\rho\ \mathit{ref}(\tau)$.

The limit (resp. grant) key part of the key-pair ρ has the raw type $\mathit{lkey}(\rho)$ (resp. $\mathit{gkey}(\rho)$). Keys are themselves program values. Therefore keys may also be under access control, and thus are associated with a key-pair. A limit (resp. grant) key ρ associated with the key-pair ρ_1 is given a type $\rho_1\ \mathit{lkey}(\rho)$ (resp. $\rho_1\ \mathit{gkey}(\rho)$).

Finally, we use $\perp\ \sigma$ to denote the type of a program value not associated with any key-pair. Such program values are not subject to access control, hence they may be accessed by any code.

2.1 Limit and Grant

To limit access rights, we introduce the syntax form

$$\texttt{limit}\ e_1, e_2, \ldots, e_{n-1}\ \texttt{in}\ e_n$$

where e_1, e_2, ..., e_{n-1} are the limit keys and e_n is the code whose access rights are limited to the resources associated with the keys e_1, e_2, ..., e_{n-1}.

Let `lkey` be the limit key of key-pair ρ. Assume that the function `h` and the pointer `t` are associated with `lkey`, and consider this definition:

$$\texttt{g = } \lambda x.\ \texttt{limit lkey in}\ x\texttt{(h, ! t)}$$

Since `g` dereferences `t`, `g` can be called only by code with access to `t`. For example, the following code should be rejected if the limit key `akey` is not associated with `t`. ($\lambda\langle x_1, \ldots, x_n\rangle.e$ is a function of n arguments.)

$$\texttt{limit akey in g}\ (\lambda\langle x, y\rangle.y)$$

Moreover, because the access rights of the body of `g` are limited to resources associated with `lkey`, the function passed to `g` may only access resources associated with `lkey`. So if `lkey` is associated only with `h` and `t`, then `g` is forbidden from accessing any resource but `h` or `t`. For example, if `u` is a pointer not equal to `t` (and is not qualified with $\perp$)

$$\texttt{g}\ (\lambda\langle x, y\rangle.\texttt{! u})$$

is forbidden.

Now that we have seen how to limit access rights, we turn to granting access rights. We introduce the syntax form

$$\texttt{grant}\ e_1\ \texttt{in}\ e_2$$

where e_1 is a *grant* key and e_2 is the code granted access to the resources associated with key e_1. Note that `grant` takes only a single expression as an argument, in contrast to `limit`, because multiple sequential `grant`s can be used to grant a set of permissions, where multiple sequential `limit`s will limit access to the intersection of access rights in the `limit` expressions.

Let `gkey` be the grant key part of the key-pair ρ. The code below grants itself access to the resources `h` and `t` in order to call `g`.

$$\texttt{grant gkey in g}\ (\lambda\langle x, y\rangle.x\ (y))$$

A `grant` is stronger than a `limit` because it enables unconditional access to the resources regardless of what access rights are available to the code's context. In Java, the analog of `limit` is the permissions attached to traversed methods and the analog of `grant` is the initial set of permissions granted for an execution thread and additional accesses granted by the special form `doPrivileged`.

We use a type and effect system [7, 9] to statically check access control policies. For each expression, we assign an effect set conservatively approximating the expression's access rights. Effect sets are finite sets of key-pair names. The type checking rules derive judgments of the form $\Gamma \vdash e : \tau; L$ (read "the expression e has the type τ in the type environment Γ and requires access rights L"). For example, the type checking rule for pointer dereferences is:

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

This rule says that the dereference of a pointer is well-typed if the access to the pointer is enabled.

The type checking rule for `limit` is more elaborate:

$$\frac{\begin{array}{c}\text{For } 1 \leq i \leq n-1,\ \Gamma \vdash e_i : \rho'_i\ \mathit{lkey}\,(\rho_i); L_i \\ \Gamma \vdash e_n : \tau_n; L_n \qquad\qquad \Gamma \vdash L_n \subseteq \{\rho_1, \rho_2, \ldots, \rho_{n-1}\}\end{array}}{\Gamma \vdash \texttt{limit}\ e_1, e_2, \ldots, e_{n-1}\ \texttt{in}\ e_n : \tau_n; L_n \cup \bigcup_{1 \leq i \leq n-1} (L_i \cup \{\rho'_i\})}$$

The second line says that the body of the expression, e_n, only accesses the resources associated with the key-pairs $\rho_1, \rho_2, \ldots, \rho_{n-1}$. In the conclusion, note the presence of L_n requires that the context have the right to access the key-pairs in L_n; this prevents e_n from gaining more access rights than its context. On the first line, since each key e_i is itself a resource, it is associated with a key-pair ρ'_i (if it is not associated with any key-pair then $\rho'_i = \perp$). Moreover, evaluating each e_i may also involve resource accesses, approximated by L_i. Therefore, the context must have access rights to the resources associated with key-pairs identified by $\bigcup_{1 \leq i \leq n-1} (L_i \cup \{\rho'_i\})$, which is included in the overall effect.

The type checking rule for `grant` is simpler:

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \rho_1\ \mathit{gkey}\,(\rho_2); L_1 \\ \Gamma \vdash e_2 : \tau; L_2 \cup \{\rho_2\}\end{array}}{\Gamma \vdash \texttt{grant}\ e_1\ \texttt{in}\ e_2 : \tau; L_1 \cup L_2 \cup \{\rho_1\}}$$

The key e_1 is required to have a type of the form $\rho_1\ \mathit{gkey}\,(\rho_2)$, and e_2 is granted the access to the resources associated with the key-pair ρ_2 as indicated by the presence of $\{\rho_2\}$ in e_2's effect. However, unlike with `limit`, the context need not have access to ρ_2.

Grant keys are dangerous because they enable unconditional access rights regardless of the context. Hence programmers should prevent arbitrary code from using grant keys and gaining unwanted access rights. By splitting keys into key-pairs instead of using a single key for both limiting and granting, we have sought to make it easy to isolate and control the ability to grant access to resources. In particular, this design allows programmers to code in a style where each grant key is forgotten immediately after it is used to grant the access right to the code that generated the key.

Nevertheless, grant keys can be useful outside of this trivial style as long as they are used with discretion. As an example, consider system calls in the UNIX operating system. Application programs are usually prohibited from accessing system resources such as printers except through system calls. We can emulate the behavior of such a protocol using grant keys. Let all printers be associated with the grant key `k_printers`. Then the system function `print` can be written

```
λ⟨x, y⟩.grant k_printers in {
  .../* code for outputting x to the printer y */ }
```

so that a program that does not have access to the printers may print by calling this function. [1]

[1] The Java security model has a similar feature: `doPrivileged` enables a code to access more resources than are available to the code's context.

2.2 Associating Resources with Key-Pairs

We next introduce the syntax forms `newkey` to generate new key-pairs and `associate` e_1 `with` e_2 to associate resources with key-pairs. The syntax form `newkey` $< e$, which generates new subkey-pairs, is introduced together with subnaming in Section 2.5.

We first describe `newkey`. We use existential types to guarantee that different key-pairs (particularly those generated by the same syntactic occurrence of `newkey`) are identified by different names.

$$\frac{}{\Gamma \vdash \texttt{newkey} : \perp (\exists \rho . \perp \langle \perp \mathit{lkey}(\rho), \perp \mathit{gkey}(\rho) \rangle); \emptyset}$$

Raw tuple types are written $\langle \tau_1, \ldots, \tau_n \rangle$. For extracting grant (resp. limit) keys from a pair, we define `lKey` (resp. `gKey`) as the projection function $\lambda x.(x.1)$ (resp. $\lambda x.(x.2)$).

Note that the type of a newly generated key-pair is qualified with $\perp$, i.e., by default the new key-pair is not associated with any key-pair and hence is available to any code. Other program values are also qualified with $\perp$ when they are created, also making them available to any code. For example, the type checking rule for allocating a new store location (i.e. a new pointer) is

$$\frac{\Gamma \vdash e : \tau; L}{\Gamma \vdash \texttt{ref}\, e : \perp \mathit{ref}(\tau); L}$$

In our system, each association between a key and a resource occurs after both the key and the resource are created, i.e., associations happen late. We use the syntax form

$$\texttt{associate}\ e_1\ \texttt{with}\ e_2$$

to associate the resource e_1 with the key-pair with limit key e_2.[2] The type checking rule for `associate` is shown below.

$$\frac{\Gamma \vdash e_1 : \rho_1\ \sigma; L_1 \qquad \Gamma \vdash e_2 : \rho_2\ \mathit{lkey}(\rho_3); L_2}{\Gamma \vdash \texttt{associate}\ e_1\ \texttt{with}\ e_2 : \rho_3\ \sigma; L_1 \cup L_2 \cup \{\rho_1, \rho_2\}}$$

The `associate` construct is similar to *type casting*. That is, given a resource of the type $\rho_1\ \sigma$, associating it with the key-pair ρ_3 has the effect of casting the type to $\rho_3\ \sigma$.

In the rule for `associate`, because the limit key e_2 may itself be associated with some key-pair (identified by ρ_2), we assert that the context must hold the access right to ρ_2. We assert that the context must also have access to the resource e_1, and this is indicated by the inclusion of ρ_1 in the effect set of the conclusion. To see why this condition is necessary, consider the code below which might not be permitted to access the pointer `ptr` but nevertheless is able dereference it by generating a new key-pair, enabling, and associating `ptr` with it. (The form `open` x `=` e unpacks the existential package e.)

```
open akeypair = newkey;
grant gKey(akeypair) in {
  ptr_casted = associate ptr with lKey(akeypair);
  ! ptr_casted;    /* ptr accessed */
}
```

This code would type check if ρ_1 were missing from the effect set.

2.3 Preventing Resource Forging

Pointers, i.e. store locations, are naturally unforgeable in a strongly typed language that does not admit explicit pointer manipulations such as casting an integer to a memory address. In our system, keys are also unforgeable. But many other resources are forgeable. For example, any program can duplicate any pure function consisting only of variables and λ's by simply having another instance of the function.

[2] We can also allow associating via grant keys and key-pairs themselves, as used in the example in Section 1. However, the ability to associate via limit keys is important, as it eliminates the need to expose grant keys (either alone or in key-pairs) for this purpose.

Our system does not automatically prevent such resource forging. This sounds unsafe because if a resource is forgeable then any code may re-create it and use the resource. Consider the file access control policy from Section 1. If file resource are strings containing the actual file name in the file system, then in any language with normal string operations it is trivial to create from scratch any file name with no access controls whatsoever.

Fortunately, such situations usually can be remedied easily. In the file access control example, we can define file resources to have an *abstract data type* ([10]) `File` defined in a program component `File Manager`. The representation of a `File` could be a string type, but this would not be visible outside of the `File Manager` component. In this design, `File Manager` would handle all file operations directly via system calls, and all other components would be required to go through `File Manager`. This can be made possible by using the access control mechanism to ensure that only `File Manager` has access to file-related system calls. When requesting some file for the first time, a component must ask `File Manager` for the appropriate `File` resource referring to the requested file. We also give `File Manager` a type signature such that, for example, calling the `File` writing function would have the effect of accessing the given `File` resource. Then `File Manager` component can ensure that `File` resources are unforgeable so as to allow access control on files by associating `File` resources with key-pairs.

In our system, we assume that a programmer takes necessary steps to make resources unforgeable if he wishes to have them under access control. For a naturally unforgeable resource like a pointer or a key, this can be as simple as associating the resource with some key-pair shortly after creating it and before making it available to other parts of the program. For other resources, use of abstract data types may be necessary, as seen in the `File` example.

2.4 Simple Example

The system described thus far is capable of expressing some fine-grain access control policies. Consider the following code which, for each node of the linked list `g`, applies the function `p` to the `item` field of the node.

```
t := g;
while (t ≠ nil) {
  p(t.item);
  t := (! t).next;
}
```

Let $\rho \neq \perp$. Let `g` have the record type `NodeType` defined below. (We take the liberty of using a C-like structure declaration for simplicity.)

```
NodeType = ρ {
  ρ ItemType item;
  ρ ref(NodeType) next;
};
```

(The first occurrence of ρ is not a binding name; it just qualifies the record type.)

We would like to use the system to limit `p` to accessing only the passed `t.item`. In particular, we want to prevent `p` from modifying the list structure of `g`, because the code is traversing it. We may do this by associating each `t.item` with a fresh key before calling `p`.

```
t := g;
while (t ≠ nil) {
  open akey = newkey;
  grant gKey(akey) in {
    theitem = associate t.item with lKey(akey);
    limit lKey(akey) in { p(theitem); };
    t := (! t).next;
  }
}
```

Note that the function `p` could be complicated, potentially calling other functions. Nevertheless, the type system is able to ensure that each function application only accesses the specified `t.item`, because the lexically scoped name (which appears when the new key-pair is opened) distinguishes each instance of `akey` and its late binding with the resource.

This example shows the system's ability to declare late, local access control policies. That is, a programmer may store a collection of resources uniformly with the same type, and when there is a need for finer access control (e.g. at the level of individual resources in the collection) the key-associations are refined.

2.5 Subkeys and Subnaming

We now introduce subkeys, and their corresponding static representation using subnaming. These additions require a few changes to features explained earlier. To motivate the introduction of subkeys, we illustrate the problem that we would encounter if the system did not support them.

Consider the file access control example from Section 1. Recall that the key-pair `shared_files` is associated with the files shared between the program components `C` and `D`. Let $\rho_{\texttt{shared}}$ identify `shared_files`. Below, the code for `C` is re-written; in particular, it uses `newkey` instead of `newkey<shared_files` to generate the key-pair `some_files`.

```
[Component C]:
  open some_files = newkey;
  ...associate file_a with lKey(some_files);
  ...associate file_b with lKey(some_files);
  ...
  f = λx.limit lKey(some_files) in
         .../* accesses the selected files */
```

In the code above, `f` gets the type

$$\perp (\tau_1 \xrightarrow{\{\rho_{\texttt{some}}\}} \tau_2)$$

where $\rho_{\texttt{some}}$ identifies the key-pair `some_files`. Here, the set above the arrow is the latent effect of the function, as usual in a type and effect system. Because $\rho_{\texttt{some}}$ is lexically scoped in `C`'s context and therefore invisible in `D`'s context, the type system fails to type check `D`'s call to `f`.

The problem stems from the inability of the type system to understand local associations outside of the local context when the lexically scoped names are completely anonymous. This prevents non-local enforcement of locally refined policies.

Our system solves this problem by allowing the programmer to declare subset relations between resources via subkeys. The syntax form `newkey<`e generates a new immediate subkey-pair of an existing key-pair. As with `associate`, we let e be just the limit key part of the key-pair; again, this avoids the need to propagate the grant key just to associate resources with the key-pair. The type checking rule for `newkey<`e is

$$\frac{\Gamma \vdash e : \rho_1 \ \mathit{lkey}\,(\rho_2); L}{\Gamma \vdash \texttt{newkey<}e : \perp (\exists \rho_3 < \rho_2 . \perp \langle \perp \mathit{lkey}\,(\rho_3), \perp \mathit{gkey}\,(\rho_3) \rangle); L \cup \{\rho_1\}}$$

Here, in the bounded existential raw type $\exists \rho_3 < \rho_2 \ldots$, the key-pair name ρ_3 is the bound name and ρ_2 is the upper-bound free in the raw type. The intuition behind this rule is that the new key-pair should be identified by some fresh name ρ_3 such that $\rho_3 < \rho_2$. The subnaming relation $<$ statically keeps track of the subkey hierarchy.

Recall that any resources associated with a key-pair are also associated with its superkey-pairs. Thus resources associated with a key-pair are a subset of its superkey-pair's resources. Then, because the resources associated with the key-pair $\rho_{\texttt{some}}$ are a subset of those associated with the key-pair $\rho_{\texttt{shared}}$, we can use `shared_files` as the superkey-pair when generating `some_files`. Here is the code from Section 1 again (using limit keys to generate and associate).

```
[Component C]:
  open some_files = newkey<lKey(shared_files);
  ...associate file_a with lKey(some_files);
  ...associate file_b with lKey(some_files);
  ...
  f = λx.limit lKey(some_files) in
        .../* accesses the selected files */
```

Subnaming induces *subeffecting* and *subtyping*. In the example, the type system can reason that $\{\rho_{\texttt{some}}\} < \{\rho_{\texttt{shared}}\}$. We add the usual type checking rules so that a subeffect (resp. subtype) may be used where its supereffect (resp. supertype) is expected. Here we have

$$\perp (\tau_1 \xrightarrow{\{\rho_{\texttt{some}}\}} \tau_2) < \perp (\tau_1 \xrightarrow{\{\rho_{\texttt{shared}}\}} \tau_2)$$

and hence `f` can be called by a context having access to $\rho_{\texttt{shared}}$. Now `D`'s call to `f` type checks.

Without subkeys and subnaming, the type system cannot track a key/resource association outside of the context of the code in which the association is made.[3] The key observation that leads to subkeys and subnaming is that locally associated resources (i.e., a subkey) can be viewed as a local refinement of some larger collection of resources that is recognized (e.g., is associated with a superkey) outside the local context. In this way, subkeys and subnaming allow the programmer to define late, local associations that can be used non-locally.

To better understand the uses of subnaming, we present a larger example. Consider an airline ticketing program that contains two components: `FlightCtrl` having control of ticketing and seating information for all flights and `TicketAgt` representing a ticket agent. There is only one `FlightCtrl` but there could be more than instance of `TicketAgt`.

Suppose each component runs its own execution thread and that only read access to the seating and ticketing information is available to the `TicketAgt` thread, but `TicketAgt` may request `FlightCtrl` make modifications to the ticketing and seating arrangement. Let information for each flight be represented by the bounded existential record type

```
FlightInfo = ∃ρ1<ρall_flights {
  ⊥ lkey(ρ1) key;
  ρ1 TicketingInfo tickets;
  ρ1 SeatingInfo seats;
  ρ2 int flight_number;
  ρ2 ref(FlightInfo) next;
};
```

The `FlightCtrl` thread is granted access to $\rho_{\texttt{all\_flights}}$, so it may access the `FlightInfo` record of every flight.

Recognizing `TicketAgt`'s request, perhaps after some authentication, `FlightCtrl` responds by retrieving the flight information for the flight being requested:

```
[FlightCtrl Component]:
  ...
  if (AuthTicketAgt(agent1, flight_num) {
    open finfo = getFlightInfo(flight_num);
    ...
```

The `TicketAgt` thread does not have write access to the `tickets` or `seatings` fields of `finfo`, so it cannot make the changes by itself. But it may create a function that does the job when called by `FlightCtrl`:

[3] To be precise, this statement is true up to absence of grant keys. In fact, without subnaming, the grant keys must be used to grant unconditional access to the non-local context when using associations non-locally. Therefore not having subnaming precludes many useful stack-based access control patterns.

```
[TicketAgt]:
  ...
  f_agent = λx.
    .../* Makes changes to
          finfo.tickets and finfo.seats */
```

FlightCtrl runs f_agent in its thread. But FlightCtrl, not fully trusting f_agent, uses the access control mechanism to check that f_agent is actually limited to modifying the seating and ticketing information of the specified flight:

```
[FlightCtrl Component]:
  ...
  limit finfo.key in f_agent()
```

The type system guarantees that f_agent does not gain unwanted accesses (such as write access to flight information for other flights).

The subnaming relation $\rho_1 < \rho_{\texttt{all\_flights}}$ implied in the type of FlightInfo is important. Existential quantification allows the program to operate on one FlightInfo at a time. Without subnaming, the bound name ρ_1 would become completely anonymous when the existential package is opened. Then, because this would mean that ρ_1 is a name unknown to FlightCtrl, FlightCtrl would be forbidden from accessing finfo, and hence would be unable to run f_agent. Subnaming allows anybody with access to $\rho_{\texttt{all\_flights}}$ to perform access control at the level of individual FlightInfo instances.

We can take this example a step further. Suppose that TicketAgt also has its own set of resources associated with the key-pair $\rho_{\texttt{agt}}$. Assume that f_agent accesses some of these resources and that FlightCtrl has access rights to $\rho_{\texttt{agt}}$.

Let ThirdPartyLib be a third party library component that TicketAgt uses to, say, help itself calculate ticket costs and print out information on the terminal screen. To let ThirdPartyLib use some of its resources in a controlled manner, TicketAgt imposes access control on its own resources:

```
[TicketAgt]:
  ...
  open lib_key = newkey;
  ...associate agt_resources.a with lKey(lib_key);
  ...associate agt_resources.b with lKey(lib_key);
```

Let $\rho_{\texttt{lib}}$ identify the key-pairs lib_key. Now suppose that f_agent runs functions from ThirdPartyLib. The programmer wishes to limit the resources the functions can use, so the definition of f_agent would look like

```
[TicketAgt]:
  ...
  f_agent = λx. {
    ...
    limit lKey(lib_key) in {
      .../* calls ThirdPartyLib's functions */ }
    ...
  }
```

Coming back to FlightCtrl, it seems reasonable to assume that the FlightCtrl thread is still capable of running f_agent because it has accesses to TicketAgt's resources as well as the flight information.

```
[FlightCtrl]:
  ...
  /* agt_key is the key-pair identified by ρ agt */
  limit finfo.key, lKey(agt_key) in f_agent()
```

$\rho \in \text{Names} \cup \{\bot, \top\} \quad x \in \text{Vars} \quad m \in \text{Integers} \quad L \subseteq_{\text{finite}} \text{Names} \cup \{\bot, \top\}$

$$
\begin{array}{llll}
\text{raw types} & \sigma & ::= & \mathit{int} \mid \langle \tau_1, \tau_2, \ldots, \tau_n \rangle \mid \mathit{ref}(\tau) \mid \tau_1 \xrightarrow{L} \tau_2 \mid \forall \rho_1 < \rho_2.\tau \mid \exists \rho_1 < \rho_2.\tau \mid \mathit{lkey}(\rho) \mid \mathit{gkey}(\rho) \\
\text{types} & \tau & ::= & \rho\,\sigma \\
\text{values} & v & ::= & m \mid \texttt{Top} \mid x \mid \langle v_1, v_2, \ldots, v_n \rangle \mid \lambda x{:}\tau.e \mid \Lambda \rho_1 < \rho_2.v \mid \texttt{pack}\ v\ \texttt{as}\ \exists \rho_2 < \rho_3.\tau \\
\text{expressions} & e & ::= & v \mid \langle e_1, e_2, \ldots, e_n \rangle \mid e.i \mid e_1\, e_2 \mid \texttt{ref}\ e \mid e_1 \texttt{ := } e_2 \mid \texttt{!}\ e \mid \texttt{spawn}\ e \mid \\
& & & e\,[\rho] \mid \texttt{pack}\ e\ \texttt{as}\ \exists \rho_2 < \rho_3.\tau \mid \texttt{open}\ x \texttt{ = } e_1\ \texttt{as}\ \tau\ \texttt{in}\ e_2 \mid \\
& & & \texttt{limit}\ e_1, e_2, \ldots, e_{n-1}\ \texttt{in}\ e_n \mid \texttt{grant}\ e_1\ \texttt{in}\ e_2 \mid \texttt{newkey}{<}e \mid \texttt{associate}\ e_1\ \texttt{with}\ e_2
\end{array}
$$

Figure 1: Source language

But because we did not create `lib_key` as subkeys, the type system does not see the associations made in the `TicketAgt` component. Indeed, the type system must give `f_agent` the type $\bot\ (\tau_1 \xrightarrow{L} \tau_2)$ where $\rho_{\texttt{lib}} \in L$, but $\rho_{\texttt{lib}}$ is not visible to `FlightCtrl`'s context.

Fortunately, subkeys provide an easy one-line fix.

```
[TicketAgt]:
  ...
  open lib_key = newkey<lKey(agt_key); /* fixed */
  ...associate agt_resources.a with lKey(lib_key);
  ...associate agt_resources.b with lKey(lib_key);
```

Now the type system may give `f_agent` the type $\bot\ (\tau_1 \xrightarrow{L'} \tau_2)$ where L' contains $\rho_{\texttt{agt}}$ instead of $\rho_{\texttt{lib}}$. Then the system is able to type check the call to `f_agent` in the `FlightCtrl` thread.

3 Formal System

We next formally present the core subset of our access control system and prove its soundness.

Figure 1 shows the source language, which consists of call-by-value first-class functions with primitives for imperative store operations and concurrency. The language is rather spare, missing a few bells and whistles seen in earlier sections for brevity (e.g., `while` and record types). These features can easily be added. We briefly explain the syntax.

As before, each type τ consists of a *name* and a *raw type*. The raw types are integers *int*, tuples $\langle \tau_1, \tau_2, \ldots, \tau_n \rangle$, pointers $\mathit{ref}(\tau)$, functions $\tau_1 \xrightarrow{L} \tau_2$, bounded universal types $\forall \rho_1 < \rho_2.\tau$, bounded existential types $\exists \rho_1 < \rho_2.\tau$, limit keys $\mathit{lkey}(\rho)$, and grant keys $\mathit{gkey}(\rho)$. In bounded quantified types, the name on the left of $<$ is the bound name and the name on the right is the upper-bound free in the raw type.

A program value v is either an integer m, the constant limit key `Top`, a variable x, a tuple $\langle v_1, v_2, \ldots, v_n \rangle$, a function $\lambda x : \tau.e$, a bounded polymorphic abstraction $\Lambda \rho_1 < \rho_2.v$, or a bounded existential package $\texttt{pack}\ v\ \texttt{as}\ \exists \rho_2 < \rho_3.\tau$.

A program expression e is either a value or one of the following syntax forms. The expression $\langle e_1, e_2, \ldots, e_n \rangle$ creates a tuple. The expression $e.i$ projects the ith value of the tuple e. The expression $e_1\, e_2$ applies the function e_1 to e_2. The expression $\texttt{ref}\ e$ allocates a new store location and initializes it to e. The expression $e_1 \texttt{:=} e_2$ assigns e_2 to the store location e_1. The expression $\texttt{!}\,e$ dereferences the pointer (i.e. the store location) e. The expression $\texttt{spawn}\ e$ spawns a new thread for the evaluation of e. The expression $e\,[\rho]$ instantiates the bounded universal abstraction e. The expression $\texttt{pack}\ e\ \texttt{as}\ \exists \rho_2 < \rho_3.\tau$ creates a bounded existential package of e. The expression $\texttt{open}\ x \texttt{ = } e_1\ \texttt{as}\ \tau\ \texttt{in}\ e_2$ unpacks the bounded existential package e_1 to evaluate e_2. Section 2 introduced the syntax forms $\texttt{limit}\ e_1, e_2, \ldots, e_{n-1}\ \texttt{in}\ e_n$, $\texttt{grant}\ e_1\ \texttt{in}\ e_2$, $\texttt{newkey} < e$, and $\texttt{associate}\ e_1\ \texttt{with}\ e_2$.

Expressions and types are equivalent up to consistent renaming of bound names and variables.

$$\frac{}{\Gamma \vdash m : \bot\ \mathit{int}; \emptyset}\ (\text{Int}) \qquad \frac{}{\Gamma \vdash \texttt{Top} : \bot\ \mathit{lkey}\,(\top); \emptyset}\ (\text{Top}) \qquad \frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau; \emptyset}\ (\text{Var})$$

$$\frac{\text{For } 1 \le i \le n,\ \Gamma \vdash e_i : \tau_i; L_i}{\Gamma \vdash \langle e_1, e_2, \ldots, e_n \rangle : \bot\ \langle \tau_1, \tau_2, \ldots, \tau_n \rangle; \bigcup_{1 \le i \le n} L_i}\ (\text{Tuple}) \qquad \frac{\Gamma \vdash e : \rho\ \langle \tau_1, \tau_2, \ldots, \tau_n \rangle; L}{\Gamma \vdash e.i : \tau_i; L \cup \{\rho\}}\ (\text{Proj})$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2; L \qquad \Gamma, x{:}\tau_1 \vdash \diamond}{\Gamma \vdash \lambda x{:}\tau_1.e : \bot\ (\tau_1 \xrightarrow{L} \tau_2); \emptyset}\ (\text{Fun}) \qquad \frac{\Gamma \vdash e_1 : \rho\ (\tau_1 \xrightarrow{L_1} \tau_2); L_2 \qquad \Gamma \vdash e_2 : \tau_1; L_3}{\Gamma \vdash e_1\ e_2 : \tau_2; L_1 \cup L_2 \cup L_3 \cup \{\rho\}}\ (\text{App})$$

$$\frac{\Gamma \vdash e : \tau; L}{\Gamma \vdash \texttt{ref}\ e : \bot\ \mathit{ref}\,(\tau); L}\ (\text{Ref}) \qquad \frac{\Gamma \vdash e_1 : \rho\ \mathit{ref}\,(\tau); L_1 \qquad \Gamma \vdash e_2 : \tau; L_2}{\Gamma \vdash e_1 \texttt{ := } e_2 : \tau; L_1 \cup L_2 \cup \{\rho\}}\ (\text{Assign}) \qquad \frac{\Gamma \vdash e : \rho\ \mathit{ref}\,(\tau); L}{\Gamma \vdash \texttt{!}\ e : \tau; L \cup \{\rho\}}\ (\text{Deref})$$

$$\frac{\Gamma \vdash e : \tau; \emptyset}{\Gamma \vdash \texttt{spawn}\ e : \bot\ \mathit{int}; \emptyset}\ (\text{Spawn})$$

$$\frac{\Gamma, \rho_1 {<} \rho_2 \vdash v : \tau; \emptyset \qquad \Gamma, \rho_1 {<} \rho_2 \vdash \diamond}{\Gamma \vdash \Lambda \rho_1 {<} \rho_2.v : \bot\ (\forall \rho_1 {<} \rho_2.\tau); \emptyset}\ (\text{Gen}) \qquad \frac{\Gamma \vdash e : \rho_1\ (\forall \rho_2 {<} \rho_3.\tau); L \qquad \Gamma \vdash \rho_4 < \rho_3}{\Gamma \vdash e\,[\rho_4] : \tau[\rho_4/\rho_2]; L \cup \{\rho_1\}}\ (\text{Inst})$$

$$\frac{\Gamma \vdash e : \tau[\rho_3/\rho_1]; L \qquad \Gamma \vdash \rho_3 < \rho_2}{\Gamma \vdash \texttt{pack}\ e\ \texttt{as}\ \exists \rho_1 {<} \rho_2.\tau : \bot\ (\exists \rho_1 {<} \rho_2.\tau); L}\ (\text{Pack}) \qquad \frac{\begin{array}{c}\Gamma \vdash e_1 : \rho_1\ (\exists \rho_2 {<} \rho_3.\tau_1); L_1 \qquad \Gamma, \rho_2 {<} \rho_3, x{:}\tau_1 \vdash e_2 : \tau_2; L_2 \\ \Gamma \vdash \tau_2 \qquad \Gamma \vdash L_2 \qquad \Gamma, \rho_2 {<} \rho_3, x{:}\tau_1 \vdash \diamond\end{array}}{\Gamma \vdash \texttt{open}\ x \texttt{ = } e_1\ \texttt{as}\ \tau_1\ \texttt{in}\ e_2 : \tau_2; L_1 \cup L_2 \cup \{\rho_1\}}\ (\text{Open})$$

$$\frac{\begin{array}{c}\text{For } 1 \le i \le n-1,\ \Gamma \vdash e_i : \rho_i'\ \mathit{lkey}\,(\rho_i); L_i \\ \Gamma \vdash e_n : \tau_n; L_n \qquad \Gamma \vdash L_n < \{\rho_1, \rho_2, \ldots, \rho_{n-1}\}\end{array}}{\Gamma \vdash \texttt{limit}\ e_1, e_2, \ldots, e_{n-1}\ \texttt{in}\ e_n : \tau_n; L_n \cup \bigcup_{1 \le i \le n-1} (L_i \cup \{\rho_i'\})}\ (\text{Limit}) \qquad \frac{\begin{array}{c}\Gamma \vdash e_1 : \rho_1\ \mathit{gkey}\,(\rho_2); L_1 \\ \Gamma \vdash e_2 : \tau; L_2 \cup \{\rho_2\}\end{array}}{\Gamma \vdash \texttt{grant}\ e_1\ \texttt{in}\ e_2 : \tau; L_1 \cup L_2 \cup \{\rho_1\}}\ (\text{Grant})$$

$$\frac{\Gamma \vdash e : \rho_1\ \mathit{lkey}\,(\rho_2); L}{\Gamma \vdash \texttt{newkey}{<}e : \bot\ (\exists \rho_3 < \rho_2.\bot\ \langle \bot\ \mathit{lkey}\,(\rho_2), \bot\ \mathit{gkey}\,(\rho_2) \rangle); L \cup \{\rho_1\}}\ (\text{NewKey})$$

$$\frac{\Gamma \vdash e_1 : \rho_1\ \sigma; L_1 \qquad \Gamma \vdash e_2 : \rho_2\ \mathit{lkey}\,(\rho_3); L_2}{\Gamma \vdash \texttt{associate}\ e_1\ \texttt{with}\ e_2 : \rho_3\ \sigma; L_1 \cup L_2 \cup \{\rho_1, \rho_2\}}\ (\text{Associate})$$

$$\frac{\Gamma \vdash e : \tau_1; L_1 \qquad \Gamma \vdash \tau_1 < \tau_2 \qquad \Gamma \vdash L_1 < L_2}{\Gamma \vdash e : \tau_2; L_2}\ (\text{Subsumption})$$

Figure 2: Type checking rules: expressions

$$\frac{\Gamma \vdash \rho}{\Gamma \vdash \rho < \rho}\ \text{(Sub Refl } \rho) \quad \frac{\Gamma \vdash \rho_1 < \rho_2 \quad \Gamma \vdash \rho_2 < \rho_3}{\Gamma \vdash \rho_1 < \rho_3}\ \text{(Sub Trans } \rho) \quad \frac{\rho_1 {<} \rho_2 \in \Gamma}{\Gamma \vdash \rho_1 < \rho_2}\ \text{(Sub } \rho) \quad \frac{\Gamma \vdash \rho}{\Gamma \vdash \bot < \rho}\ \text{(Sub } \bot)$$

$$\frac{\Gamma \vdash \rho_1 < \rho_2 \quad \Gamma \vdash \sigma_1 < \sigma_2}{\Gamma \vdash \rho_1\ \sigma_1 < \rho_2\ \sigma_2}\ \text{(Sub } \tau) \quad \frac{\Gamma \vdash \sigma}{\Gamma \vdash \sigma < \sigma}\ \text{(Sub Refl } \sigma) \quad \frac{\Gamma \vdash \sigma_1 < \sigma_2 \quad \Gamma \vdash \sigma_2 < \sigma_3}{\Gamma \vdash \sigma_1 < \sigma_3}\ \text{(Sub Trans } \sigma)$$

$$\frac{\text{For } 1 \le i \le n,\ \Gamma \vdash \tau_i < \tau_i{}'}{\Gamma \vdash \langle \tau_1, \tau_2, \ldots, \tau_n \rangle < \langle \tau_1{}', \tau_2{}', \ldots, \tau_n{}' \rangle}\ \text{(Sub Tuple)} \quad \frac{\Gamma \vdash \tau_3 < \tau_1 \quad \Gamma \vdash \tau_2 < \tau_4 \quad \Gamma \vdash L_1 < L_2}{\Gamma \vdash \tau_1 \xrightarrow{L_1} \tau_2 < \tau_3 \xrightarrow{L_2} \tau_4}\ \text{(Sub Fun)}$$

$$\frac{\Gamma \vdash \rho_3 < \rho_2 \quad \Gamma, \rho_1 {<} \rho_3 \vdash \tau_1 < \tau_2 \quad \Gamma, \rho_1 {<} \rho_3 \vdash \diamond}{\Gamma \vdash (\forall \rho_1 {<} \rho_2 . \tau_1) < (\forall \rho_1 {<} \rho_3 . \tau_2)}\ \text{(Sub Univ)} \quad \frac{\Gamma \vdash \rho_2 < \rho_3 \quad \Gamma, \rho_1 {<} \rho_2 \vdash \tau_1 < \tau_2 \quad \Gamma, \rho_1 {<} \rho_2 \vdash \diamond}{\Gamma \vdash (\exists \rho_1 {<} \rho_2 . \tau_1) < (\exists \rho_1 {<} \rho_3 . \tau_2)}\ \text{(Sub Exists)}$$

$$\frac{\Gamma \vdash L}{\Gamma \vdash \emptyset < L}\ \text{(Sub Effect } \emptyset) \quad \frac{\Gamma \vdash \rho_1 < \rho_2 \quad \Gamma \vdash L_1 < L_2}{\Gamma \vdash L_1 \cup \{\rho_1\} < L_2 \cup \{\rho_2\}}\ \text{(Sub Effect } \{\rho\})$$

Figure 3: Type checking rules: subnaming, subtyping, subeffecting

$$\frac{}{\Gamma \vdash \bot}\ \text{(Name } \bot) \quad \frac{}{\Gamma \vdash \top}\ \text{(Name } \top) \quad \frac{\rho_1 {<} \rho_2 \in \Gamma}{\Gamma \vdash \rho_1}\ \text{(Name } \rho)$$

$$\frac{\Gamma \vdash \rho \quad \Gamma \vdash \sigma}{\Gamma \vdash \rho\ \sigma}\ \text{(Type)} \quad \frac{}{\Gamma \vdash \mathit{int}}\ \text{(RType Int)} \quad \frac{\text{For } 1 \le i \le n,\ \Gamma \vdash \tau_i}{\Gamma \vdash \langle \tau_1, \tau_2, \ldots, \tau_n \rangle}\ \text{(RType Tuple)}$$

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \mathit{ref}(\tau)}\ \text{(RType Ref)} \quad \frac{\Gamma \vdash \tau_1 \quad \Gamma \vdash \tau_2 \quad \Gamma \vdash L}{\Gamma \vdash \tau_1 \xrightarrow{L} \tau_2}\ \text{(RType Fun)}$$

$$\frac{\Gamma, \rho_1 {<} \rho_2 \vdash \tau \quad \Gamma, \rho_1 {<} \rho_2 \vdash \diamond}{\Gamma \vdash \forall \rho_1 {<} \rho_2 . \tau}\ \text{(RType Univ)} \quad \frac{\Gamma, \rho_1 {<} \rho_2 \vdash \tau \quad \Gamma, \rho_1 {<} \rho_2 \vdash \diamond}{\Gamma \vdash \exists \rho_1 {<} \rho_2 . \tau}\ \text{(RType Exists)}$$

$$\frac{\Gamma \vdash \rho}{\Gamma \vdash \mathit{lkey}(\rho)}\ \text{(RType LKey)} \quad \frac{\Gamma \vdash \rho}{\Gamma \vdash \mathit{gkey}(\rho)}\ \text{(RType GKey)} \quad \frac{\text{For each } \rho \in L,\ \Gamma \vdash \rho}{\Gamma \vdash L}\ \text{(Effect)}$$

$$\frac{}{\emptyset \vdash \diamond}\ \text{(Env } \emptyset) \quad \frac{\Gamma \vdash \tau \quad \Gamma \vdash \diamond \quad x \notin \mathit{dom}(\Gamma)}{\Gamma, x{:}\tau \vdash \diamond}\ \text{(Env } x) \quad \frac{\Gamma \vdash \rho_2 \quad \Gamma \vdash \diamond \quad \rho_1 \notin \mathit{dom}(\Gamma) \cup \{\bot, \top\}}{\Gamma, \rho_1 {<} \rho_2 \vdash \diamond}\ \text{(Env } \rho)$$

Figure 4: Type checking rules: well-formed names, types, effects, environments

3.1 Static Semantics

Type environments are sequences of variable-to-type bindings and names annotated with a supername.

$$\Gamma ::= \emptyset \mid \Gamma, x{:}\tau \mid \Gamma, \rho_1{<}\rho_2$$

Figure 2 shows type checking rules for expressions. The meaning of $\Gamma \vdash e : \tau; L$ is that the effect set L conservatively approximates e's access rights. A *program* is a closed expression. A program e is well-typed iff $\emptyset \vdash e : \tau; \emptyset$ for some τ, i.e., iff e can be type checked with no enabled access rights. Let us now discuss some of the important points in the type checking rules.

As explained in Section 2.5, (NewKey) assigns a bounded existential type to a newly created key-pair. The constant `Top` serves as the initial key. (Top) assigns `Top` the limit-key type $\perp$ *lkey* $(\top)$. The fact that `Top` is not a grant key is important, as otherwise any code would be able to access any resource by granting itself `Top`. The syntax form `newkey` is equivalent to `newkey<Top`.

Subnaming induces subtyping and subeffecting (see Figure 3), which are then used in two places: (Subsumption) and (Limit). (Subsumption) allows a subtype (resp. subeffect) to be used wherever its supertype (resp. supereffect) is expected. (Subsumption) formally captures the intention that whatever is associated with key-pairs are also associated with its super-key-pairs.

(Limit) replaces the rule introduced in Section 2 by using the subeffecting relation $<$ instead of the subset relation. This is needed to allow contexts with access rights limited to some key-pair to execute a code with access rights limited to its superkey-pair when the code accesses only what are allowed by the context. For example, consider the following function `g` which takes a resource and accesses it. We wish to limit `g`'s access rights to the key-pair `g_key` identified by $\rho_{\mathtt{g}}$.

$$\mathtt{g} = \Lambda\rho{<}\rho_{\mathtt{g}}.\lambda x{:}\rho\ \sigma.\mathtt{limit\ lKey(g\_key)\ in}\ \ldots$$

The type system can give the `g` the raw type

$$\forall\rho{<}\rho_{\mathtt{g}}.(\rho\ \sigma \xrightarrow{\{\rho\}} \tau)$$

for some σ and τ so that a caller whose access rights are locally refined with respect to $\rho_{\mathtt{g}}$ is able call `g` with its resource.

Figure 3 also shows that, as usual, subtyping is invariant under pointer types. Subtyping is also invariant for raw key types, i.e., *lkey* $(\rho_1) <$ *lkey* (ρ_2) then $\rho_1 = \rho_2$, and analogously for grant key types. It is easy to see that covariant keys would be unsound because limiting (resp. granting) ρ_1 should not somehow limit (resp. grant) ρ_2 when $\rho_1 < \rho_2$ and $\rho_1 \neq \rho_2$. To see that contravariance also fails, observe that associating ρ_1 should not somehow associate ρ_2 when $\rho_2 < \rho_1$ and $\rho_1 \neq \rho_2$.[4]

Section 2 discusses (Grant) and (Associate). Recall that values are created with no associated key-pair. Thus the rules for introductory forms (Int), (Tuple), (Fun), (Ref), (Gen), (Pack) and (NewKey) all qualify their values with $\perp$. The corresponding eliminatory forms (Proj), (App), (Assign), (Deref), (Inst), (Open), (Limit), (Grant), (NewKey) and (Associate) that use the values require an effect set consistent with the required access rights. Note that $\Gamma \vdash \perp < \rho$ for any $\Gamma \vdash \rho$.

As usual in a type and effect system, the latent effect of a function is recorded in its type at (Fun) and used at (App).

(Spawn) implies that a newly spawned thread starts without any access rights. We could alternatively have the spawned thread inherit the access rights of the spawner. In this case, we would add the rule

$$\frac{\Gamma \vdash e : \tau; L}{\Gamma \vdash \mathtt{spawn}\, e : \perp\ \mathit{int}; L}\ \text{(Spawn alt)}$$

The only non-syntax directed rule is (Subsumption). Effect sets are finite and subtyping is structural, so type checking is decidable.

[4] We could relax invariance by assigning a pair of a covariant name and a contravariant name for each key type. Here we stick with invariance for simplicity.

3.2 Sketch of Type Soundness

We sketch a proof of soundness. A complete version of the proof appears in Appendix A.

The first step is to define a dynamic semantics rich enough to classify an access violation as a run-time error. This is only to prove soundness of the system; a real implementation should follow a straightforward dynamic semantics that ignores access control checks.

We introduce the set KeyPairs such that each element of KeyPairs denotes a key-pair. Each program value is now annotated with a key-pair, with the intended meaning that the value is associated with the key-pair. So a program value, say the function $\lambda x{:}\tau.e$, associated with the key-pair $k \in$ KeyPairs is

$$k\ \lambda x{:}\tau.e$$

We type check the annotated values by keeping track of names identifying $k \in$ KeyPairs in the type environment.

$$\Gamma ::= \ldots \mid k{:}\rho$$

We use the special key-pairs `bot` and `top` to denote non-association and the `Top` key-pair, respectively.

The dynamic semantics needs to track enabled accesses for each context. Like in an eager stack-inspection semantics, we maintain exactly the enabled accesses in the syntax form `access` A `in` e where $A \subseteq_{\text{finite}}$ KeyPairs $\cup \{\texttt{bot}, \texttt{top}\}$ denote the enabled accesses for the expression e. This approach makes the soundness proof easy. The following rule is used to type check `access` A `in` e.

$$\frac{\Gamma \vdash e : \tau; L \qquad \Gamma \vdash L < \{\rho \mid \Gamma \vdash k : \rho \text{ for some } k \in A\}}{\Gamma \vdash \texttt{access}\ A\ \texttt{in}\ e : \tau; \emptyset}$$

`access` A `in` e is a run-time syntax form not available to the source program. Each `access` A `in` e appears as the result of the evaluation of `limit` or `grant`.

Evaluation is defined as a sequence of call-by-value small-step reductions of the form

$$(S, K, \langle e_1, e_2, \ldots, e_n\rangle) \longrightarrow (S', K', \langle e'_1, e'_2, \ldots, e'_{n'}\rangle)$$

where $e_1, e_2, \ldots, e_n, e'_1, e'_2, \ldots, e'_{n'}$ are the execution threads, S, S' are stores mapping store locations to program values, and K, K' are key-managers representing immediate sub-key-pair relation between key-pairs. So k_1 is a (immediate or non-immediate) sub-key-pair of k_2 iff $k_1 \mapsto k_2 \in K^*_\perp$ where $K^*_\perp$ is the reflexive, transitive closure of K lifted with bottom. In this section, we restrict the reduction rules to the case for one execution thread to save space.

To make evaluation fail at access violations, the dynamic semantics checks the enabled accesses whenever a program value is used. For example, when calling a function associated with the key-pair k, the dynamic semantics checks whether k is enabled in the current set of access rights A.

$$\frac{k \in \mathit{Close}_K(A)}{\begin{array}{l}(S, K, E[\texttt{access}\ A\ \texttt{in}\ R[(k\ \lambda x{:}\tau.e)\ v]]) \\ \qquad\qquad \longrightarrow (S, K, E[\texttt{access}\ A\ \texttt{in}\ R[e[v/x]]])\end{array}}$$

Close_K is a function such that $k \in \mathit{Close}_K(A)$ iff k is a sub-key-pair of some key-pair in A in the key-pair relation K. If the condition $k \in \mathit{Close}_K(A)$ is false, then this reduction cannot be taken, which implies that the evaluation for this thread gets stuck, indicating a run-time error.

The evaluation contexts E and R are used to find the redex and the set representing the enabled accesses for the redex. Intuitively, $E[\texttt{access}\ A\ \texttt{in}\ R[e]]$ means that e is in an evaluation context and A is the set of its enabled accesses.

We use the special constant value `err` to signal a run-time error. Whenever an execution thread gets stuck, i.e., when it reduces to a irreducible non-value, it immediately reduces to `err`. Note that `err` has no type.

As mentioned before, `limit` and `grant` introduce occurrences of `access` A `in` e. The new access rights B after `limit` is reduced is the intersection of the previous access rights A with those specified in `limit`.

(Here $\texttt{lkey}\,(k)$ denotes the limit key value (unannotated) of the key-pair k.)

$$\frac{\begin{array}{c}\{k'_1, k'_2, \ldots, k'_n\} \subseteq Close_K(A) \\ B = Close_K(A) \cap Close_K(\{k_1, \ldots, k_n\})\end{array}}{\begin{array}{l}(S, K, E[\texttt{access}\ A\ \texttt{in}\ R[\\ \quad \texttt{limit}\ k'_1\ \texttt{lkey}\,(k_1), \ldots, k'_n\ \texttt{lkey}\,(k_n)\ \texttt{in}\ e]]) \longrightarrow \\ \quad (S, K, E[\texttt{access}\ A\ \texttt{in}\ R[\texttt{access}\ B\ \texttt{in}\ e]])\end{array}}$$

In contrast, the new access rights B after `grant` is reduced is the union of the previous access rights A with those specified in `grant`. (Here $\texttt{gkey}\,(k)$ denotes the grant key value (unannotated) of the key-pair k.)

$$\frac{\begin{array}{c}k_1 \in Close_K(A) \\ B = A \cup \{k_2\}\end{array}}{\begin{array}{l}(S, K, E[\texttt{access}\ A\ \texttt{in}\ R[\texttt{grant}\ k_1\ \texttt{gkey}\,(k_2)\ \texttt{in}\ e]]) \\ \quad \longrightarrow (S, K, E[\texttt{access}\ A\ \texttt{in}\ R[\texttt{access}\ B\ \texttt{in}\ e]])\end{array}}$$

The reduction for `associate` is relatively simple. Here r is an unannotated program value. The reduction changes r's annotation from k_1 to k_3.

$$\frac{\{k_1, k_2\} \subseteq Close_K(A)}{\begin{array}{l}(S, K, E[\texttt{access}\ A\ \texttt{in}\ R[\\ \qquad \texttt{associate}\ k_1\ r\ \texttt{with}\ k_2\ \texttt{lkey}\,(k_3)]]) \\ \qquad\qquad \longrightarrow (S, K, E[\texttt{access}\ A\ \texttt{in}\ R[k_3\ r]])\end{array}}$$

The reduction for `newkey` creates a new key-pair k_3 as the immediate sub-key-pair of k_2.

$$\frac{k_1 \in Close_K(A) \qquad k_3 \notin dom(K) \cup \{\texttt{bot}, \texttt{top}\}}{\begin{array}{l}(S, K, E[\texttt{access}\ A\ \texttt{in}\ R[\texttt{newkey}{<}k_1\ \texttt{lkey}\,(k_2)]]]) \longrightarrow \\ \qquad (S, K \cup \{k_3 \mapsto k_2\}, E[\texttt{access}\ A\ \texttt{in}\ R[v]])\end{array}}$$

where v is the existential package containing the new key-pair value (not shown).

Appendix A.1 shows the complete dynamic semantics.

Recall that we have intentionally enriched the dynamic semantics with explicit key-pairs just so that we could identify access violations. A real implementation of the run-time system does not need key-pairs because our type system checks for access violations at compile time.

The next step is to define a well-typed program state.

Definition 1 $\Gamma \vdash (S, K, \langle e_1, e_2, \ldots, e_n\rangle)$ *(read "the program state* $(S, K, \langle e_1, e_2, \ldots, e_n\rangle)$ *is well-typed under* Γ*") iff*

(1) $\Gamma \vdash \diamond$.

(2) For each $1 \leq i \leq n$, $\Gamma \vdash e_i : \tau_i; \emptyset$ *for some* τ_i.

(3) $\ell \in dom(S)$ *iff* $\ell \in dom(\Gamma)$.

(4) $k \in dom(K)$ *iff* $k \in dom(\Gamma)$.

(5) $x \notin dom(\Gamma)$.

(6) If $\Gamma \vdash \ell : \tau$ *and* $S(\ell) = v$ *then* $\Gamma \vdash v : \tau; \emptyset$.

(7) Suppose $\Gamma \vdash k_1 : \rho_1$ *and* $\Gamma \vdash k_2 : \rho_2$. *Then* $\Gamma \vdash \rho_1 < \rho_2$ *iff* $k_1 \mapsto k_2 \in K^*_\perp$.

The first condition says that Γ must be well-formed. The second condition says that each execution thread is well-typed under Γ with effect set $\emptyset$. The third and the fourth conditions say that Γ and S, K must have matching store locations and key-pairs. The fifth condition says that Γ must not contain variables. The sixth condition expresses the usual well-typed store condition. The last condition says that the key-manager K contains exactly the sub-key-pair relationship implied by the type environment Γ.

Recall that `err` is non-typable. The soundness proof then reduces to showing the following theorem.

Theorem 1 (Subject Reduction) *If* $\Gamma \vdash (S, K, \vec{e})$ *and* $(S, K, \vec{e}) \longrightarrow (S', K', \vec{e}')$, *then there is* Γ' *such that* $\Gamma' \vdash (S', K', \vec{e})$.

The theorem is proved by induction on the type derivation and case analysis on reduction kinds. The immediate corollary is that a well-typed program does not cause access violations.

4 Extensions

We next discuss a few possible extensions to the system.

4.1 Run-Time Checks

Our system is powerful enough to statically enforce many fine-grain access control policies. But the programmer may still encounter situations where the static checking feels overly restrictive. We may extend our system with run-time checks.

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \rho_1 \ \mathit{lkey}\,(\rho_2); L_1 \\ \Gamma \vdash e_2 : \tau; L_2 \cup \{\rho_2\} \qquad \Gamma \vdash e_3 : \tau; L_2\end{array}}{\Gamma \vdash \texttt{have-access}\ e_1\ \texttt{?}\ e_2\ e_3 : \tau; L_1 \cup L_2}$$

The syntax form `have-access` e_1 `?` e_2 e_3 checks whether the current context has e_1 enabled, and if so evaluates e_2 or else evaluates e_3.

The disadvantage of this extension is that the run-time system now must track key values and enabled access rights. Without this extension, our system is completely static, and therefore a program with access control has no run-time overhead over an equivalent program without access control. Nevertheless, even with this extension, the run-time system only needs to check enabled access rights at `have-key`'s. Hence the overhead should be minimal.

4.2 Effect Kinds

In our airline ticketing example, we assumed that the `TicketAgt` thread has read access but not write access to the flight information. Strictly speaking, the system we described cannot distinguish different kinds of effects on the same resource.

It is easy to extend the system with effect kind constants for the primitives in the language (e.g., distinguishing the effect of dereferencing from the effect of assigning or associating) and programmer-defined effect kind variables (e.g., the effect kinds *readSeating*, *editSeating* for `SeatingInfo`). Effect kinds qualify effects, i.e., each effect is now of the form $\eta\ \rho$ where η is an effect kind. Then the programmer may, for example, limit e's access to only reading a selected `SeatingInfo` by

`limit` *readSeating*`(theseating_key) in` e

and the type system checks that e's effect set is a subeffect of $\{\mathit{readSeating}\ \rho\}$, where ρ identifies `theseating_key`.

4.3 Type Variables, Effect Variables, and More

We restricted the formal system to just bounded quantified types over names to make the presentation concise. But because the system relies only on standard type-system techniques, it is straightforward to extend the system with type variables and effect variables so as to admit bounded quantification over them.

Existentially quantified type variables naturally encode abstract data types [10]. We may also want a native support for parameterized recursive data types. A practical implementation may be to add recursively-defined named data types seen in languages like ML.

5 Relation to Must Alias Analysis

We briefly discuss the relation between our access control system and must alias analysis. In order to express locally refined access control policies, our access control checks constraints of the form "resources used by the expression e must-alias one of the resources in the set A," where A may be defined locally. The first step toward this goal is generation of lexically scoped names and late binding of names with program values. Lexically scoped names allow the creation of new names distinguishable from other names, and late binding allows existing program values to be must-aliased with these names. Together, they support creation of fine must-aliasing relations. Note that relations are not lexically scoped; they may be transported outside of a lexical scope by existential or universal name quantification.

However, there is a problem when these names are left disjoint. The problem occurs when the names collide (with other names or with their lexical boundaries), for example, when the type system needs to equate two types differing in names. The problem becomes more severe in a setting like stack-based access control because names propagate and collide not only through types of expressions but also through their effects. This is the problem subnaming solves. With subnaming, names are no longer disjoint. So the type system is able to make compromises by using supernames when possible. This has the effect of "downgrading" a fine must-aliasing relation to a coarser one.

6 Related Work

Stack-based access control is used in Java [8] and the CLR [4]. Pottier et al [12] present a type system that can statically check a subset of Java's access control policies. Their system is faithful to the Java security model and hence conservatively approximates Java's dynamic access control. It is not our goal to match Java's access control system, but we believe it is possible to encode Java's mechanism in a way so that access policies enforceable by Pottier et al's system can also be enforced by ours. We also believe that our system is more expressive, because of its ability to locally refine access control policies.

The implementation of run-time systems for stack-based dynamic access control is a research subject in itself. In addition to direct inspection of the execution stack at run-time, which is done by Java and the CLR, several other techniques have been proposed. One approach is security-passing style where enabled access rights are explicitly passed as function arguments [15]. Also, an implementation technique based on code instrumentation has been proposed [13]. We believe that, should the need for run-time checks arise in our system (e.g., as discussed in Section 4.1), we can add such checks using any of these techniques. Besides efficient implementation, studying dynamic semantics has lead to a deeper understanding of the stack-based access control mechanism [15, 6]. It is interesting to note that Pottier et al [12] first develop their static system on the security-passing-style converted language, then translate back to obtain a corresponding system for the source language.

There are a variety of ways to perform access control besides the stack-based approach. Abadi et al [1] present a system based on examining execution history. Their system is a generalization of the stack-based mechanism where the access rights at a point in time are determined not only by the access rights encoded in the stack but also by those of functions that have already returned. Using our syntax, their system roughly can be explained as splitting `limit` into `start_limit` and `end_limit` and analogously for `grant`. Statically checking history-based policies in a convincing manner would require reasoning about the sequential behavior of programs, and it is unclear whether we would be able to adapt our system to statically checking history-based access control policies.

Researchers have also proposed systems for statically enforcing generic security policies. These systems include Hoare style proof-systems [11, 3] and a higher-order type system [14]. One of their goals is a portable framework that is less tied to the source language. It is difficult to compare our system with these as the problem domains are so different.

Our system is an application of type and effect systems [7, 9]. Type and effect systems are widely used for checking non-standard program properties; the work of Pottier et al also uses a type and effect system [12].

Finally, there is a connection to work on aliasing control, particularly previous work on the programming construct `restrict`, originally studied in [5] and extended in [2]. Briefly, `restrict` is a mechanism for specifying and checking local non-aliasing of store locations. To this end, abstract location names are used to approximate store locations so that location names distinguish local aliases of store locations from other aliases. Beyond what has been previously published on this subject, we noticed that, instead of using incomparable location names, we could use subnaming so that each local alias of a set of store locations S is a subtype of the type of all aliases of S. This captures the notion that each local alias is a refinement of the set of all aliases. Our use of subnaming for refinement of access control policies exploits the same idea.

7 Conclusions

We have presented a new system for defining and enforcing fine-grain access control. The system is simple, relying only on standard techniques from the types literature, yet powerful enough to express many non-trivial access control policies statically. The system draws its power from two key features: late and locally defined access control policies and non-local enforcement of the policies. We make the former possible through late binding between resources and key-pairs identified by lexically scoped names. To make the latter possible, we use subnaming.

At the core, our use of subnaming is a kind of local must alias analysis. This technique itself appears to be applicable to a variety of problems in addition to access control.

References

[1] M. Abadi and C. Fournet. Access Control based on Exection History. In *Proceedings of the tenth Annual Network and Distributed System Security Symposium*, pages 107–121, Feb. 2003.

[2] A. Aiken, J. S. Foster, J. Kodumal, and T. Terauchi. Checking and Inferring Local Non-Aliasing. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, California, June 2003.

[3] C. Colby, K. Crary, R. Harper, P. Lee, and F. Pfenning. Automated techniques for provably safe mobile code. *Theoretical Computer Science*, 290:1175–1199, 2003. Special issue on *Dependable Computing*.

[4] ECMA. *Standard ECMA-335: Common Language Infrastructure*, 2002. http://www.ecma-international.org/publications/ standards/ecma-335.htm.

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

[6] C. Fournet and A. G. Gordon. Stack Inspection: Theory and Variants. In *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 307–318, Portland, Oregon, Jan. 2002.

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

[8] L. Gong. *Java Security Architecture (JDK 1.2)*, 1998. http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-spec.doc.html.

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

[10] J. C. Mitchell. Abstract Types Have Existential Type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

[11] G. Necula and P. Lee. Safe, Untrusted Agents using Proof-Carrying Code. In I. G. Vigna, editor, *Special Issue on Mobile Agent Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 61–91. Springer-Verlag, 1998.

$$x \in \text{Vars} \quad \ell \in \text{Locations} \quad k \in \text{KeyPairs} \cup \{\texttt{bot}, \texttt{top}\} \quad A \subseteq_{\text{finite}} \text{KeyPairs} \cup \{\texttt{bot}, \texttt{top}\}$$

$$\begin{array}{llll}
\text{raw values} & r & ::= & m \mid \texttt{loc}\,(\ell) \mid \texttt{lkey}\,(k) \mid \texttt{gkey}\,(k) \mid \langle v_1, v_2, \ldots, v_n\rangle \mid \lambda x{:}\bullet.e \mid \Lambda \bullet .v \mid \texttt{pack}\; v \;\texttt{as}\; \bullet \\
\text{values} & v & ::= & x \mid k\; r \mid \texttt{err} \\
\text{expressions} & e & ::= & v \mid \langle e_1, e_2, \ldots, e_n\rangle \mid e.i \mid e_1\; e_2 \mid \texttt{ref}\; e \mid e_1 \;\texttt{:=}\; e_2 \mid \texttt{!}\; e \mid \texttt{spawn}\; e \mid \\
 & & & e\,[\bullet] \mid \texttt{pack}\; e \;\texttt{as}\; \bullet \mid \texttt{open}\; x = e_1 \;\texttt{as}\; \bullet \;\texttt{in}\; e_2 \mid \\
 & & & \texttt{limit}\; e_1, e_2, \ldots, e_{n-1} \;\texttt{in}\; e_n \mid \texttt{grant}\; e_1 \;\texttt{in}\; e_2 \mid \texttt{newkey<}e \mid \texttt{associate}\; e_1 \;\texttt{with}\; e_2 \mid \texttt{access}\; A \;\texttt{in}\; e
\end{array}$$

Figure 5: Target language

$$\begin{array}{llll}
R\text{-contexts} & R & ::= & [\,] \mid \langle \ldots, v_i, R, \ldots\rangle \mid R.i \mid R\; e \mid v\; R \mid \texttt{ref}\; R \mid R \;\texttt{:=}\; e \mid v \;\texttt{:=}\; R \mid R\,[\bullet] \mid \texttt{pack}\; R \;\texttt{as}\; \bullet \mid \texttt{open}\; x = R \;\texttt{as}\; \bullet \;\texttt{in}\; e \mid \\
 & & & \texttt{limit}\; \ldots, v_i, R, \ldots \;\texttt{in}\; e \mid \texttt{grant}\; R \;\texttt{in}\; e \mid \texttt{newkey<}R \mid \texttt{associate}\; R \;\texttt{with}\; e \mid \texttt{associate}\; v \;\texttt{with}\; R \\
E\text{-contexts} & E & ::= & [\,] \mid \langle \ldots, v_i, E, \ldots\rangle \mid E.i \mid E\; e \mid v\; E \mid \texttt{ref}\; E \mid E \;\texttt{:=}\; e \mid v \;\texttt{:=}\; E \mid E\,[\bullet] \mid \texttt{pack}\; E \;\texttt{as}\; \bullet \mid \texttt{open}\; x = E \;\texttt{as}\; \bullet \;\texttt{in}\; e \mid \\
 & & & \texttt{limit}\; \ldots, v_i, E, \ldots \;\texttt{in}\; e \mid \texttt{grant}\; E \;\texttt{in}\; e \mid \texttt{newkey<}E \mid \texttt{associate}\; E \;\texttt{with}\; e \mid \texttt{associate}\; v \;\texttt{with}\; E \mid E \mid \\
 & & & \texttt{access}\; A \;\texttt{in}\; E
\end{array}$$

Figure 6: Evaluation contexts

[12] F. Pottier, C. Skalka, and S. Smith. A Systematic Approach to Static Access Control. In D. Sands, editor, *10th European Symposium on Programming*, volume 2028 of *Lecture Notes in Computer Science*, pages 30–45, Genova, Italy, 2001. Springer-Verlag.

[13] Úlfar Erlingsson and F. B. Schneider. IRM Enforcement of Java Stack Inspection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 246–255, Oakland, California, May 2000.

[14] D. Walker and K. Watkins. A Type System for Expressive Security Policies. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 254–267, Boston, Massachusetts, Jan. 2000.

[15] D. S. Wallach, A. W. Appel, and E. W. Felten. SAFKASI: A Security Mechnism for Language-based Systems. *ACM Transactions on Software Engineering and Methodology*, 9(4):341–378, Oct. 2000.

A Type Soundness

We sometimes use the symbol $-$ to indicate "don't care."

A.1 Dynamic Semantics

The first step in the dynamic semantics is to translate the source program into the language shown in Figure 5. The translation involves type-erasure, replacing each name or type annotation with a $\bullet$. We erase names and types so that we do not need to worry about proper alpha-renaming of names during reductions. Also each program value in the source program is translated into a program value in of the target program by annotating them with `bot`, i.e., for each v in the source, `bot` v. Finally, the translation replaces each `Top` with `bot lkey (top)`.

Let e be the translated target program. Then the evaluation proceeds from the initial state $(\emptyset, \emptyset, \langle e\rangle)$ followed by a sequence of small-step reductions shown in Figure 7. Each state is of the form $(S, K, \vec{e})$ where S is the store mapping store locations to values, K is the key-pair manager representing immediate sub-key-pair relation between key-pairs, and $\vec{e}$ are the execution threads.

The evaluation contexts E and R are shown in Figure 6. The only difference between E and R is that R lacks `access` A `in` R.

$$\frac{}{(S, K, \langle \ldots, E[\langle v_1, v_2, \ldots, v_n\rangle], \ldots\rangle) \longrightarrow (S, K, \langle \ldots, E[\texttt{bot}\ \langle v_1, v_2, \ldots, v_n\rangle], \ldots\rangle)}\ \text{[Tuple]}$$

$$\frac{k \in \mathit{Close}_K(A)}{(S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[(k\ \langle \ldots, v_i, \ldots\rangle).i]], \ldots\rangle) \longrightarrow (S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[v_i]], \ldots\rangle)}\ \text{[Proj]}$$

$$\frac{k \in \mathit{Close}_K(A)}{(S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[(k\ \lambda x{:}\bullet.e)\ v]], \ldots\rangle) \longrightarrow (S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[e[v/x]]], \ldots\rangle)}\ \text{[App]}$$

$$\frac{\ell \notin \mathit{dom}(S)}{(S, K, \langle \ldots, E[\texttt{ref}\ v], \ldots\rangle) \longrightarrow (S \cup \{\ell \mapsto v\}, K, \langle \ldots, E[\texttt{bot}\ \texttt{loc}\ (\ell)], \ldots\rangle)}\ \text{[Ref]}$$

$$\frac{k \in \mathit{Close}_K(A)}{(S \cup \{\ell \mapsto -\}, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[(k\ \texttt{loc}\ (\ell))\ \texttt{:=}\ v]], \ldots\rangle) \longrightarrow (S \cup \{\ell \mapsto v\}, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[v]], \ldots\rangle)}\ \text{[Assign]}$$

$$\frac{k \in \mathit{Close}_K(A)}{(S \cup \{\ell \mapsto v\}, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[\texttt{!}\ k\ \texttt{loc}\ (\ell)]], \ldots\rangle) \longrightarrow (S \cup \{\ell \mapsto v\}, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[v]], \ldots\rangle)}\ \text{[Deref]}$$

$$\frac{}{(S, K, \langle \ldots, E[\texttt{spawn}\ e], \ldots\rangle) \longrightarrow (S, K, \langle \ldots, E[\texttt{bot}\ 0], \ldots, \texttt{access}\ \emptyset\ \texttt{in}\ e\rangle)}\ \text{[Spawn]}$$

$$\frac{k \in \mathit{Close}_K(A)}{(S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[(k\ \Lambda \bullet .v)\ [\bullet]]], \ldots\rangle) \longrightarrow (S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[v]], \ldots\rangle)}\ \text{[Inst]}$$

$$\frac{}{(S, K, \langle \ldots, E[\texttt{pack}\ v\ \texttt{as}\ \bullet], \ldots\rangle) \longrightarrow (S, K, \langle \ldots, E[\texttt{bot}\ \texttt{pack}\ v\ \texttt{as}\ \bullet], \ldots\rangle)}\ \text{[Pack]}$$

$$\frac{k \in \mathit{Close}_K(A)}{(S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[\texttt{open}\ x\ \texttt{=}\ (k\ \texttt{pack}\ v\ \texttt{as}\ \bullet)\ \texttt{as}\ \bullet\ \texttt{in}\ e]], \ldots\rangle) \longrightarrow (S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[e[v/x]]], \ldots\rangle)}\ \text{[Open]}$$

$$\frac{\{k'_1, k'_2, \ldots, k'_n\} \subseteq \mathit{Close}_K(A)}{\begin{array}{l}(S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[\texttt{limit}\ k'_1\ \texttt{lkey}\ (k_1), k'_2\ \texttt{lkey}\ (k_2), \ldots, k'_n\ \texttt{lkey}\ (k_n)\ \texttt{in}\ e]], \ldots\rangle) \longrightarrow \\ \qquad (S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[\texttt{access}\ \mathit{Close}_K(A) \cap \mathit{Close}_K(\{k_1, k_2, \ldots, k_n\})\ \texttt{in}\ e]], \ldots\rangle)\end{array}}\ \text{[Limit]}$$

$$\frac{k_1 \in \mathit{Close}_K(A)}{(S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[\texttt{grant}\ k_1\ \texttt{gkey}\ (k_2)\ \texttt{in}\ e]], \ldots\rangle) \longrightarrow (S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[\texttt{access}\ A \cup \{k_2\}\ \texttt{in}\ e]], \ldots\rangle)}\ \text{[Grant]}$$

$$\frac{k_1 \in \mathit{Close}_K(A) \qquad\qquad k_3 \notin \mathit{dom}(K) \cup \{\texttt{bot}, \texttt{top}\}}{\begin{array}{l}(S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[\texttt{newkey<}k_1\ \texttt{lkey}\ (k_2)]], \ldots\rangle) \longrightarrow \\ \qquad (S, K \cup \{k_3 \mapsto k_2\}, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[\texttt{bot}\ \texttt{pack}\ \texttt{bot}\ \langle \texttt{bot}\ \texttt{lkey}\ (k_3), \texttt{bot}\ \texttt{gkey}\ (k_3)\rangle\ \texttt{as}\ \bullet]], \ldots\rangle)\end{array}}\ \text{[NewKey]}$$

$$\frac{\{k_1, k_2\} \subseteq \mathit{Close}_K(A)}{(S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[\texttt{associate}\ k_1\ r\ \texttt{with}\ k_2\ \texttt{lkey}\ (k_3)]], \ldots\rangle) \longrightarrow (S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[k_3\ r]], \ldots\rangle)}\ \text{[Associate]}$$

$$\frac{}{(S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ v], \ldots\rangle) \longrightarrow (S, K, \langle \ldots, E[v], \ldots\rangle)}\ \text{[InAccess]}$$

Figure 7: Small-step reduction rules

$$\frac{\Gamma \vdash k : \rho}{\Gamma \vdash k\ m : \rho\ \mathit{int}; \emptyset}\ \text{(Int'ed)} \qquad \frac{\Gamma \vdash k : \rho \qquad \Gamma \vdash \ell : \tau}{\Gamma \vdash k\ \mathtt{loc}\,(\ell) : \rho\ \mathit{ref}\,(\tau); \emptyset}\ \text{(Loc)}$$

$$\frac{\Gamma \vdash k_1 : \rho_1 \qquad \Gamma \vdash k_2 : \rho_2}{\Gamma \vdash k_1\ \mathtt{lkey}\,(k_2) : \rho_1\ \mathit{lkey}\,(\rho_2); \emptyset}\ \text{(LKey)} \qquad \frac{\Gamma \vdash k_1 : \rho_1 \qquad \Gamma \vdash k_2 : \rho_2}{\Gamma \vdash k_1\ \mathtt{gkey}\,(k_2) : \rho_1\ \mathit{gkey}\,(\rho_2); \emptyset}\ \text{(GKey)}$$

$$\frac{\begin{array}{c}\Gamma \vdash k : \rho \\ \text{For } 1 \leq i \leq n,\ \Gamma \vdash v_i : \tau_i; \emptyset\end{array}}{\Gamma \vdash k\ \langle v_1, v_2, \ldots, v_n\rangle : \rho\ \langle \tau_1, \tau_2, \ldots, \tau_n\rangle; \emptyset}\ \text{(Tuple'ed)} \qquad \frac{\Gamma \vdash k : \rho \qquad \Gamma, x{:}\tau_1 \vdash e : \tau_2; L \qquad \Gamma, x{:}\tau_1 \vdash \diamond}{\Gamma \vdash k\ \lambda x{:}\bullet.e : \rho\ (\tau_1 \xrightarrow{L} \tau_2); \emptyset}\ \text{(Fun'ed)}$$

$$\frac{\Gamma \vdash k : \rho_3 \qquad \Gamma, \rho_1 {<} \rho_2 \vdash v : \tau; \emptyset \qquad \Gamma, \rho_1 {<} \rho_2 \vdash \diamond}{\Gamma \vdash k\ \Lambda \bullet .v : \rho_3\ (\forall \rho_1 {<} \rho_2.\tau); \emptyset}\ \text{(Gen'ed)} \qquad \frac{\Gamma \vdash e : \rho_1\ (\forall \rho_2 {<} \rho_3.\tau); L \qquad \Gamma \vdash \rho_4 < \rho_3}{\Gamma \vdash e\,[\bullet] : \tau[\rho_4/\rho_2]; L \cup \{\rho_1\}}\ \text{(Inst }\bullet\text{)}$$

$$\frac{\Gamma \vdash k : \rho_4 \qquad \Gamma \vdash v : \tau[\rho_3/\rho_1]; \emptyset \qquad \Gamma \vdash \rho_3 < \rho_2}{\Gamma \vdash k\ \mathtt{pack}\ v\ \mathtt{as}\ \bullet : \rho_4\ (\exists \rho_1 {<} \rho_2.\tau); \emptyset}\ \text{(Packed)}$$

$$\frac{\Gamma \vdash e : \tau[\rho_3/\rho_1]; L \qquad \Gamma \vdash \rho_3 < \rho_2}{\Gamma \vdash \mathtt{pack}\ e\ \mathtt{as}\ \bullet : \bot\ (\exists \rho_1 {<} \rho_2.\tau); L}\ \text{(Pack }\bullet\text{)} \qquad \frac{\begin{array}{c}\Gamma \vdash e_1 : \rho_1\ (\exists \rho_2 {<} \rho_3.\tau_1); L_1 \qquad \Gamma, \rho_2 {<} \rho_3, x{:}\tau_1 \vdash e_2 : \tau_2; L_2 \\ \Gamma \vdash \tau_2 \qquad \Gamma \vdash L_2 \qquad \Gamma, \rho_2 {<} \rho_3, x{:}\tau_1 \vdash \diamond\end{array}}{\Gamma \vdash \mathtt{open}\ x\ \mathtt{=}\ e_1\ \mathtt{as}\ \bullet\ \mathtt{in}\ e_2 : \tau_2; L_1 \cup L_2 \cup \{\rho_1\}}\ \text{(Open }\bullet\text{)}$$

$$\frac{\Gamma \vdash e : \tau; L \qquad \Gamma \vdash L < \{\rho \mid \Gamma \vdash k : \rho \text{ for some } k \in A\}}{\Gamma \vdash \mathtt{access}\ A\ \mathtt{in}\ e : \tau; \emptyset}\ \text{(InAccess)}$$

$$\frac{k{:}\rho \in \Gamma}{\Gamma \vdash k : \rho}\ \text{(KeyPair } k\text{)} \qquad \frac{}{\Gamma \vdash \mathtt{top} : \top}\ \text{(KeyPair } \mathtt{top}\text{)} \qquad \frac{}{\Gamma \vdash \mathtt{bot} : \bot}\ \text{(KeyPair } \mathtt{bot}\text{)} \qquad \frac{\ell{:}\tau \in \Gamma}{\Gamma \vdash \ell : \tau}\ \text{(Location } \ell\text{)}$$

$$\frac{\Gamma \vdash \rho \qquad \Gamma \vdash \diamond}{\Gamma, k{:}\rho \vdash \diamond}\ \text{(Env } k\text{)} \qquad \frac{\Gamma \vdash \tau \qquad \Gamma \vdash \diamond}{\Gamma, \ell{:}\tau \vdash \diamond}\ \text{(Env } \ell\text{)}$$

Figure 8: Type checking rules: additional rules for the target language

Close_K is a function such that $k \in \mathit{Close}_K(A)$ iff k is a sub-key-pair of some key-pair in A in the key-pair relation K. More concretely, we define the reflexive, transitive closure of K lifted with `bot` as

$$\begin{array}{rcl} K^*_\bot & = & \bigcup_{n \in \omega} R^n(K \cup \{\mathtt{bot} \mapsto k \mid k \in \mathit{dom}(K) \cup \{\mathtt{top}\}\}) \\ & & \cup \{k \mapsto k \mid k \in \mathit{dom}(K) \cup \{\mathtt{bot}, \mathtt{top}\}\} \end{array}$$

where

$$\begin{array}{l} R(K) = K \cup \{k_1 \mapsto k_2 \mid k_1 \mapsto k_3 \in K \text{ and} \\ \qquad\qquad\qquad\qquad\quad k_3 \mapsto k_2 \in K \text{ for some } k_3\} \end{array}$$

Then Close_K can be defined as

$$\mathit{Close}_K(A) = \{k_1 \mid k_1 \mapsto k_2 \in K^*_\bot \text{ for some } k_2 \in A\}$$

If the state $(S, K, \langle \ldots, e, \ldots\rangle)$ cannot be reduced by the rules in Figure 7 but e is not a value, then e immediately reduces to a special constant `err`, i.e.

$$(S, K, \langle \ldots, e, \ldots\rangle) \longrightarrow (S, K, \langle \ldots, \mathtt{err}, \ldots\rangle)$$

A.2 Type Soundness

The static semantics for the target language is the type checking rules from Section 3.1 plus the additional rules in Figure 8. Each type environment is now a sequence of variable-to-type bindings, names annotated with a supername, location-to-type bindings, and key-pair-to-name bindings.

$$\Gamma ::= \emptyset \mid \Gamma, x{:}\tau \mid \Gamma, \rho_1{<}\rho_2 \mid \ell{:}\tau \mid k{:}\rho$$

Note that `err` has no type. The following theorem connects source programs to target programs.

Theorem 2 *Let e_1 be an expression in the source language and e_2 be the corresponding translated expression. Then if $\Gamma \vdash e_1 : \tau; L$ then $\Gamma \vdash e_2 : \tau; L$.*

Proof: By induction on the type checking derivation. □

In the rest of the paper, we always refer to expressions in the target language unless specified otherwise.

We re-state the definition of well-typed program state.

Definition 2 $\Gamma \vdash (S, K, \langle e_1, e_2, \ldots, e_n\rangle)$ *(read "the program state $(S, K, \langle e_1, e_2, \ldots, e_n\rangle)$ is well-typed under Γ") iff*

(1) $\Gamma \vdash \diamond$.

(2) For each $1 \leq i \leq n$, $\Gamma \vdash e_i : -; \emptyset$.

(3) $\ell \in dom(S)$ *iff* $\ell \in dom(\Gamma)$.

(4) $k \in dom(K)$ *iff* $k \in dom(\Gamma)$.

(5) $x \notin dom(\Gamma)$.

(6) If $\Gamma \vdash \ell : \tau$ *and* $S(\ell) = v$ *then* $\Gamma \vdash v : \tau; -$.

(7) Suppose $\Gamma \vdash k_1 : \rho_1$ *and* $\Gamma \vdash k_2 : \rho_2$. *Then* $\Gamma \vdash \rho_1 < \rho_2$ *iff* $k_1 \mapsto k_2 \in K_\perp^*$.

Lemma 1 *If $\Gamma \vdash \diamond$ and $\Gamma \vdash e : \tau; L$ then $\Gamma \vdash \tau$ and $\Gamma \vdash L$.*

Proof: By induction on the type checking derivation. □

Lemma 2 (Substitution) *(1) If $\Gamma, x{:}\tau_1 \vdash \diamond$, $\Gamma, x{:}\tau_1 \vdash e : \tau_2; L$, and $\Gamma \vdash v : \tau_1; \emptyset$, then $\Gamma \vdash e[v/x] : \tau_2; L$.*

(2) If $\Gamma, \rho_1{<}\rho_2 \vdash \diamond$, $\Gamma, \rho_1{<}\rho_2 \vdash e : \tau; L$, and $\Gamma \vdash \rho_3 < \rho_2$, then $\Gamma \vdash e : \tau[\rho_3/\rho_1]; L[\rho_3/\rho_1]$.

(3) If $\Gamma, \rho_1{<}\rho_2 \vdash \diamond$, $\Gamma, \rho_1{<}\rho_2 \vdash \tau_1 < \tau_2$, and $\Gamma \vdash \rho_3 < \rho_2$, then $\Gamma \vdash \tau_1[\rho_3/\rho_1] < \tau_2[\rho_3/\rho_1]$.

(4) If $\Gamma, \rho_1 < \rho_2, x : \tau_1 \vdash \diamond$, $\Gamma, \rho_1 < \rho_2, x : \tau_1 \vdash e : \tau_2; L$, and $\Gamma \vdash \rho_3 < \rho_2$, then $\Gamma, x : \tau_1[\rho_3/\rho_1] \vdash e : \tau_2[\rho_3/\rho_1]; L[\rho_3/\rho_1]$.

Proof: By induction on the type checking derivation. □

Lemma 3 *Suppose $\Gamma_1 \vdash \diamond$ and $\Gamma_1 \vdash e : \tau; L$. Then for any $\Gamma_2 \supseteq \Gamma_1$ such that $\Gamma_2 \vdash \diamond$, $\Gamma_2 \vdash e : \tau; L$.*

Proof: The only non-trivial case is when Γ_2 contains more instances of name bindings, i.e., $\rho{<}-$'s, than Γ_1, since type environment well-formedness conditions may fail with more names. But it is easy to see that whenever a new name is introduced in the environment, we may choose a name that has not appeared in the environment (see (Gen'ed) and (Open •)). By Lemma 1, the choices do not affect the conclusions. Hence the lemma follows. □

Lemma 4 (Replacement) *Suppose $\Gamma_1 \vdash \diamond$ and $\Gamma_1 \vdash E[e_1] : \tau; L$. Then there is a sub-derivation $\Gamma_1 \vdash e_1 : \tau_1; L_1$ for some τ_1, L_1. Furthermore, for any e_2 and $\Gamma_2 \supseteq \Gamma_1$ such that $\Gamma_2 \vdash \diamond$ and $\Gamma_2 \vdash e_2 : \tau_1; L_1$, we have $\Gamma_2 \vdash E[e_2] : \tau; L$.*

Proof: This follows from the usual replacement argument and Lemma 3. □

Lemma 5 *Suppose $\Gamma \vdash R[e] : \tau; L$. Let $\Gamma \vdash e : \tau_1; L_1$ be a sub-derivation. Then $\Gamma \vdash L_1 < L$.*

Proof: By induction on the type checking derivation. □

Lemma 6 *If* $\Gamma \vdash v : \tau; L$ *then* $\Gamma \vdash v : \tau; \emptyset$.

Proof: By induction on the type checking derivation. □

We are now prepared prove our main theorem, re-stated here.

Theorem 3 (Subject Reduction) *If* $\Gamma \vdash (S, K, \vec{e})$ *and* $(S, K, \vec{e}) \longrightarrow (S', K', \vec{e}')$*, then there is* Γ' *such that* $\Gamma' \vdash (S', K', \vec{e})$.

Proof: Firstly, we show that for each $e_i \in \vec{e}$, if e_i is not a value then $(S, K, \vec{e})$ is of the form matching the left hand side of $\longrightarrow$ in one of the reduction rules from Figure 7 with e_i as the evaluating thread. To see this, first note that for [Proj], [App], [Ref], [Assign], [Deref], [Open], [Limit], [Grant], [NewKey], and [Associate], the redex indeed does appear in some context of the form $E[\texttt{access}\ A\ \texttt{in}\ R[\ ]]$ since otherwise it must appear in some context of the form $R[\ ]$ but then by Lemma 5 $\Gamma \not\vdash e_i : -; \emptyset$. Secondly, for [Assign] (ref. [Deref]), the location assigned (ref. dereferenced) must be in the store because otherwise it violates Definition 2 (3).

Therefore it suffices to show the statement in the theorem holds for each forms matching $(S, K, \vec{e})$.

[Tuple] Suppose

$$\Gamma \vdash (S, K, \langle \ldots, E[\langle v_1, v_2, \ldots, v_n \rangle], \ldots \rangle)$$

So we have $\Gamma \vdash E[\langle v_1, v_2, \ldots, v_n \rangle] : -; \emptyset$. By inspection of the type checking rules, there must be a sub-derivation with (Tuple) as the last rule.

$$\frac{\text{For } 1 \le i \le n,\ \Gamma \vdash v_i : \tau_i; L_i}{\Gamma \vdash \langle v_1, v_2, \ldots, v_n \rangle : \bot\ \langle \tau_1, \tau_2, \ldots, \tau_n \rangle; \bigcup_{1 \le i \le n} L_i}$$

We have

$$\begin{array}{l}(S, K, \langle \ldots, E[\langle v_1, v_2, \ldots, v_n \rangle], \ldots \rangle) \longrightarrow \\ \qquad (S, K, \langle \ldots, E[\texttt{bot}\ \langle v_1, v_2, \ldots, v_n \rangle], \ldots \rangle)\end{array}$$

Let $\Gamma' = \Gamma$. We have by (Tuple'ed)

$$\frac{\begin{array}{c}\Gamma' \vdash \texttt{bot} : \bot \\ \text{For } 1 \le i \le n,\ \Gamma' \vdash v_i : \tau_i; \emptyset\end{array}}{\Gamma' \vdash \texttt{bot}\ \langle v_1, v_2, \ldots, v_n \rangle : \bot\ \langle \tau_1, \tau_2, \ldots, \tau_n \rangle; \emptyset}$$

Then by Lemma 1 and (Subsumption),

$$\Gamma' \vdash \texttt{bot}\ \langle v_1, v_2, \ldots, v_n \rangle : \bot\ \langle \tau_1, \tau_2, \ldots, \tau_n \rangle; \bigcup_{1 \le i \le n} L_i$$

Hence by Lemma 4, it follows that

$$\Gamma' \vdash (S, K, \langle \ldots, E[\texttt{bot}\ \langle v_1, v_2, \ldots, v_n \rangle], \ldots \rangle)$$

[Proj] Suppose

$$\Gamma \vdash (S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[(k\ \langle \ldots, v_i, \ldots \rangle).i]], \ldots \rangle)$$

So we have $\Gamma \vdash E[\texttt{access}\ A\ \texttt{in}\ R[(k\ \langle \ldots, v_i, \ldots \rangle).i]] : -; \emptyset$. By inspection of the type checking rules, there must be a sub-derivation with (Proj) as the last rule.

$$\frac{\Gamma \vdash k\ \langle \ldots, v_i, \ldots \rangle : \rho\ \langle \tau_1, \tau_2, \ldots, \tau_n \rangle; L}{\Gamma \vdash (k\ \langle \ldots, v_i, \ldots \rangle).i : \tau_i; L \cup \{\rho\}}$$

We first show that

$$\begin{array}{l}(S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[(k\ \langle \ldots, v_i, \ldots \rangle).i]], \ldots \rangle) \\ \qquad \longrightarrow (S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[v_i]], \ldots \rangle)\end{array}$$

It suffices to show that $k \in Close_K(A)$. By Lemma 5 and (InAccess), it must be the case that

$$\Gamma \vdash \{\rho\} < \{\rho \mid \Gamma \vdash k : \rho \text{ for some } k \in A\}$$

By inspection of the type checking rules, we see that there must be ρ' such that

$$\Gamma \vdash k : \rho' \quad \Gamma \vdash \rho' < \rho$$

So by Definition 2 (7), $k \in Close_K(A)$.

Now let $\Gamma' = \Gamma$. We next show that

$$\Gamma' \vdash (S, K, \langle \ldots, E[\mathtt{access}\ A\ \mathtt{in}\ R[v_i]], \ldots\rangle)$$

By inspection of the type checking rules, we see that the sub-derivation of the premise must end with (Tuple'ed) followed by zero or more (Subsumption)'s. So

$$\Gamma' \vdash v_i : \tau_i'; \emptyset \quad \Gamma' \vdash \tau_i' < \tau_i \quad \Gamma' \vdash \emptyset < L \cup \{\rho\}$$

So by Lemma 1 and (Subsumption)

$$\Gamma' \vdash v_i : \tau_i; L \cup \{\rho\}$$

So by Lemma 4 it follows that,

$$\Gamma' \vdash (S, K, \langle \ldots, E[\mathtt{access}\ A\ \mathtt{in}\ R[v_i]], \ldots\rangle)$$

[App] Suppose

$$\Gamma \vdash (S, K, \langle \ldots, E[\mathtt{access}\ A\ \mathtt{in}\ R[(k\ \lambda x{:}\bullet.e)\ v]], \ldots\rangle)$$

So we have $\Gamma \vdash E[\mathtt{access}\ A\ \mathtt{in}\ R[(k\ \lambda x{:}\bullet.e)\ v]] : -; \emptyset$. By inspection of the type checking rules, there must be a sub-derivation with (App) as the last rule.

$$\frac{\Gamma \vdash k\ \lambda x{:}\bullet.e : \rho\ (\tau_1 \xrightarrow{L_1} \tau_2); L_2 \qquad \Gamma \vdash v : \tau_1; L_3}{\Gamma \vdash k\ \lambda x{:}\bullet.e\ v : \tau_2; L_1 \cup L_2 \cup L_3 \cup \{\rho\}}$$

We first show that

$$\begin{array}{l}(S, K, \langle \ldots, E[\mathtt{access}\ A\ \mathtt{in}\ R[(k\ \lambda x{:}\bullet.e)\ v]], \ldots\rangle) \longrightarrow \\ \qquad (S, K, \langle \ldots, E[\mathtt{access}\ A\ \mathtt{in}\ R[e[v/x]]], \ldots\rangle)\end{array}$$

It suffices to show that $k \in Close_K(A)$. By Lemma 5 and (InAccess), it must be the case that

$$\Gamma \vdash \{\rho\} < \{\rho \mid \Gamma \vdash k : \rho \text{ for some } k \in A\}$$

By inspection of the type checking rules, we see that there must be ρ' such that

$$\Gamma \vdash k : \rho' \quad \Gamma \vdash \rho' < \rho$$

So by Definition 2 (7), $k \in Close_K(A)$.

Now let $\Gamma' = \Gamma$. We next show that

$$\Gamma' \vdash (S, K, \langle \ldots, E[\mathtt{access}\ A\ \mathtt{in}\ R[e[v/x]]], \ldots\rangle)$$

By inspection of the type checking rules, we see that the sub-derivation of the first premise must end with (Fun'ed) followed by zero or more (Subsumption)'s. So

$$\begin{array}{c}\Gamma', x{:}\tau_1' \vdash e : \tau_2'; L_1' \quad \Gamma', x{:}\tau_1' \vdash \diamond \\ \Gamma' \vdash \tau_1 < \tau_1' \quad \Gamma' \vdash \tau_2' < \tau_2 \\ \Gamma' \vdash L_1' < L_1 \quad \Gamma' \vdash \emptyset < L_2\end{array}$$

So by (Subsumption),

$$\Gamma' \vdash v : \tau_1'; L_3$$

and by Lemma 2 (1), Lemma 1, and (Subsumption),

$$\Gamma' \vdash e[v/x] : \tau_2; L_1 \cup L_2 \cup L_3$$

So by Lemma 4 it follows that

$$\Gamma' \vdash (S, K, \langle \ldots, E[\mathtt{access}\ A\ \mathtt{in}\ R[e[v/x]]], \ldots\rangle)$$

[Ref] Suppose

$$\Gamma \vdash (S, K, \langle \ldots, E[\texttt{ref}\ v], \ldots \rangle)$$

So we have $\Gamma \vdash E[\texttt{ref}\ v] : -; \emptyset$. By inspection of the type checking rules, there must be a sub-derivation with (Ref) as the last rule.

$$\frac{\Gamma \vdash v : \tau; L}{\Gamma \vdash \texttt{ref}\ v : \bot\ \mathit{ref}(\tau); L}$$

We have

$$\begin{array}{l}(S, K, \langle \ldots, E[\texttt{ref}\ v], \ldots \rangle) \longrightarrow \\ \qquad (S \cup \{\ell \mapsto v\}, K, \langle \ldots, E[\texttt{bot}\ \texttt{loc}\ (\ell)], \ldots \rangle)\end{array}$$

for $\ell \notin \mathit{dom}(S)$. Let $\Gamma' = \Gamma, \ell{:}\tau$. We have by (Loc)

$$\frac{\Gamma' \vdash \texttt{bot} : \bot \qquad \Gamma' \vdash \ell : \tau}{\Gamma' \vdash \texttt{bot}\ \texttt{loc}\ (\ell) : \bot\ \mathit{ref}(\tau); \emptyset}$$

Then by Lemma 1 and (Subsumption),

$$\Gamma' \vdash \texttt{bot}\ \texttt{loc}\ (\ell) : \bot\ \mathit{ref}(\tau); L$$

Now, we have $\Gamma' \vdash \ell : \tau$, $(S \cup \{\ell \mapsto v\})(\ell) = v$, and $\Gamma \vdash v : \tau; -$. Hence by Lemma 3 and Lemma 4, it follows that

$$\Gamma' \vdash (S \cup \{\ell \mapsto v\}, K, \langle \ldots, E[\texttt{bot}\ \texttt{loc}\ (\ell)], \ldots \rangle)$$

[Assign] Suppose

$$\Gamma \vdash (S \cup \{\ell \mapsto -\}, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[(k\ \texttt{loc}\ (\ell))\ \texttt{:=}\ v]], \ldots \rangle)$$

So we have $\Gamma \vdash E[\texttt{access}\ A\ \texttt{in}\ R[(k\ \texttt{loc}\ (\ell))\ \texttt{:=}\ v]] : -; \emptyset$. By inspection of the type checking rules, there must be a sub-derivation with (Assign) as the last rule.

$$\frac{\Gamma \vdash k\ \texttt{loc}\ (\ell) : \rho\ \mathit{ref}(\tau); L_1 \qquad \Gamma \vdash v : \tau; L_2}{\Gamma \vdash (k\ \texttt{loc}\ (\ell))\ \texttt{:=}\ v : \tau; L_1 \cup L_2 \cup \{\rho\}}$$

We first show that

$$\begin{array}{l}(S \cup \{\ell \mapsto -\}, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[(k\ \texttt{loc}\ (\ell))\ \texttt{:=}\ v]], \ldots \rangle) \\ \qquad \longrightarrow (S \cup \{\ell \mapsto v\}, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[v]], \ldots \rangle)\end{array}$$

It suffices to show that $k \in \mathit{Close}_K(A)$. By Lemma 5 and (InAccess), it must be the case that

$$\Gamma \vdash \{\rho\} < \{\rho \mid \Gamma \vdash k : \rho \text{ for some } k \in A\}$$

By inspection of the type checking rules, we see that there must be ρ' such that

$$\Gamma \vdash k : \rho' \quad \Gamma \vdash \rho' < \rho$$

So by Definition 2 (7), $k \in \mathit{Close}_K(A)$.

Now let $\Gamma' = \Gamma$. We next show that

$$\Gamma' \vdash (S \cup \{\ell \mapsto v\}, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[v]], \ldots \rangle)$$

By inspection of the type checking rules, we see that the sub-derivation of the first premise must end with (Loc) followed by zero or more (Subsumption)'s. So

$$\Gamma' \vdash \ell : \tau$$

(Subtyping is invariant under pointer types.) Hence Definition 2 (6) is satisfied. Also by Lemma 1 and (Subsumption)

$$\Gamma \vdash v : \tau; L_1 \cup L_2 \cup \{\rho\}$$

Hence by Lemma 4, it follows that

$$\Gamma' \vdash (S \cup \{\ell \mapsto v\}, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[v]], \ldots \rangle)$$

[Deref] Suppose

$$\Gamma \vdash (S \cup \{\ell \mapsto v\}, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[\texttt{!}\ k\ \texttt{loc}\,(\ell)]], \ldots \rangle)$$

So we have $\Gamma \vdash E[\texttt{access}\ A\ \texttt{in}\ R[\texttt{!}\ k\ \texttt{loc}\,(\ell)]] : -; \emptyset$. By inspection of the type checking rules, there must be a sub-derivation with (Deref) as the last rule.

$$\frac{\Gamma \vdash k\ \texttt{loc}\,(\ell) : \rho\ \mathit{ref}(\tau); L}{\Gamma \vdash \texttt{!}\ k\ \texttt{loc}\,(\ell) : \tau; L \cup \{\rho\}}$$

We first show that

$$\begin{array}{l}(S \cup \{\ell \mapsto v\}, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[\texttt{!}\ k\ \texttt{loc}\,(\ell)]], \ldots \rangle) \\ \quad \longrightarrow (S \cup \{\ell \mapsto v\}, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[v]], \ldots \rangle)\end{array}$$

It suffices to show that $k \in \mathit{Close}_K(A)$. By Lemma 5 and (InAccess), it must be the case that

$$\Gamma \vdash \{\rho\} < \{\rho \mid \Gamma \vdash k : \rho \text{ for some } k \in A\}$$

By inspection of the type checking rules, we see that there must be ρ' such that

$$\Gamma \vdash k : \rho' \quad \Gamma \vdash \rho' < \rho$$

So by Definition 2 (7), $k \in \mathit{Close}_K(A)$.

Now let $\Gamma' = \Gamma$. We next show that

$$\Gamma' \vdash (S \cup \{\ell \mapsto v\}, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[v]], \ldots \rangle)$$

By inspection of the type checking rules, we see that the sub-derivation of the premise must end with (Loc) followed by zero or more (Subsumption)'s. So

$$\Gamma' \vdash \ell : \tau$$

(Subtyping is invariant under pointer types.) Hence by Definition 2 (6) and Lemma 6, it must be the case that

$$\Gamma \vdash v : \tau; \emptyset$$

So by Lemma 1 and (Subsumption)

$$\Gamma \vdash v : \tau; L \cup \{\rho\}$$

Hence by Lemma 4, it follows that

$$\Gamma' \vdash (S \cup \{\ell \mapsto v\}, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[v]], \ldots \rangle)$$

[Spawn] Suppose

$$\Gamma \vdash (S, K, \langle \ldots, E[\texttt{spawn}\ e], \ldots \rangle)$$

So we have $\Gamma \vdash E[\texttt{spawn}\ e] : -; \emptyset$. By inspection of the type checking rules, there must be a sub-derivation with (Spawn) as the last rule.

$$\frac{\Gamma \vdash e : \tau; \emptyset}{\Gamma \vdash \texttt{spawn}\ e : \bot\ \mathit{int}; \emptyset}$$

We have

$$\begin{array}{l}(S, K, \langle \ldots, E[\texttt{spawn}\ e], \ldots \rangle) \longrightarrow \\ \quad (S, K, \langle \ldots, E[\texttt{bot}\ 0], \ldots, \texttt{access}\ \emptyset\ \texttt{in}\ e \rangle)\end{array}$$

Let $\Gamma' = \Gamma$. We have by (Int'ed)

$$\frac{\Gamma' \vdash \texttt{bot} : \bot}{\Gamma' \vdash \texttt{bot}\ 0 : \bot\ \mathit{int}; \emptyset}$$

Also by (InAccess),

$$\frac{\Gamma' \vdash e : \tau; \emptyset \qquad \Gamma' \vdash \emptyset < \emptyset}{\Gamma' \vdash \texttt{access}\ \emptyset\ \texttt{in}\ e : \tau; \emptyset}$$

Hence by Lemma 4, it follows that

$$\Gamma' \vdash (S, K, \langle \ldots, E[\texttt{bot}\ 0], \ldots, \texttt{access}\ \emptyset\ \texttt{in}\ e \rangle)$$

[Inst] Suppose

$$\Gamma \vdash (S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[(k\ \Lambda \bullet .v)\ [\bullet]]], \ldots \rangle)$$

So we have $\Gamma \vdash E[\texttt{access}\ A\ \texttt{in}\ R[(k\ \Lambda \bullet .v)\ [\bullet]]] : -; \emptyset$. By inspection of the type checking rules, there must be a sub-derivation with (Inst $\bullet$) as the last rule.

$$\frac{\Gamma \vdash k\ \Lambda \bullet .v : \rho_1\ (\forall \rho_2 < \rho_3 . \tau); L \qquad \Gamma \vdash \rho_4 < \rho_3}{\Gamma \vdash (k\ \Lambda \bullet .v)\ [\bullet] : \tau[\rho_4/\rho_2]; L \cup \{\rho_1\}}$$

We first show that

$$\begin{array}{l}(S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[(k\ \Lambda \bullet .v)\ [\bullet]]], \ldots \rangle) \\ \qquad \longrightarrow (S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[v]], \ldots \rangle)\end{array}$$

It suffices to show that $k \in \mathit{Close}_K(A)$. By Lemma 5 and (InAccess), it must be the case that

$$\Gamma \vdash \{\rho\} < \{\rho \mid \Gamma \vdash k : \rho \text{ for some } k \in A\}$$

By inspection of the type checking rules, we see that there must be ρ' such that

$$\Gamma \vdash k : \rho' \quad \Gamma \vdash \rho' < \rho$$

So by Definition 2 (7), $k \in \mathit{Close}_K(A)$.

Now let $\Gamma' = \Gamma$. We next show that

$$\Gamma' \vdash (S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[v]], \ldots \rangle)$$

By inspection of the type checking rules, we see that the sub-derivation of the first premise must end with (Gen'ed) followed by zero or more (Subsumption)'s. So

$$\begin{array}{c}\Gamma', \rho_2 < \rho_3' \vdash v : \tau'; \emptyset \quad \Gamma', \rho_2 < \rho_3' \vdash \diamond \\ \Gamma' \vdash \rho_3 < \rho_3' \quad \Gamma' \vdash \tau' < \tau \\ \Gamma' \vdash \emptyset < L\end{array}$$

So we have

$$\Gamma' \vdash \rho_4 < \rho_3'$$

and by Lemma 2 (2), Lemma 1, and (Subsumption),

$$\Gamma' \vdash v : \tau[\rho_4/\rho_2]; L \cup \{\rho_1\}$$

So by Lemma 4 it follows that

$$\Gamma' \vdash (S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[v]], \ldots \rangle)$$

[Pack] Suppose

$$\Gamma \vdash (S, K, \langle \ldots, E[\texttt{pack}\ v\ \texttt{as}\ \bullet], \ldots \rangle)$$

So we have $\Gamma \vdash E[\texttt{pack}\ v\ \texttt{as}\ \bullet] : -; \emptyset$. By inspection of the type checking rules, there must be a sub-derivation with (Pack $\bullet$) as the last rule.

$$\frac{\Gamma \vdash v : \tau[\rho_3/\rho_1]; L \qquad \Gamma \vdash \rho_3 < \rho_2}{\Gamma \vdash \texttt{pack}\ v\ \texttt{as}\ \bullet : \bot\ (\exists \rho_1 < \rho_2 . \tau); L}$$

We have

$$\begin{array}{l}(S, K, \langle \ldots, E[\texttt{pack}\ v\ \texttt{as}\ \bullet], \ldots \rangle) \longrightarrow \\ \qquad (S, K, \langle \ldots, E[\texttt{bot}\ \texttt{pack}\ v\ \texttt{as}\ \bullet], \ldots \rangle)\end{array}$$

Let $\Gamma' = \Gamma$. We have by (Packed)

$$\frac{\Gamma' \vdash \texttt{bot} : \bot \qquad \Gamma' \vdash v : \tau[\rho_3/\rho_1]; \emptyset \qquad \Gamma' \vdash \rho_3 < \rho_2}{\Gamma' \vdash \texttt{bot}\ \texttt{pack}\ v\ \texttt{as}\ \bullet : \bot\ (\exists \rho_1 < \rho_2 . \tau); \emptyset}$$

Then by Lemma 1 and (Subsumption),

$$\Gamma' \vdash \texttt{bot}\ \texttt{pack}\ v\ \texttt{as}\ \bullet : \bot\ (\exists \rho_1 < \rho_2 . \tau); L$$

Hence by Lemma 4, it follows that

$$\Gamma' \vdash (S, K, \langle \ldots, E[\texttt{bot}\ \texttt{pack}\ v\ \texttt{as}\ \bullet], \ldots \rangle)$$

[Open] Suppose

$$\Gamma \vdash (S, K, \langle \ldots, E[\texttt{access } A \texttt{ in } R[\\ \texttt{open } x = (k \texttt{ pack } v \texttt{ as } \bullet) \texttt{ as } \bullet \texttt{ in } e]], \ldots \rangle)$$

So we have

$$\Gamma \vdash E[\texttt{access } A \texttt{ in } R[\texttt{open } x = (k \texttt{ pack } v \texttt{ as } \bullet) \texttt{ as } \bullet \texttt{ in } e]] : -; \emptyset$$

By inspection of the type checking rules, there must be a sub-derivation with (Open $\bullet$) as the last rule.

$$\frac{\begin{array}{c}\Gamma \vdash (k \texttt{ pack } v \texttt{ as } \bullet) : \rho_1\ (\exists \rho_2 < \rho_3.\tau_1); L_1 \\ \Gamma, \rho_2 < \rho_3, x : \tau_1 \vdash e : \tau_2; L_2 \\ \Gamma \vdash \tau_2 \qquad \Gamma \vdash L_2 \qquad \Gamma, \rho_2 < \rho_3, x : \tau_1 \vdash \diamond\end{array}}{\Gamma \vdash \texttt{open } x = (k \texttt{ pack } v \texttt{ as } \bullet) \texttt{ as } \bullet \texttt{ in } e : \tau_2; L_1 \cup L_2 \cup \{\rho_1\}}$$

We first show that

$$(S, K, \langle \ldots, E[\texttt{access } A \texttt{ in } R[\\ \texttt{open } x = (k \texttt{ pack } v \texttt{ as } \bullet) \texttt{ as } \bullet \texttt{ in } e]], \ldots \rangle) \\ \longrightarrow (S, K, \langle \ldots, E[\texttt{access } A \texttt{ in } R[e[v/x]]], \ldots \rangle)$$

It suffices to show that $k \in Close_K(A)$. By Lemma 5 and (InAccess), it must be the case that

$$\Gamma \vdash \{\rho_1\} < \{\rho \mid \Gamma \vdash k : \rho \text{ for some } k \in A\}$$

By inspection of the type checking rules, we see that there must be ρ_1' such that

$$\Gamma \vdash k : \rho_1' \quad \Gamma \vdash \rho_1' < \rho_1$$

So by Definition 2 (7), $k \in Close_K(A)$.

Now let $\Gamma' = \Gamma$. We next show that

$$\Gamma' \vdash (S, K, \langle \ldots, E[\texttt{access } A \texttt{ in } R[e[v/x]]], \ldots \rangle)$$

By inspection of the type checking rules, we see that the sub-derivation of the first premise must end with (Packed) followed by zero or more (Subsumption)'s. That is, for some ρ_4

$$\begin{array}{c}\Gamma' \vdash v : \tau_1'[\rho_4/\rho_2]; \emptyset \\ \Gamma' \vdash \rho_4 < \rho_3' \quad \Gamma' \vdash \rho_3' < \rho_3 \\ \Gamma', \rho_2 < \rho_3' \vdash \tau_1' < \tau_1\end{array}$$

So we have by Lemma 2 (3)

$$\Gamma' \vdash \tau_1'[\rho_4/\rho_2] < \tau_1[\rho_4/\rho_2]$$

so

$$\Gamma' \vdash v : \tau_1[\rho_4/\rho_2]; \emptyset$$

Furthermore, because $\Gamma' \vdash \tau_2$, $\Gamma' \vdash L_2$ and $\Gamma' \vdash \rho_4 < \rho_2$, we have by Lemma 2 (4)

$$\Gamma', x : \tau_1[\rho_4/\rho_2] \vdash e : \tau_2; L_2$$

Hence by Lemma 2 (1)

$$\Gamma' \vdash e[v/x] : \tau_2; L_2$$

Finally by Lemma 4 it follows that

$$\Gamma' \vdash (S, K, \langle \ldots, E[\texttt{access } A \texttt{ in } R[e[v/x]]], \ldots \rangle)$$

[Limit] Suppose

$$\Gamma \vdash (S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[\\ \texttt{limit}\ k'_1\ \texttt{lkey}\,(k_1), \ldots, k'_n\ \texttt{lkey}\,(k_n)\ \texttt{in}\ e]], \ldots \rangle)$$

So we have

$$\Gamma \vdash E[\texttt{access}\ A\ \texttt{in} R[\\ \texttt{limit}\ k'_1\ \texttt{lkey}\,(k_1), \ldots, k'_n\ \texttt{lkey}\,(k_n)\ \texttt{in}\ e]] : -; \emptyset$$

By inspection of the type checking rules, there must be a sub-derivation with (Limit) as the last rule.

$$\frac{\begin{array}{c}\text{For } 1 \leq i \leq n,\ \Gamma \vdash k'_i\ \texttt{lkey}\,(k_i) : \rho'_i\ \mathit{lkey}\,(\rho_i); L_i \\ \Gamma \vdash e : \tau; L \qquad\qquad \Gamma \vdash L < \{\rho_1, \rho_2, \ldots, \rho_n\}\end{array}}{\begin{array}{r}\Gamma \vdash \texttt{limit}\ k'_1\ \texttt{lkey}\,(k_1), \ldots, k'_n\ \texttt{lkey}\,(k_n)\ \texttt{in}\ e : \\ \tau; L \cup \bigcup_{1 \leq i \leq n-1}(L_i \cup \{\rho'_i\})\end{array}}$$

Let $B = \mathit{Close}_K(A) \cap \mathit{Close}_K(\{k_1, k_2, \ldots, k_n\})$. We first show that

$$(S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[\\ \texttt{limit}\ k'_1\ \texttt{lkey}\,(k_1), \ldots, k'_n\ \texttt{lkey}\,(k_n)\ \texttt{in}\ e]], \ldots \rangle) \longrightarrow \\ (S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[\texttt{access}\ B\ \texttt{in}\ e]], \ldots \rangle)$$

It suffices to show that for $1 \leq i \leq n$, $k'_i \in \mathit{Close}_K(A)$. By Lemma 5 and (InAccess), it must be the case that

$$\Gamma \vdash \{\rho'_i\} < \{\rho \mid \Gamma \vdash k : \rho \text{ for some } k \in A\}$$

By inspection of the type checking rules, we see that there must be ρ''_i such that

$$\Gamma \vdash k'_i : \rho''_i \quad \Gamma \vdash \rho''_i < \rho'_i$$

So by Definition 2 (7), $k'_i \in \mathit{Close}_K(A)$.

Now let $\Gamma' = \Gamma$. We next show that

$$\Gamma' \vdash (S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[\texttt{access}\ B\ \texttt{in}\ e]], \ldots \rangle)$$

Firstly, by Lemma 5

$$\Gamma' \vdash L < \{\rho \mid \Gamma \vdash k : \rho \text{ for some } k \in A\}$$

Hence by $\Gamma' \vdash L < \{\rho_1, \rho_2, \ldots, \rho_n\}$ and Definition 2 (7), it follows that

$$\Gamma' \vdash L < \{\rho \mid \Gamma \vdash k : \rho \text{ for some } k \in B\}$$

(Since subtyping is invariant for limit key types.) Then by Lemma 1 and (Subsumption),

$$\Gamma' \vdash \texttt{access}\ B\ \texttt{in}\ e : \tau; L \cup \bigcup_{1 \leq i \leq n-1} L_i \cup \{\rho'_i\}$$

Hence by Lemma 4, it follows that

$$\Gamma' \vdash (S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[\texttt{access}\ B\ \texttt{in}\ e]], \ldots \rangle)$$

[Grant] Suppose

$$\Gamma \vdash (S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[\texttt{grant}\ k_1\ \texttt{gkey}\,(k_2)\ \texttt{in}\ e]], \ldots \rangle)$$

So we have $\Gamma \vdash E[\texttt{access}\ A\ \texttt{in}\ R[\texttt{grant}\ k_1\ \texttt{gkey}\,(k_2)\ \texttt{in}\ e]] : -; \emptyset$. By inspection of the type checking rules, there must be a sub-derivation with (Grant) as the last rule.

$$\frac{\begin{array}{c}\Gamma \vdash k_1\ \texttt{gkey}\,(k_2) : \rho_1\ \mathit{gkey}\,(\rho_2); L_1 \\ \Gamma \vdash e : \tau; L_2 \cup \{\rho_2\}\end{array}}{\Gamma \vdash \texttt{grant}\ k_1\ \texttt{gkey}\,(k_2)\ \texttt{in}\ e : \tau; L_1 \cup L_2 \cup \{\rho_1\}}$$

We first show that

$$(S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[\texttt{grant}\ k_1\ \texttt{gkey}\ (k_2)\ \texttt{in}\ e]], \ldots \rangle) \longrightarrow \\ (S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[\texttt{access}\ A \cup \{k_2\}\ \texttt{in}\ e]], \ldots \rangle)$$

It suffices to show that $k_1 \in Close_K(A)$. By Lemma 5 and (InAccess), it must be the case that

$$\Gamma \vdash \{\rho_1\} < \{\rho \mid \Gamma \vdash k : \rho \text{ for some } k \in A\}$$

By inspection of the type checking rules, we see that there must be ρ' such that

$$\Gamma \vdash k_1 : \rho' \quad \Gamma \vdash \rho' < \rho_1$$

So by Definition 2 (7), $k_1 \in Close_K(A)$.

Now let $\Gamma' = \Gamma'$. We next show that

$$\Gamma' \vdash (S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[\texttt{access}\ A \cup \{k_2\}\ \texttt{in}\ e]], \ldots \rangle)$$

By Lemma 5

$$\Gamma' \vdash L < \{\rho \mid \Gamma \vdash k : \rho \text{ for some } k \in A\}$$

Therefore by Definition 2 (7), it follows that

$$\Gamma' \vdash L \cup \{\rho_2\} < \{\rho \mid \Gamma \vdash k : \rho \text{ for some } k \in A \cup \{k_2\}\}$$

(Since subtyping is invariant for grant key types.) Then by Lemma 1 and (Subsumption),

$$\Gamma' \vdash \texttt{access}\ A \cup \{k_2\}\ \texttt{in}\ e : \tau; L_1 \cup L_2 \cup \{\rho_2\}$$

Hence by Lemma 4, it follows that

$$\Gamma' \vdash (S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[\texttt{access}\ A \cup \{k_2\}\ \texttt{in}\ e]], \ldots \rangle)$$

[NewKey] Suppose

$$\Gamma \vdash (S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[\texttt{newkey<}k_1\ \texttt{lkey}\ (k_2)]], \ldots \rangle)$$

So we have $\Gamma \vdash E[\texttt{access}\ A\ \texttt{in}\ R[\texttt{newkey<}k_1\ \texttt{lkey}\ (k_2)]] : -; \emptyset$. By inspection of the type checking rules, there must be a sub-derivation with (NewKey) as the last rule.

$$\frac{\Gamma \vdash k_1\ \texttt{lkey}\ (k_2) : \rho_1\ \mathit{lkey}\ (\rho_2); L}{\begin{array}{l}\Gamma \vdash \texttt{newkey<}k_1\ \texttt{lkey}\ (k_2) : \\ \quad \bot\ (\exists \rho_3 < \rho_2 . \bot\ \langle \bot\ \mathit{lkey}\ (\rho_3), \bot\ \mathit{gkey}\ (\rho_3) \rangle); L \cup \{\rho_1\}\end{array}}$$

We first show that

$$\begin{array}{l}(S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[\texttt{newkey<}k_1\ \texttt{lkey}\ (k_2)]], \ldots \rangle) \longrightarrow \\ (S, K \cup \{k_3 \mapsto k_2\}, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[\\ \quad \texttt{bot pack bot}\ \langle \texttt{bot lkey}\ (k_3), \texttt{bot gkey}\ (k_3) \rangle\ \texttt{as}\ \bullet]], \ldots \rangle)\end{array}$$

for some $k_3 \notin dom(K) \cup \{\texttt{bot}, \texttt{top}\}$. It suffices to show that $k_1 \in Close_K(A)$. By Lemma 5 and (InAccess), it must be the case that

$$\Gamma \vdash \{\rho_1\} < \{\rho \mid \Gamma \vdash k : \rho \text{ for some } k \in A\}$$

By inspection of the type checking rules, we see that there must be ρ' such that

$$\Gamma \vdash k_1 : \rho' \quad \Gamma \vdash \rho' < \rho_1$$

So by Definition 2 (7), $k_1 \in Close_K(A)$.

Pick $\rho_3 \notin dom(\Gamma) \cup \{\bot, \top\}$. Let $\Gamma' = \Gamma, \rho_3 {<} \rho_2, k{:}\rho_3$. We next show that

$$\begin{array}{l}\Gamma' \vdash (S, K \cup \{k_3 \mapsto k_2\}, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[\\ \quad \texttt{bot pack bot}\ \langle \texttt{bot lkey}\ (k_3), \texttt{bot gkey}\ (k_3)\rangle\ \texttt{as}\ \bullet]], \ldots\rangle)\end{array}$$

By (Packed) and (Tuple'ed), we have

$$\begin{array}{l}\Gamma' \vdash \texttt{bot pack bot}\ \langle \texttt{bot lkey}\ (k_3), \texttt{bot gkey}\ (k_3)\rangle\ \texttt{as}\ \bullet : \\ \quad \bot\ (\exists \rho_3 < \rho_2 . \bot\ \langle \bot\ \mathit{lkey}\ (\rho_2), \bot\ \mathit{gkey}\ (\rho_2)\rangle); \emptyset\end{array}$$

Hence by Lemma 1 and (Subsumption),

$$\begin{array}{l}\Gamma' \vdash \texttt{bot pack bot}\ \langle \texttt{bot lkey}\ (k_3), \texttt{bot gkey}\ (k_3)\rangle\ \texttt{as}\ \bullet : \\ \quad \bot\ (\exists \rho_3 < \rho_2 . \bot\ \langle \bot\ \mathit{lkey}\ (\rho_2), \bot\ \mathit{gkey}\ (\rho_2)\rangle); L \cup \{\rho_1\}\end{array}$$

Also, it is easy to see that Definition 2 (4) and (7) is satisfied. Hence

$$\begin{array}{l}\Gamma' \vdash (S, K \cup \{k_3 \mapsto k_2\}, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[\\ \quad \texttt{bot pack bot}\ \langle \texttt{bot lkey}\ (k_3), \texttt{bot gkey}\ (k_3)\rangle\ \texttt{as}\ \bullet]], \ldots\rangle)\end{array}$$

[Associate] Suppose

$$\begin{array}{l}\Gamma \vdash (S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[\\ \qquad \texttt{associate}\ k_1\ r\ \texttt{with}\ k_2\ \texttt{lkey}\ (k_3)]], \ldots\rangle)\end{array}$$

So we have

$$\Gamma \vdash E[\texttt{access}\ A\ \texttt{in}\ R[\texttt{associate}\ k_1\ r\ \texttt{with}\ k_2\ \texttt{lkey}\ (k_3)]] : -; \emptyset$$

By inspection of the type checking rules, there must be a sub-derivation with (Associate) as the last rule.

$$\frac{\Gamma \vdash k_1\ r : \rho_1\ \sigma; L_1 \qquad \Gamma \vdash k_2\ \texttt{lkey}\ (k_3) : \rho_2\ \mathit{lkey}\ (\rho_3); L_2}{\Gamma \vdash \texttt{associate}\ k_1\ r\ \texttt{with}\ k_2\ \texttt{lkey}\ (k_3) : \rho_3\ \sigma; L_1 \cup L_2 \cup \{\rho_1, \rho_2\}}$$

We first show that

$$\begin{array}{l}(S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[\\ \quad \texttt{associate}\ k_1\ r\ \texttt{with}\ k_2\ \texttt{lkey}\ (k_3)]], \ldots\rangle) \longrightarrow \\ \qquad (S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[k_3\ r]], \ldots\rangle)\end{array}$$

It suffices to show that $k_1 \in \mathit{Close}_K(A)$ and $k_2 \in \mathit{Close}_K(A)$. By Lemma 5 and (InAccess), it must be the case that

$$\Gamma \vdash \{\rho_1\} < \{\rho \mid \Gamma \vdash k : \rho \text{ for some } k \in A\}$$

By inspection of the type checking rules, we see that there must be ρ' such that

$$\Gamma \vdash k_1 : \rho' \quad \Gamma \vdash \rho' < \rho_1$$

So by Definition 2 (7), $k_1 \in \mathit{Close}_K(A)$. The case for k_2 is analogous.

Now let $\Gamma' = \Gamma$. We next show that

$$\Gamma' \vdash (S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[k_3\ r]], \ldots\rangle)$$

The cases split between (Int'ed), (Loc), (Lkey), (Gkey), (Tuple'ed), (Fun'ed), (Gen'ed), and (Packed) depending on σ, but in each case we have

$$\Gamma' \vdash k_3\ r : \rho_3\ \sigma; \emptyset$$

(Since subtyping is invariant for limit key types.) Then by Lemma 1 and (Subsumption),

$$\Gamma' \vdash k_3\ r : \rho_3\ \sigma; L_1 \cup L_2 \cup \{\rho_1, \rho_2\}$$

Hence by Lemma 4, it follows that

$$\Gamma' \vdash (S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ R[k_3\ r]], \ldots\rangle)$$

[InAccess] Suppose

$$\Gamma \vdash (S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ v], \ldots \rangle)$$

So we have $\Gamma \vdash E[\texttt{access}\ A\ \texttt{in}\ v] : -; \emptyset$. By inspection of the type checking rules, there must be a sub-derivation with (InAccess) as the last rule.

$$\frac{\Gamma \vdash v : \tau; L \qquad \Gamma \vdash L < \{\rho \mid \Gamma \vdash k : \rho \text{ for some } k \in A\}}{\Gamma \vdash \texttt{access}\ A\ \texttt{in}\ v : \tau; \emptyset}$$

We have

$$(S, K, \langle \ldots, E[\texttt{access}\ A\ \texttt{in}\ v], \ldots \rangle) \longrightarrow (S, K, \langle \ldots, E[v], \ldots \rangle)$$

Let $\Gamma' = \Gamma$. Then by Lemma 6,

$$\Gamma \vdash v : \tau; \emptyset$$

Hence by Lemma 4, it follows that

$$\Gamma' \vdash (S, K, \langle \ldots, E[v], \ldots \rangle)$$

This concludes the proof of Theorem 3. □

Let e be a source program. We say that e gets stuck if it cannot be translated to the target language or if its evaluation reaches a state of the form $(S, K, \langle \ldots, err, \ldots \rangle)$.

Corollary 1 (Type Soundness) *If the program e is well-typed then it does not get stuck.*

Proof: Recall that each source program e is well-typed iff $\emptyset \vdash e : -; \emptyset$. The corollary follows from Theorem 2 and Theorem 3 since the initial state $(\emptyset, \emptyset, \langle e \rangle)$ is well-typed under the type environment $\emptyset$. □ The corollary implies that a well-typed program does not violate its access control policy.